Proyecto RHEA

Contenido

[Visión general 2](#_Toc59098187)

[Big picture: 2](#_Toc59098188)

[Estructura del proyecto: 2](#_Toc59098189)

[Leyenda para el resto del documento: 2](#_Toc59098190)

[rhea.metamodels 4](#_Toc59098191)

[rhea.parsers 6](#_Toc59098192)

[rhea.generators 7](#_Toc59098193)

[rhea.aafm 8](#_Toc59098194)

[rhea.languageanalysis 9](#_Toc59098195)

[rhea.transformaciones 10](#_Toc59098196)

[rhea.evaluation 11](#_Toc59098197)

[Escenario principal: 11](#_Toc59098198)

[Métricas para evaluar 11](#_Toc59098199)

[Feature models a evaluar 11](#_Toc59098200)

[rhea.interfaces 12](#_Toc59098201)

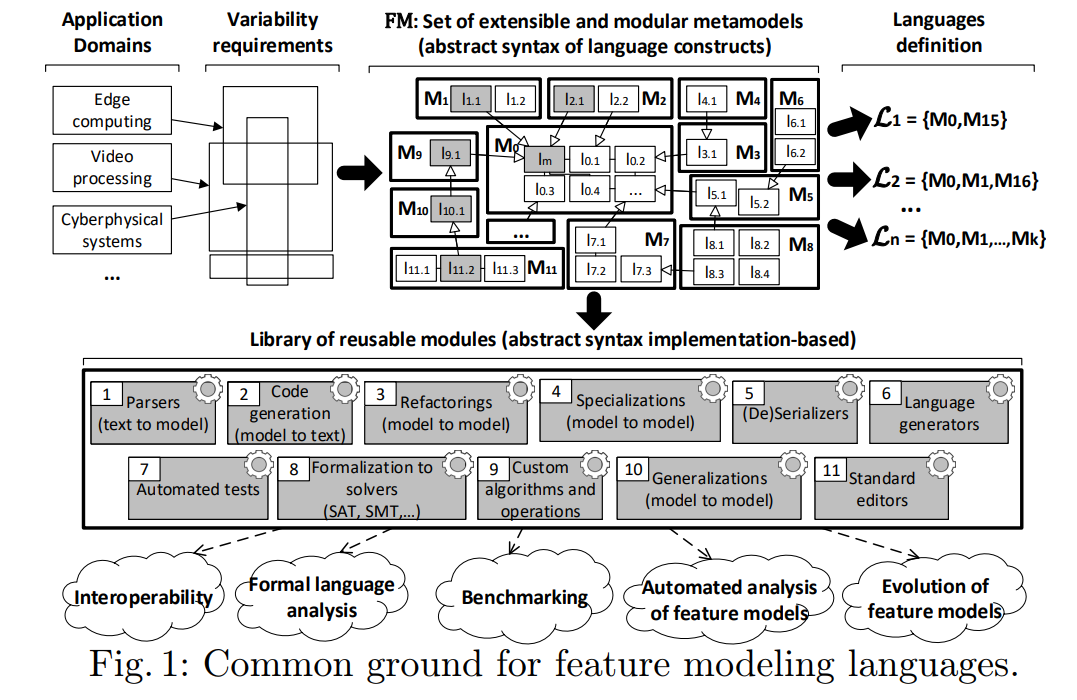
[Otros 13](#_Toc59098202)

[Herramientas de feature models 13](#_Toc59098203)

[Artículos 13](#_Toc59098204)

