FAMA Framework
FAMA Extensions Development Guide

Revision History			
0.1	June 2014	Creation	Andrés Paz, I2T Research Group, Icesi University
1.0	July 2014	First release	Andrés Paz, I2T Research Group, Icesi University

The first version of the work described here was performed in the context of the FAMA Framework project and the SHIFT project during the stay at University of Seville (Seville, Spain), partly funded by Banco Santander under the program Becas Iberoamérica Jóvenes Profesores Investigadores Colombia and Colciencias (Colombian Administrative Department of Science, Technology and Innovation).

This guide is designed to get you up and running for developing extensions to the FAMA Framework and, as an example, it shows the process of developing and integrating a new reasoner. It is organized as follows: Section 1 presents prerequisites; Section 2 provides an overview of the FAMA Framework, an overview of the FAMA Extensions layer, an overview of the FAMA Reasoners and gives details about the Choco library for constraint satisfaction problems and constraint programming; Section 3 explains how to set up the development environment, which includes how to check out the source code from the repository and import it into the IDE as well as common problems while building the source code; Section 4 explains how to create a new FAMA Extension and particularly how to develop a new reasoner; Section 5 shows how to package and test a FAMA Extension; and Section 6 describes how to check in a FAMA Extension developed from scratch and changes made to an existing one.

1. Prerequisites

It is recommended for you to have a minimal knowledge and experience in the following:

- Software development
- Software engineering, including source configuration management, preferably Git)
- Java and several third-party development tools like Maven and JUnit
- Eclipse IDE and plug-in development
- Software Product Lines (SPL) Engineering

2. Overview of the FAMA Framework

FAMA (FeAture Model Analyzer) is an open source Java framework under LGPLv3 license for the automated analysis of feature models. It provides useful information extracted from these models, such as the number of products that can be derived from them, and possible errors they may have.

FAMA is developed as OSGi plug-ins, which allow an easier development and assembly. The framework has three layers, as shown in Figure 1, namely public interfaces, SPL Core and FAMA Extensions. The SPL Core provides the necessary services to allow interoperation between the

extension modules but keeping them uncoupled. The extensions layer is composed of various modules, such as metamodels, operations and reasoners to name a few.

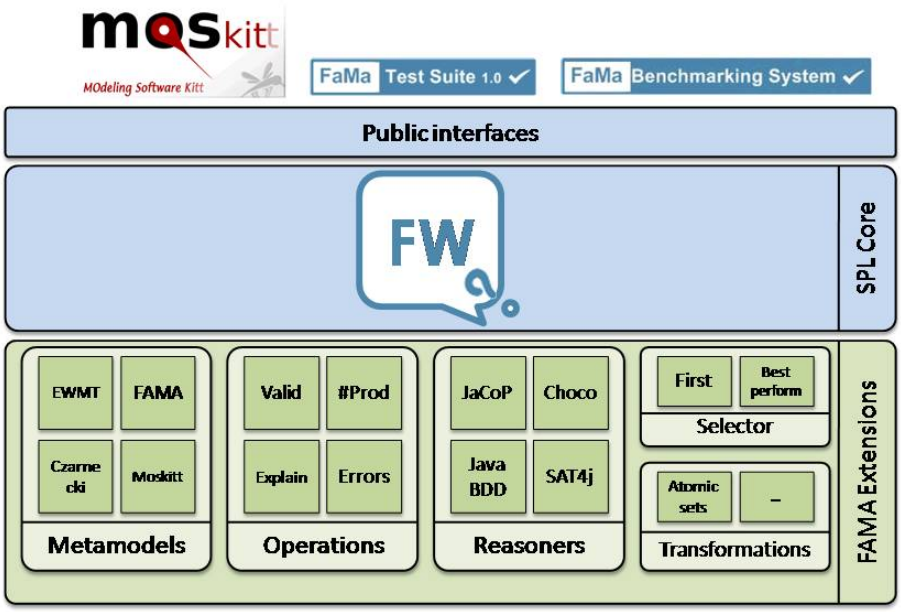

Figure 1. FAMA Framework architecture

2.1. Overview of feature models

A feature model is a hierarchical tree structure representing all the products that can be derived from a Software Product Line (SPL) in terms of "features". Features are distinctive characteristics of a software system. Every feature model comprises features and the relationships among them. The first feature in the feature model, called the root feature, is used to identify the SPL and should always be present in every product.

The remaining features emerge from this root feature in a hierarchical tree structure. Relationships between features may be of two types. On the one hand, parental relationships, i.e. between a parent feature and its child features. Parental relationships can be categorized as mandatory, optional or set. On the other hand, non-parental relationships, also known as cross-tree constraints. The most common cross-tree constraints are requires and excludes. All these features are defined as follows:

- **Mandatory:** A mandatory relationship states that if a parent feature is present in a product its child feature must be present too.
- **Optional:** An optional relationship states that if a parent feature is present in a product its child feature may or may not be present.
- **Set:** A set of child features has a set relation with their parent when a number of them can be included in the products in which its parent feature appears. The amount of child features that may be included is given by the relation cardinality. The relation cardinality is expressed as a range of values [x,y], where $x \leq y$ and $y \leq$ number of child features. Two special cases of set relations can be identified: Alternative and Or relations.