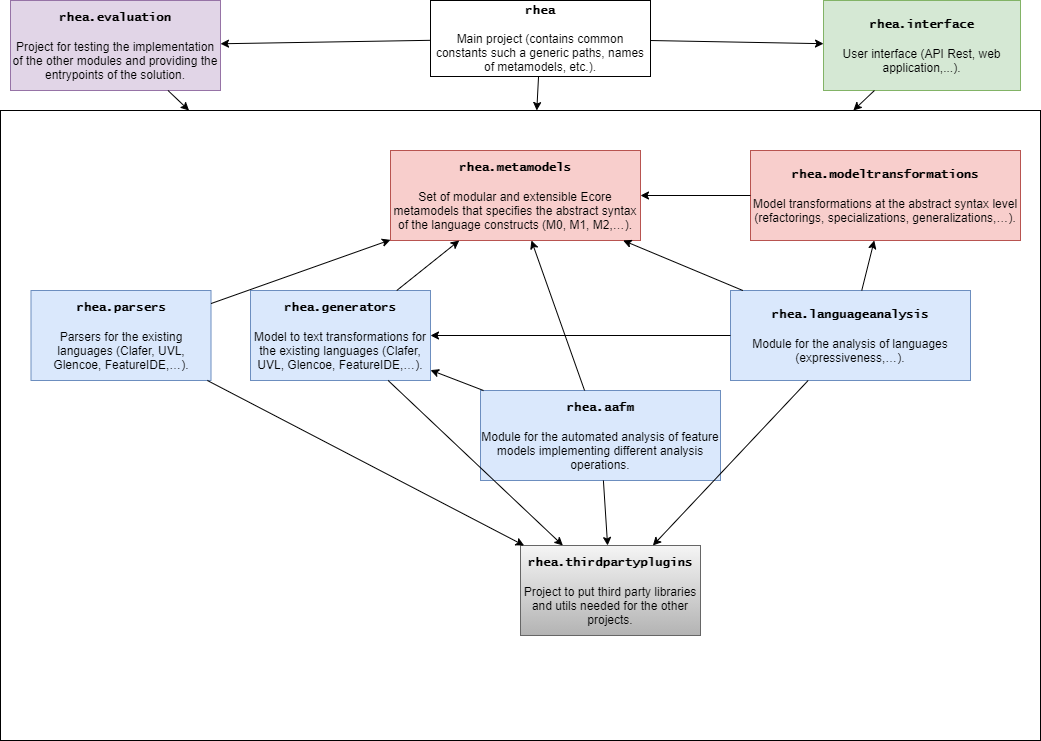
# Visión general

## Big picture:



## Estructura del proyecto:

**Repositorio Git:** https://github.com/CAOSD-group/rhea



## Leyenda para el resto del documento:

Tareas (ToDos)

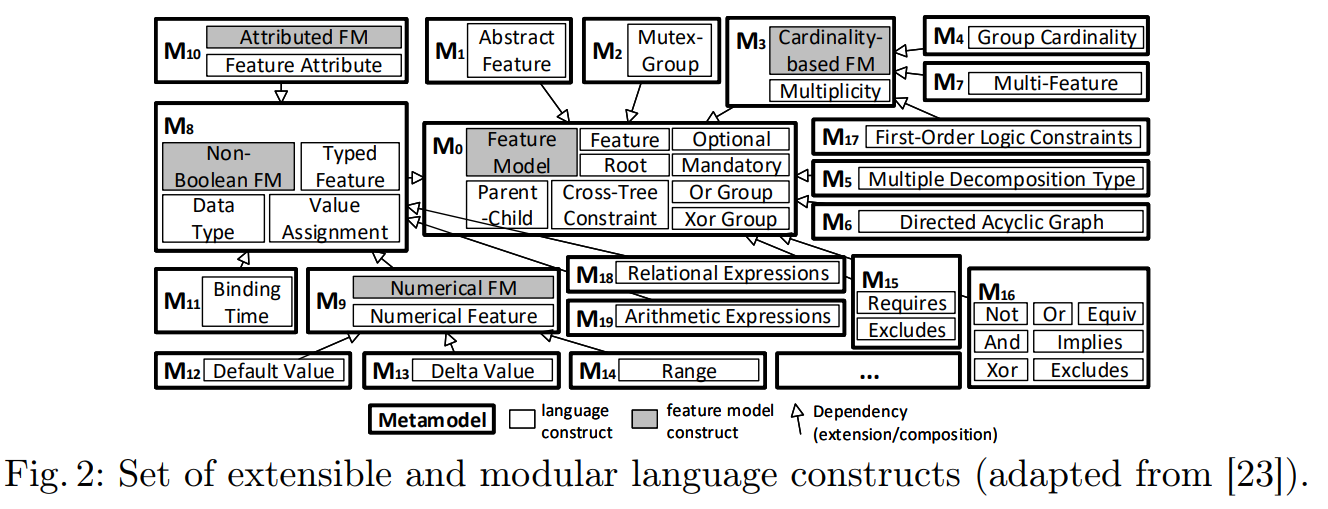
✔️: hecho.