- ***Alternative:*** A set of child features have an Alternative relation with their parent when only one child can be present when its parent feature is present in the product. In this case the relation cardinality can be expressed as [0,1].
- ***Or:*** A set of child features have an Or relation with their parent when one or more of them can be present when its parent feature is present in the product. In this case the relation cardinality can be expressed as [0,n], where n is the number of child features.

– **Requires:** A requires relationship is a cross tree constraint that states that if feature A requires feature B then if feature A is present in a product, feature B must be present too.
– **Excludes:** An excludes relationship is a cross tree constraint that states that if feature A excludes feature B then features A and B can not be present at the same time in a product.

2.2. Overview of the FAMA Extensions Layer

Since the FAMA Framework is component-oriented, it allows for its easy extension to functionality and feature model support through new components that are known as FAMA Extensions. This FAMA Extensions will live in the FAMA Extensions Layer (see Figure 1). Currently the FAMA Framework includes several FAMA Extensions that provide support for creating feature models conforming to various feature metamodels, operations over feature models, reasoners, and model transformations. Metamodel extensions provide different approaches to represent a feature model; supported metamodels are Czarnecki's cardinality-based feature models[1], Moskitt[2] and an own FAMA metamodel. Operation extensions specify concerns that arise when handling feature models and that need to be solved, such as what products can be derived from a given feature model or are there any errors present in the feature model. Operation extensions are used closely with reasoner extensions. A reasoner is meant to execute the operations over feature models and obtain the required information or properties from it. There are different techniques to perform these operations, and the constraint satisfaction problem technique is one of them. Reasoner extensions are usually implemented to use a specific technique, for example through out this guide we will be developing a new reasoner to support the analysis of feature models represented through constraint satisfaction problems; following sections will explain more about what constraint satisfactions problems are and how are they used to build a representation of feature models. Anyone can create a FAMA Extension that extends the framework in one of these aspects or create one for an entirely new purpose. For more detail on the FAMA Extensions available please refer to [2].

2.3. Overview of a Constraint Satisfaction Problem representation of feature models

A Constraint Satisfaction Problem (CSP) is defined as a three-tuple composed of a set of problem variables, a set of domain values for each of the variables, and a set of constraints over those variables that restrict the values they can take. A CSP is solved by the assignment of a value to every variable in its domain through search and propagation in such way that all constraints are

[1] http://gsd.uwaterloo.ca/fmp
[2] http://www.moskitt.org/eng/que-es-moskitt/

simultaneously satisfied. A feature model can be mapped to a CSP. The features from the feature model will be represented as variables with a domain equivalent to its specific cardinality. The representations for the relationships are as follows (more detail about these representations can be found in [1]):

- **Mandatory:** A mandatory relation between a parent feature and its child can be represented with the logical expression $parent\ present \Leftrightarrow child\ present$.
- **Optional:** An optional relation between a parent feature and its child can be represented with the logical expression $parent\ not\ present \Rightarrow child\ not\ present$.
- **Set:** A set constraint between a parent feature and its children with cardinality $[x, y]$ can be represented with the logical expression $(parent\ present\ \Rightarrow x\ \leq \sum children\ present \leq y)\ \wedge (parent\ not\ present \Rightarrow children\ not\ present)$, equivalent to an if-then-else. Both the Alternative and Or cases are handled by this expression.
- **Requires:** A Requires relation between an origin feature and its destination can be represented with the logical expression $origin\ present \Rightarrow destination\ present$.
- **Excludes:** An Excludes relation between an origin feature and its destination can be represented with the logical expression $\neg(origin\ present \wedge destination\ present)$ that can also be expressed as $origin\ present \Rightarrow destination\ not\ present$.

A product that can be derived from the feature model represented as a CSP is, thus, one of the possible solutions to the CSP.

2.4. Overview of Choco and Choco 3

In order to solve CSPs we need to use one of the many tools that have been developed for this purpose. Since the FAMA Framework is built on top of Java and all extensions are encouraged to build using Java we have chosen to work with Choco for the development of our new reasoner. Choco is an open source Java library developed for constraint satisfaction problems (CSP) and constraint programming (CP). It is meant to be used for teaching and research purposes, although several industrial companies use it.

Choco 3 [5] is the newest and current release to date of Choco, however, it breaks all backwards compatibility with previous versions of the tool as it was completely rewritten from the ground up. This new version of Choco integrates various types of variables (e.g. integer, Boolean, set, graph, real), various constraints (natively supports explained contraints) (e.g. classical arithmetic constraints, alldifferent, count, nvalues), various search strategies (e.g. first fail, smallest, impact-based, activity-based), and an explanation-based engine. However, the Choco 3 library no longer provides native support for biconditional expressions, so to represent the mandatory relationship with Choco 3 an equivalent expression needs to be used. This expression is $(parent\ present\ \Rightarrow\ child\ present) \wedge (child\ present\ \Rightarrow\ parent\ present)$.