**✇**: work in progress (medio a hacer).

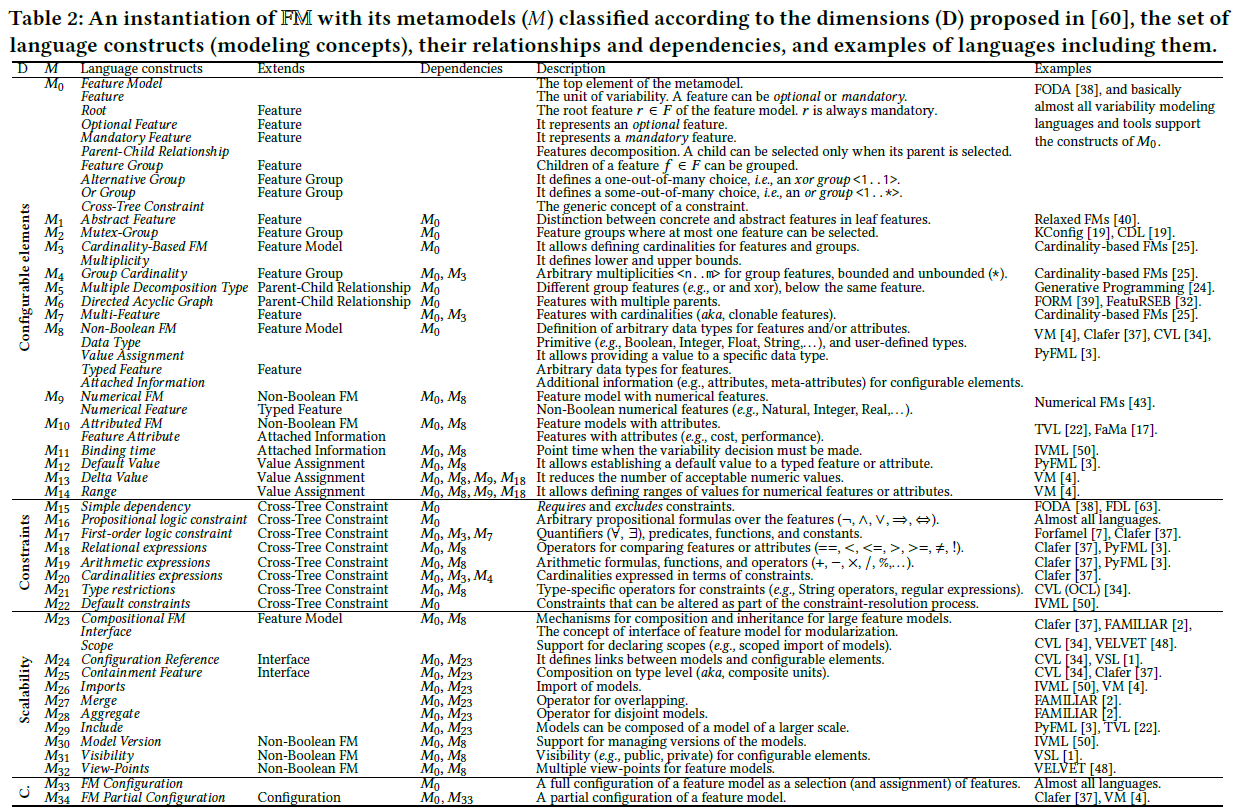
❌: por hacer.

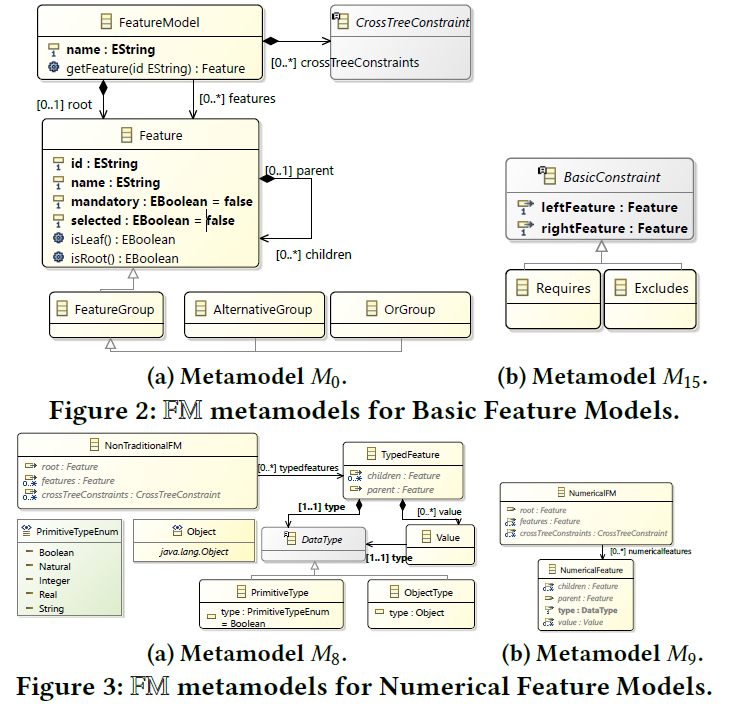
# rhea.metamodels

Definición de los metamodelos modulares (ver paper Horcas2020\_SPLC).



* ✔️Modelos de características básicos (**BasicFMs.ecore**) **Constructores:** FeatureModel, Feature, AbstractFeature, Root, Optional, Mandatory, Parent-Child, FeatureGroup (abstract), AlternativeGroup, SelectionGroup, CrossTreeConstraints (abstract)
* ✔️Restricciones básicas (**BasicCTCs.ecore**)  
  **Constructores:** BasicContraint, Requires, Excludes
* ✔️Restricciones en lógica proposicional (**PropLogicCTCs.ecore**)  
  **Constructores:** AdvanceConstraint, Term, TermType (feature, not, and, or, xor, implies, excludes, equiv)
* **✇** Modelos de características con cardinalidad (**CardinalityBasedFMs.ecore**)  
  **Constructores:** GroupCardinality, Multiplicity, Mutex-Group  
  **Tarea:** Añadir constructor para características Clonables.
* ❌Características con tipos de datos (**DataTypes.ecore**) [Hay que rehacerlo]  
  **Constructores:** DataType, PrimitiveType, TypedFeature, Value, ObjectType, NonBooleanFM
* ❌Características numéricas (**NumericalFMs.ecore**) [Hay que rehacerlo]  
  Constructores: NumericalFM, NumericalFeature
* ❌Características con atributos (**AttributedFMs.ecore**) [Hay que rehacerlo]  
  **Constructores:** AttributedFM, Attribute





# rhea.parsers

Implementación de los parsers para los diferentes lenguajes de variabilidad (transformaciones de la sintaxis concreta a la abstracta).

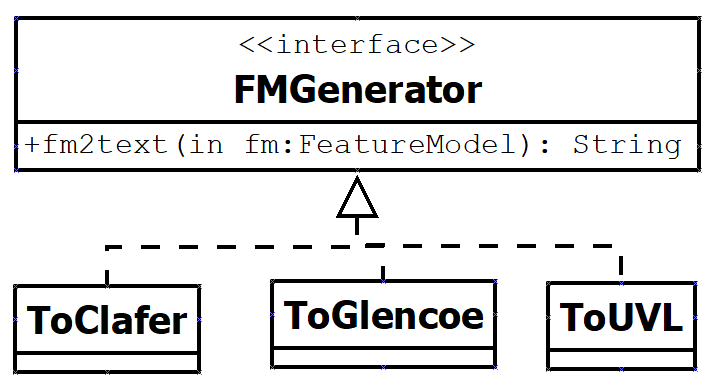


* **✇** Parser para Clafer [complicado]  
  **Constructores soportados actualmente:** FeatureModel, Feature, Root, Optional, Mandatory, Parent-Child, AlternativeGroup, SelectionGroup, GroupCardinality, Mutex-Groups,  
  **Tarea:** Añadir el resto de constructores (restricciones, numéricas,…)
* **✇** Parser para Glencoe  
  **Constructores soportados actualmente:** FeatureModel, Feature, Root, Optional, Mandatory, Parent-Child, AlternativeGroup, SelectionGroup, GroupCardinality  
  **Tarea:** Añadir el resto de constructores (restricciones, numéricas,…)
* **✇** Parser para UVL [por hacer, hay una librería en Java ya hecha por los propios autores, disponible en el thirdpartyplugins.]
* **✇** Parser para FeatureIDE [casi terminado, falta corregir los errores al exportarlo del otro proyecto de Josemi, y falta añadir las restricciones complejas, ahora mismo solo soporta IMPLIES y EXCLUDES].
* **✇** Parser para SPLOT [fácil con su propia librería, casi terminado, falta pasar el feature model de su librería a la sintaxis abstracta.]