3. Setting up the Development Environment

This guide was developed and tested for FAMA Framework 1.2 using Eclipse Modeling Tools IDE Kepler Release on OS X 10.9 with Java 1.7 and the most recent release of the necessary plug-ins as of May 2014. All of the requirements are summarized in Table 1.

You may use any package solution of the Eclipse IDE; just make sure to grab the correct 32-bit or 64-bit version depending on your machine. It is also suggested to use JDK 5 or newer.

Before getting started, you will need to install a couple Eclipse plug-ins to help with development. You can use the normal Eclipse process to install these plug-ins through the menu option *Help » Install New Software…*

- A Git plugin (*e.g.* EGit) (Update site: http://download.eclipse.org/releases/kepler/ under the *Collaboration* category) or, if you prefer, a standalone Git client (*e.g.* SourceTree)
- The Eclipse Maven plug-ins m2e (Maven Integration for Eclipse) and m2e - slf4j (over logback logging) (Update site: http://download.eclipse.org/releases/kepler/ under the *Collaboration* category)
- ANTLR UI (Update site: http://antlreclipse.sourceforge.net/updates/)
- ANTLR plugin for Eclipse 2.7.6. Note: ANTLR plugin for Eclipse 2.7.6 is not available through an Eclipse update site. You must download it from http://sourceforge.net/projects/antlreclipse/ and manually install the jar by placing it in the plugins folder found in your Eclipse's installation folder.

Table 1. Summary of required software

Requirement	Version	URL
Java Development Kit	5 or newer	http://www.oracle.com/technetwork/java/javase/downloads/index.html
Eclipse	4.3 or newer	https://www.eclipse.org/downloads/ (Download the correct 32-bit or 64-bit version depending on your operating system)
EGit	3.3.2 or newer	http://download.eclipse.org/releases/kepler/ (Eclipse update site under the *Collaboration* Category)
Maven Integration for Eclipse	1.4.0	http://download.eclipse.org/releases/kepler/ (Eclipse update site under the *Collaboration* Category select both m2e and m2e - slf4j)
ANTLR UI	4.1.1	http://antlreclipse.sourceforge.net/updates/ (Eclipse update site)
ANTLR	2.7.6	http://sourceforge.net/projects/antlreclipse/ (Download and manually install the jar by placing it in the plugins folder found in your Eclipse's installation folder)

3.1. Checking out the FAMA Framework source code from GitHub using EGit

1. Switch to the *Git* perspective in Eclipse.
2. Click the button *Clone a Git Repository* and type the URL https://github.com/ISA-Research-Group/FaMA. It is strongly recommended you fork the project from this URL

and check out the code from the fork. This will make it easier to contribute changes back to the main project later. To create a fork you need to sign up (create an account) or sign in to GitHub. After that you can go back to the FAMA Framework's GitHub page and click on the Fork button at the top right corner. Then in Eclipse's *Clone a Git Repository* wizard type the URL of your fork.

3. The *Host* and *Repository path* fields should populate automatically. Click *Next >*.
4. Keep the *develop* branch ticked and untick the other checkboxes. Click *Next >*. (The master branch is for release purposes only and doesn't contain additional source files required for development)
5. Make any changes you wish in the *Local Destination* dialogue and click *Finish*.
6. Wait for the *FAMA* repository to be cloned.

3.2. Importing the FAMA Framework source code into Eclipse

7. Right click the FAMA repository and select the option *Import Maven Projects…*
8. Select the projects you want to import. Click *Next >*. For example, for the development of a new reasoner you would have to select the projects: *FaMaAttributedModel*, *FaMaFeatureModel*, *FaMaSDK*, *FaMa-Docs*, FaMaTestSuite and *Benchmark*. The last two will be used for testing purposes.
9. Maven's dependency copy goal is not supported by the m2e plug-in, so set the action from *Resolve Later* to *Do Not Execute (add to pom)*. Click *Finish*.
10. Wait for the projects to be imported and the indexing to finish.

If you already checked out the FAMA Framework source code using an external tool, such as SourceCode, you can import it into Eclipse under the menu option *File » Import*.

11. Browse to *General* and select the *Maven Projects* element. Click *Next >*.
12. Browse to the root folder of the source code and then select all the projects of interest.

When everything works correctly, you should see each module checked out as an individual Eclipse project in your workspace. Note that on the initial checkout, it takes a while longer to initially build all the projects in your workspace. In the background, Maven is downloading all project dependencies.

3.3. Common problems building the source code

When Maven finishes downloading and resolving all project dependencies you might encounter several problems. You will have to review the projects one by one and fix the errors.

The typical problems involve missing referenced libraries. To solve these errors, go to the lib folder of the project under the project's root folder (note: in some projects it is located under a resources folder), select all the libraries in the folder, right click and select the option *Build Path » Configure Build Path…*

For those projects involving unit tests you need to add a JUnit library to the classpath. The Eclipse distribution comes with a bundled JUnit library.

13. Hover the mouse pointer over an error in a Java class that imports the JUnit library.

14. Select the option *Fix project setup…*
15. Select the option *Add JUnit library to the build path*. Click *OK*.

Depending on your JDK version you will get errors in the project *FaMaAttributedModel* and *FAMAFeatureModel* regarding the *@Override* annotation. As of JDK 6 this annotation's behavior changed and it is only necessary when implementing inherited abstract methods from a superclass. You can safely comment the line to fix the problem.

For the projects *FaMaAttributedModel* and *FAMAFeatureModel* you must toggle the ANTLR project nature. Select both projects, right click over one of them and select the option *Toggle ANTLR project nature*. Eclipse should proceed to rebuild these projects.
For the project *FaMaAttributedModel* it is required to recompile some ANTLR files.

16. Go to the *FaMaAttributedModel* project and expand the package *es.us.isa.FAMA.parser*.
17. Right click the files *Analex.g*, *Anasint.g* and *TreeParser.g* and select the option *Compile ANTLR Grammar*. Note: You must do this on a per file basis. Eclipse should proceed to rebuild the project.

4. Creating a New FAMA Extension

Before getting started, you will need to create a new plug-in project in your workspace and set it to use Maven. You can use the normal Eclipse process to create projects through the menu option *File » New » Project…*

18. Select the option *Plug-in Project*. Click *Next >*.
19. Give the project a name and select as path the path to where the other FAMA project were checked out. In this case we will be adding a new reasoner, CHOCO 3, as a new FAMA Extension so the project's name is *Choco3Reasoner*. Click *Next >*.
20. Keep the option *Generate an Activator* ticked and in the *Activator* field set the activator's canonical class name (full name including package) to the following naming pattern: *«2-letter country code».«University/Institution».«research group».«project name».osgi.Activator*. For the Choco 3 reasoner it is: *co.icesi.i2t.Choco3.osgi.Activator*. Click *Finish*. This class is used to connect the FAMA Extension to the FAMA Framework and will be edited once the FAMA Extension you develop is finished.
21. Right click the newly created plug-in project and select the option *Configure » Convert to Maven Project*.
22. A dialog will pop-up prompting to create a new Maven POM file. Fill out the form with the following information and click *Finish*.
    - Group Id: *es.us.isa*
    - Version: *1.0.0*
    - Packaging: *bundle*
    - Name: *«give the artifact a name»*. For example, for the Choco 3 reasoner it is: *FaMa Choco 3 Reasoner*.
    - Description: *«give a description for the artifact»*. For example, for the Choco 3 reasoner it is: *A reasoner for FaMa, based on Choco Solver v3. Built by I2T Research Group in collaboration with the FaMa Team*.

23. Maven should open the pom file. Switch to the *pom.xml* tab. Note: Eclipse may mark the line *<packaging>bundle</packaging>* in error. This will go away with the following steps.
24. After the *<description></description>* element and before the closing *</project>* tag paste the following *pom.xml* fragments:
    a. Plug-in repositories (see Listing 1)

```
<pluginRepositories>
	<pluginRepository>
		<id>mc-release</id>
		<name>Local Maven repository of releases</name>
		<url>http://mc-repo.googlecode.com/svn/maven2/releases</url>
		<snapshots>
			<enabled>false</enabled>
		</snapshots>
		<releases>
			<enabled>true</enabled>
		</releases>
	</pluginRepository>
</pluginRepositories>
```

Listing 1. pluginRepositories tag

    b. Distribution management (see Listing 2)

```
<distributionManagement>
	<repository>
		<id>isa.devel</id>
		<name>ISA archiva</name>

<url>http://devel.isa.us.es/glassfish/archiva/repository/internal</url>
	</repository>
</distributionManagement>
```

Listing 2. distributionManagement tag

    c. Repositories (see Listing 3). Notice the last repository marked in red; it is specific to the development of the Choco 3 reasoner. You should replace this repository with the ones concerning your particular FAMA extension if needed; otherwise you can safely remove those lines.

```
<repositories>
	<repository>
		<id>isa.devel</id>
		<name>ISA archiva</name>
		<url>http://devel.isa.us.es/glassfish/archiva</url>
		<layout>default</layout>
		<releases>
			<enabled>true</enabled>
			<updatePolicy>always</updatePolicy>
			<checksumPolicy>warn</checksumPolicy>
		</releases>
		<snapshots>
			<enabled>false</enabled>
			<updatePolicy>never</updatePolicy>
			<checksumPolicy>warn</checksumPolicy>
		</snapshots>
```

```
        </repository>
        <repository>
                <id>choco.repos</id>
                <url>http://www.emn.fr/z-info/choco-repo/mvn/repository/</url>
        </repository>
</repositories>
```