|  |  |
| --- | --- |
| Feature model con características numéricas (size:int) | El mismo feature model de la izquierda representado en Clafer: |

# rhea.generators

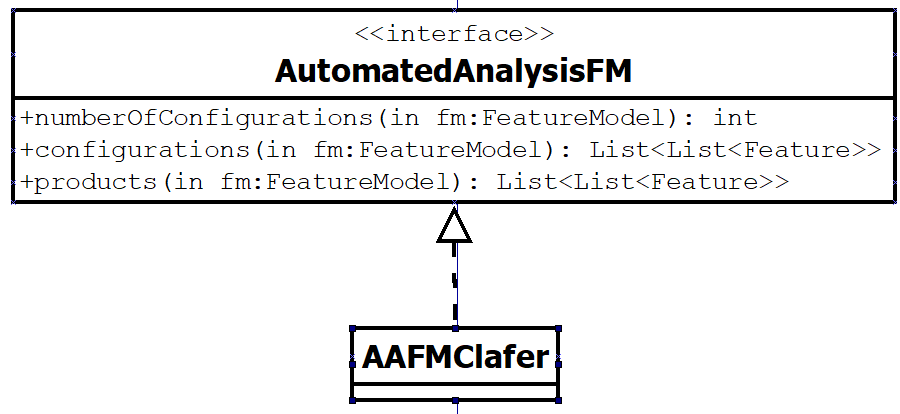
Implementación de los generadores (Model-to-text transformations) para los diferentes lenguajes de variabilidad (transformaciones de la sintaxis abstracta a la concreta).



* **✇** Generator para Clafer. Se usa su gramática .cf, se genera el código con la herramienta BNFC y ANTLR4. (Ver documento *gramatica-Clafer.docx* para detalles de este proceso).  
  **Constructores soportados actualmente:** FeatureModel, Feature, Root, Optional, Mandatory, Parent-Child, AlternativeGroup, SelectionGroup, GroupCardinality, Mutex-Group, Advanced constraints,  
  **Tarea:** Añadir el resto de constructores (numéricas,…)
* ❌Generator para Glencoe
* ❌Generator para UVL

# rhea.aafm

Implementación de las operaciones de análisis de los feature models en base a la sintaxis abstracta. Necesario para calcular los productos de los features models (Ver artículo de Benavides 2010 SLR).



* **✇** Implementación basada en los binarios de Clafer y Chocosolver   
  **Operaciones soportadas:** configuraciones, número de configuraciones, productos.  
  **Constructores soportados actualmente:** todos los de Clafer para ficheros Clafer y los mismos que el generador de Clafer para nuestros feature models.
* ❌Implementación basada en SAT4J y/o Chocosolver y otros solvers (BDD y/o #SAT.