Listing 3. repositories tag

d. Dependencies (see Listing 4). Notice the first dependency marked in red; it is specific to the development of the Choco 3 reasoner. You should replace this dependency with the ones concerning your particular FAMA extension, if needed; otherwise you can safely remove those lines.

```
<dependencies>
        <dependency>
                <groupId>choco</groupId>
                <artifactId>choco-solver</artifactId>
                <version>3.2.0</version>
        </dependency>
        <dependency>
                <groupId>es.us.isa</groupId>
                <artifactId>FaMaSDK</artifactId>
                <version>1.2.0</version>
        </dependency>
        <dependency>
                <groupId>org.osgi</groupId>
                <artifactId>osgi_R4_core</artifactId>
                <version>1.0</version>
                <scope>provided</scope>
                <optional>true</optional>
        </dependency>
        <dependency>
                <groupId>org.osgi</groupId>
                <artifactId>osgi_R4_compendium</artifactId>
                <version>1.0</version>
                <scope>provided</scope>
                <optional>true</optional>
        </dependency>
</dependencies>
```

Listing 4. dependencies tag

e. Build (see Listing 5). Notice in the final lines of this fragment marked in red; they are specific to the development of the Choco 3 reasoner. You should replace them according to your particular FAMA extension.

```
<build>
        <defaultGoal>test</defaultGoal>
        <directory>target</directory>
        <sourceDirectory>src</sourceDirectory>
        <outputDirectory>target/classes</outputDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <testOutputDirectory>target/test-classes</testOutputDirectory>
        <resources>
                <resource>
                        <directory>resources</directory>
                </resource>
```

```
		</resources>
		<testResources>
			<testResource>
				<directory>resources</directory>
			</testResource>
		</testResources>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<configuration>
					<source>5</source>
					<target>5</target>
				</configuration>
			</plugin>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-jar-plugin</artifactId>
				<configuration>
					<archive>
						<!--<manifestFile>META-
INF/MANIFEST.MF</manifestFile> -->

	<addMavenDescriptor>false</addMavenDescriptor>
						<!-- <manifestEntries> <Built-By>I2T
Research Group in collaboration with FaMa Team</Built-By> </manifestEntries> -->
					</archive>
				</configuration>
			</plugin>
			<plugin>
				<groupId>com.mycila.maven-license-plugin</groupId>
				<artifactId>maven-license-plugin</artifactId>
				<configuration>
					<excludes>
						<exclude>**/**.xml/**</exclude>
					</excludes>
					<header>copyrightNotice.txt</header>
				</configuration>
				<executions>
				</executions>
			</plugin>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-source-plugin</artifactId>
				<executions>
					<execution>
						<id>jar</id>
						<phase>package</phase>
						<goals>
							<goal>jar</goal>
						</goals>
						<configuration>

	<excludeResources>true</excludeResources>

	<overWriteReleases>true</overWriteReleases>

	<overWriteSnapshots>false</overWriteSnapshots>
```

```
        <overWriteIfNewer>true</overWriteIfNewer>
                                        </configuration>
                                    </execution>
                                </executions>
                            </plugin>
                            <plugin>
                                <groupId>org.apache.felix</groupId>
                                <artifactId>maven-bundle-plugin</artifactId>
                                <extensions>true</extensions>
                                <version>1.4.0</version>
                                <configuration>
                                    <instructions>
                                        <Bundle-
SymbolicName>${pom.groupId}.${pom.artifactId}</Bundle-SymbolicName>
                                        <Bundle-Name>${pom.artifactId}</Bundle-
Name>
                                        <Built-By>I2T Research Group in
collaboration with FaMa Team</Built-By>
                                        <Export-Package>co.icesi.i2t.*</Export-
Package>
                                        <Bundle-
Activator>co.icesi.i2t.Choco3.osgi.Activator
                                        </Bundle-Activator>
                                        <Import-
Package>org.osgi.framework,*;resolution:=optional
                                        </Import-Package>
                                        <Bundle-ClassPath>lib/choco-parser-3.2.0-
jar-with-dependencies.jar, .</Bundle-ClassPath>
                                    </instructions>
                                </configuration>
                            </plugin>
                        </plugins>
        </build>
```

Listing 5. build tag

25. Download any third-party libraries used in your FAMA extension and create a folder in your project to store them. The Choco 3 library may be downloaded from the URL http://www.emn.fr/z-info/choco-repo/mvn/repository/choco/choco-parser/3.2.0/choco-parser-3.2.0-jar-with-dependencies.jar
    f. First you will need to create a new source folder. Right click over your *FAMA extension project* and select the option *New » Source folder*. Name it *resources*.
    g. Right click over the resources source folder and select the option *New » Folder*. Name this folder *lib*.
    h. Copy and paste the jar files of the third-party libraries into the *lib* folder.