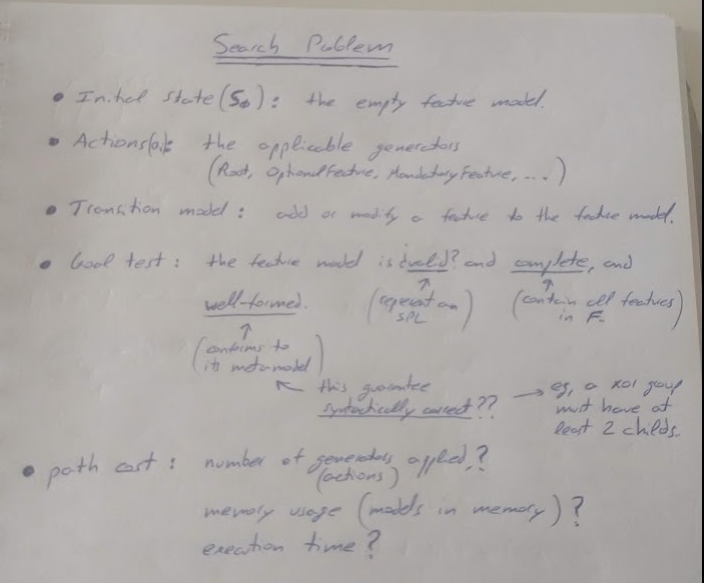
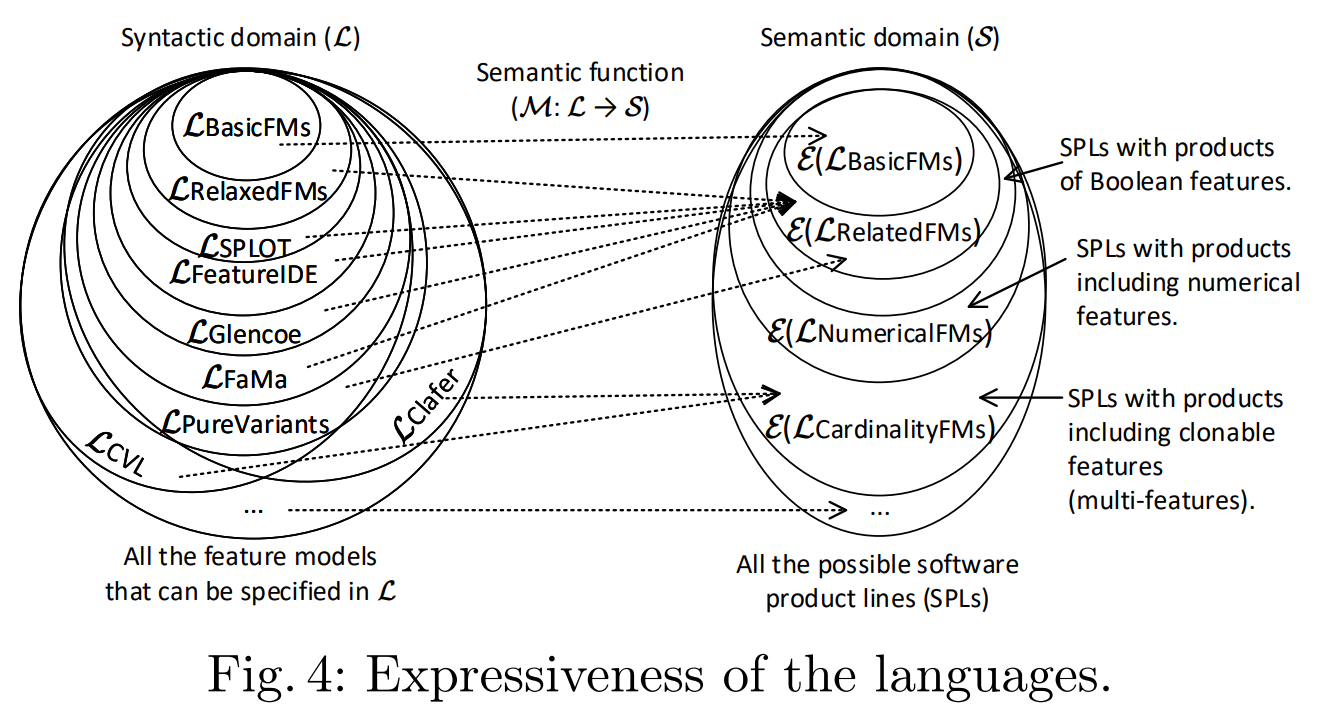
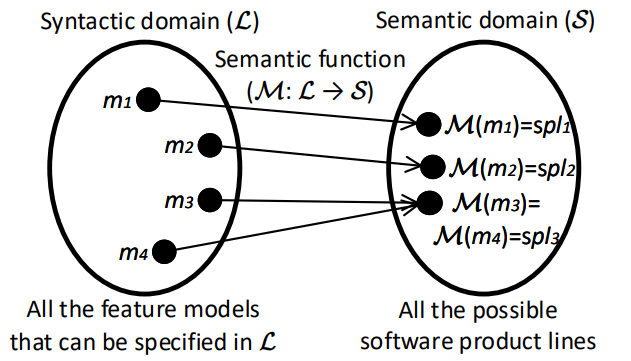
# rhea.languageanalysis