4.1. Developing a New Reasoner

Every new reasoner added to the FAMA Framework should, at least, provide the implementation for handling simple feature models. If the solver used by the reasoner allows its use on extended feature models (*e.g.* attributed feature models), then an implementation for handling attributed feature models (currently the only extended feature model offered by the FAMA Framework) is strongly suggested. It is also suggested, if possible, to provide an implementation for the question ExplainErrors in both simple feature models and extended feature models. Follow these steps to develop a new reasoner. The development of the Choco 3

reasoner for handling simple feature models is provided as an example. It is suggested you use the Choco 3 reasoner source code to guide your development as it provides documentation and comments describing what the required methods should do and provide in return.

26. Each implementation should comprise 2 packages: one to store the reasoner and abstract question classes, and another one to store the various question implementations. The packages should follow the naming pattern *«2-letter country code».«University/Institution».«research group».«project name».«feature model type»*. The second package's name, the one holding the question implementations, should end with *questions*. For the Choco 3 reasoner we have the packages:
    - *co.icesi.i2t.Choco3Reasoner.simple*
    - *co.icesi.i2t.Choco3Reasoner.simple.questions*
    - *co.icesi.i2t.Choco3Reasoner.attributed*
    - *co.icesi.i2t.Choco3Reasoner.attributed.questions*
    - *co.icesi.i2t.Choco3Reasoner4Exp.simple*
    - *co.icesi.i2t.Choco3Reasoner4Exp.simple.questions*
    - *co.icesi.i2t.Choco3Reasoner4Exp.attributed*
    - *co.icesi.i2t.Choco3Reasoner4Exp.attributed.questions*
27. For each implementation, the package with the naming pattern *«2-letter country code».«University/Institution».«research group».«project name».«feature model type»* will contain the reasoner's class implementation extending from the abstract reasoner class corresponding to the feature model type it will handle. The names of these classes should follow the naming pattern *«name of reasoner»Reasoner* (using CamelCase). For example, for the simple feature model we created a class in the package *co.icesi.i2t.Choco3Reasoner.simple* and named it *Choco3Reasoner* that extends the *FeatureModelReasoner* abstract class provided by the *FaMaSDK* project in the package *es.us.isa.FAMA.Reasoner*.
    This reasoner class should define the necessary methods to create a representation of the feature model, as described in Subsections 2.3 and 2.4, that the reasoner can use to be able to provide specific information about it. The required methods to implement are:
    - **addFeature:** Creates a representation of the given feature in the reasoner.
    - **addRoot:** Marks the given feature as the root feature and ensures that it is present in every product.
    - **addMandatory:** Creates a representation of a mandatory relationship between the given parent and child features.
    - **addOptional:** Creates a representation of an optional relationship between the given parent and child features.
    - **addExcludes:** Creates a representation of an excludes relationship between the given origin and destination features.
    - **addRequires:** Creates a representation of a requires relationship between the given origin and destination features.
    - **addSet:** Creates a representation of a set relationship between the given parent and its children features.
    - **applyStagedConfiguration:** Applies a staged configuration to the representation of the feature model in the reasoner.

- **unapplyStagedConfiguration:** Removes the staged configuration applied with the previous method from the representation of the feature model in the reasoner.
- **ask:** Provides an answer to a given question about specific information of the feature model, for example the number of products that can be derived from a given feature model. A question represents an analysis operation, like the one in the example, over a given feature model.

Note: For the reasoners in the packages of ExplainErrors (4Exp) you must inherit from the respective implementation's reasoner created previously in this step and not from the abstract reasoner class provided by the *FaMaSDK* project.

28. Also, the package from step 27 will also contain an abstract class implementing the interface *Question* provided by the *FaMaSDK* project in the package *es.us.isa.FAMA.Reasoner*. This class will define the required behavior that every question should implement. It is suggested that this abstract class defines a three-phased answering process to provide an answer. The phases should be: pre-answer, answer, and post-answer. The pre-answer phase should prepare the reasoner and other resources needed prior to answering the question. The answer phase should answer the question using the given reasoner. And, finally, the post-answer phase should release any resources associated with answering the question. The name of this class should follow the naming pattern *«name of reasoner»Question* (using CamelCase). For example, for the simple feature model we created a new class in the package *co.icesi.i2t.Choco3Reasoner.simple* and named it *Choco3Question* that implements the *Question* interface.
29. The package with the naming pattern *«2-letter country code».«University/Institution».«research group».«project name»* will contain the implementation for the performance result of the implemented reasoner. The names of this class should follow the naming pattern *«name of reasoner»PerformanceResult* (using CamelCase). For example, for the Choco 3 reasoner we created a class in the package *co.icesi.i2t.Choco3Reasoner* and named it *Choco3PerformanceResult* that extends the *PerformanceResult* abstract class provided by the *Benchmark* project in the package *es.us.isa.FAMA.Benchmarking*. This class should retrieve measures recorded from the reasoner while solving a question. The following measures are retrieved for the Choco 3 reasoner:
    - **Node count:** Gives the amount of nodes in the search tree. A node is possible value assignment to a variable. The search tree represents all possible value assignments to all the variables in the CSP.
    - **Current depth:** Gives the current depth of the search tree. It corresponds to the number of variables.
    - **Backtrack count:** Gives the amount of jumps to the last variable when a conflict with some other value arises.
    - **Time count:** Gives the time, in seconds, taken by the reasoner to solve the CSP. It includes constraint propagation time.
30. The package with the naming pattern *«2-letter country code».«University/Institution».«research group».«project name».«feature model type».questions* will contain the implementation for the questions supported by the implemented reasoner. The names of the classes implementing the questions should

follow the naming pattern *«name of reasoner»«name of question»* (using CamelCase). The next subsection will describe how to create a new question.
For example the questions implemented in the Choco 3 reasoner for simple feature models are: (the questions marked with a * use the default implementation provided by the *FaMaSDK* project)

- **Core features*:** This question is answered by calculating the features that are present in every product that can be derived from the feature model with the specified constraints. This operation gives the opposite result of the variant features operation.
- **Variant features*:** This operation calculates the features that are not present on every product that can be derived from the feature model with the specified constraints. This operation gives the opposite result of the core features operation.
- **Dead features*:** This question is answered by calculating the features that are not present in any product that can be derived from the feature model with the specified constraints. A feature is dead if it cannot appear in any product due to wrong definitions of cross-tree constraints.
- **Unique features*:** This question is answered by the features that are unique to a product that can be derived from the feature model with the specified constraints.
- **Number of products:** This question is answered by calculating the number of products that can be derived from the feature model with the specified constraints.
- **One product:** This question is answered by calculating a valid product of a feature model that can be derived from the feature model with the specified constraints.
- **Valid product*:** This question is answered by determining if a product is valid or not for a given feature model represented by a CSP. A product is a valid product if it is a solution of the CSP.
- **Products:** This question is answered by calculating all valid products that can be derived from the feature model with the specified constraints.
- **Valid configuration*:** This question is answered by analyzing if a configuration is valid or not. A configuration is a non-finished product that can need more features to be a valid product.
- **Commonality*:** This question is answered by calculating the percentage of products represented by the feature model including the input configuration. $Commonality = \frac{number\ of\ products\ after\ applying\ a\ configuration}{number\ of\ products}$
- **Variability*:** This operation calculates the variability degree of a feature model. The variability degree is the ratio between the number of products and 2^n where n is the number of features considered. In particular, 2^n is the potential number of products represented by a feature model assuming that any combination of features is allowed. The root and non-leaf features are often not considered. $Variability = \frac{number\ of\ products}{2^n}$, where n is the number of features considered.

- **Homogeneity*:** This question is answered by calculating the degree of homogeneity for the products that can be derived from the feature model with the specified constraints. Homogeneity is related with the number of unique features among products. The more unique features, the less homogeneous the feature model is. $Homogeneity = 1 - \left(\frac{number\ of\ unique\ features}{number\ of\ products}\right)$
- **Valid:** This question is answered by analyzing if a CSP representing a feature model is valid. A CSP is valid if after propagating the specified constraints at least one solution is found.
- **Detect errors*:** This question is answered by looking for errors on a feature model.
- **Explain invalid product:** This question provides as answer the options to repair an invalid product for a given feature model.
- **Valid configuration errors:** This question is answered by looking for explanations for errors in configurations of a given feature model.

Some questions have been deprecated in FAMA Framework 1.2 and therefore are not implemented in the Choco 3 reasoner for simple feature models. These questions are:

- **Filter:** This question takes as input a feature model and a partial configuration, and provides as answer the set of products that can be derived from the model and include the features from the input configuration. This question has been deprecated as this operation was included into the existing reasoners, and all additional reasoners must provide native support, through the methods *applyStagedConfiguration* and *unapplyStagedConfiguration*.
- **Set:** This question answers a list of given questions. A question can be added to the list by using the method *addQuestion*. This question has been deprecated.

Some questions were not implemented:

- **Explain errors in feature model diagram:** When a feature model has errors, this question is answered by looking for explanations for those errors.
- **Valid configuration errors AW (Another Way):** This question is the same as valid configuration errors but provides a more efficient implementation.

4.2. Developing a new question to be answered by the reasoner

The previous questions are the default questions a reasoner should be able to answer, but it should not be limited to this set. New questions may require a complete implementation to be answered or they can use or compose the solutions of already existing questions.

31. The new question should extend the abstract class created in step 28.
32. Provide the implementation for the three phases of the answering process. If you require to use or compose the solutions of already existing questions do not call the answer phase directly. You must use the ask method provided by the reasoner class implemented in step 27, as it will ensure that the three-phased answering process is carried out completely for each question.

4.3. Connecting the new FAMA Extension to the FAMA Framework

FAMA Extensions are loaded dynamically based on availability. The concept behind dynamic loading is to associate implementations to the available interfaces. Extensions can be loaded

using either OSGi or a Java class loader. Remember the *Activator* class from step 20? This class's responsibility is to load the FAMA Extension using OSGi and make it available to be used by the FAMA Framework. The SPL Core of the FAMA Framework, implemented by the project *FaMaSDK*, has an *Activator* class too. In this case the *FaMaSDK*'s Activator is used to load all the FAMA Extensions available.

Your FAMA Extension's *Activator* class should define two methods: *start* and *stop*. The *start* method is called when the FAMA Extension is activated and should register the services you implemented in it. The *stop* method should unregister such services. For example, for the Choco 3 reasoner the *start* method creates an instance for the reasoner's implementation that handles simple feature models as well as for each of the supported questions, and then registers them as services in the OSGi context so they can be available to *FaMaSDK*. The stop method unregisters the questions and the reasoner from the OSGi context.

If OSGi is not available or not used, the Extension can be loaded using a Java class loader, which works in a similar way as OSGi but services are not registered through a Java class. Depending on the type of extension (*e.g.* reasoner, metamodel) it would be required to have more than one loader configuration file. The FAMA loader configuration file is named *FaMaConfig.xml* and is found in the project *FaMa-Docs*. For a new reasoner extension, a `<reasoner />` tag should be added in-between the `<questionTrader></questionTrader>` tags. For the Choco 3 reasoner implementation we inserted the tag in Listing 6:

```
        <reasoner          id="Choco3Simple"          file="lib/Choco3Reasoner_1.0.0.jar"
class="co.icesi.i2t.Choco3Reasoner.simple.Choco3Reasoner" />
```

Listing 6. Reasoner tag added to the FAMA loader configuration file

Particularly for a reasoner extension, questions need to be loaded independently. The reasoner project should contain its own loader configuration file, in the *resources* folder, with interface-class pair tags. For a reasoner extension the loader configuration file needs to follow the format shown in Listing 7.

```
<reasoner>
        <question interface="«interface canonical name»" class="implementation class's
canonical name"/>
</reasoner>
```

Listing 7. FAMA Reasoner Extension loader configuration file template

The Choco 3 reasoner loader configuration file for simple feature models looks like the fragment in Listing 8:

```
<reasoner>
        <question interface="es.us.isa.FAMA.Reasoner.questions.CommonalityQuestion"
class="co.icesi.i2t.Choco3Reasoner.simple.questions.Choco3CommonalityQuestion"/>
        ...
</reasoner>
```

Listing 8. Fragment of the Choco 3 Reasoner loader configuration file for simple feature models

5. Package and test a FAMA Extension

6. Committing FAMA Extensions into the Git repository

When you first created the plug-in project back in Section 4 it was only placed in your workspace and not in the Git repository. You can use the EGit plug-in to perform this action.

33. To put your new FAMA Extensions project under the FAMA Framework Git repository, right click on your project and select *Team » Share Project*.
34. Select *Git* as the repository type. Click *Next >*.
35. Keep the option *Use or create repository in parent folder of project* ticked.

You have saved the project to the local Git repository in your computer; we still need to commit changes to the remote repository in your GitHub account.

36. Open the Git Staging view through the main menu option *Window » Show View » Other… » Git » Git Staging*.
37. In this view you select the project files from the list in the *Unstaged Changes* area and drag them to the *Staged Changes* area. For the Choco 3 reasoner we only select the files that belong to the project *Choco3Reasoner*.
38. Write a descriptive commit message and click the *Commit and Push* button.

Perform steps 33 to 38 for the initial changes. Afterwards the project will be under Git version control in both the remote (GitHub) and local (your computer) repositories. If you make further changes to the project and want to commit them to the Git repositories, you can use the Git Staging view. Drag only the files you want to commit from the *Unstaged Changes* area to the *Staged Changes* area. Write a meaningful commit message and click the *Commit and Push* button.

When you are ready to submit your new FAMA Extension to be included as part of the FAMA Framework you must go to your GitHub account and repository where your project is stored and click the *Compare and pull request* button. Once the pull request is sent, the ISA Research Group will review the set of changes, discuss them and pull the changes into the main FAMA Framework repository.

References

[1] David Benavides, Sergio Segura, Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. Information Systems, Volume 35, Issue 6, September 2010, pages 615-636, ISSN 0306-4379, http://dx.doi.org/10.1016/j.is.2010.01.001.

[2] Jesús García Galán. FaMa Framework (MSc. Thesis). Escuela Técnica Superior de Ingeniería Informática, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla. 2009.

[3] ISA Research Group. FaMa Framework User Manual. Escuela Técnica Superior de Ingeniería Informática, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla. 2011.

[4] ISA Research Group. FaMa Framework v1.2.0. Escuela Técnica Superior de Ingeniería Informática, Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla. 2014.

[5] Charles Prud'homme, Jean-Guillaume Fages. Choco3 Documentation Release 3.2.0. École des Mines de Nantes. 2014.