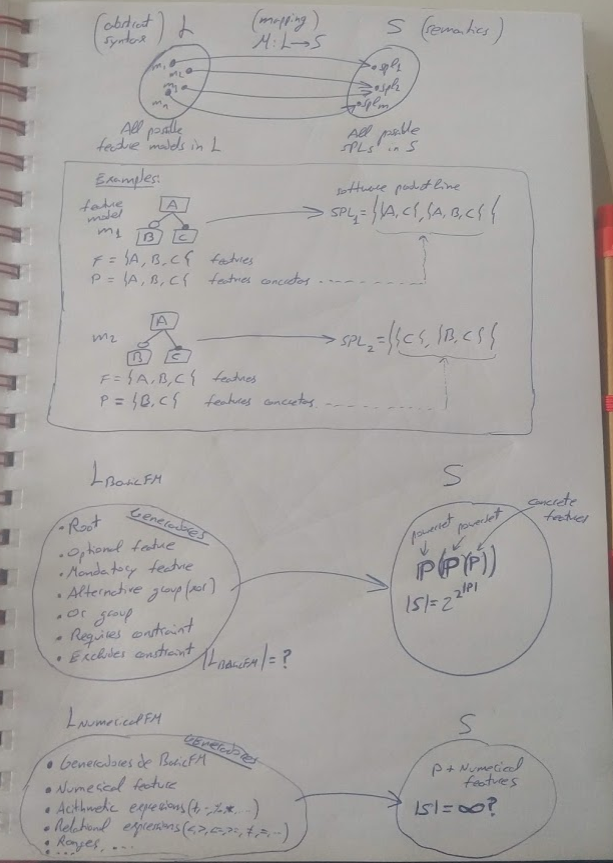
Este proyecto ahora mismo tiene dos clases:

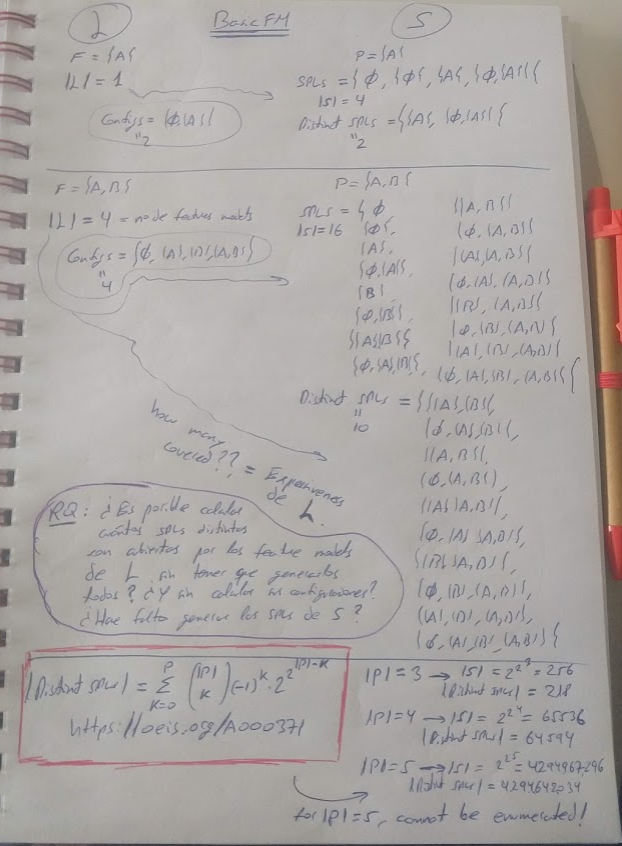
* Una clase **LanguageExpresiveness** ejecutable para analizar la expresividad a nivel de constructores de lenguajes. Esta clase también tiene código para generar todos los posibles productos, configuraciones y productos distintos de una SPL dado un conjunto de features concretas y abstracts (independientemente de del feature model).
* Una clase **FeatureModelGenerator** para generar todos los posibles feature models dados un conjunto de constructores del lenguaje y un conjunto de features.

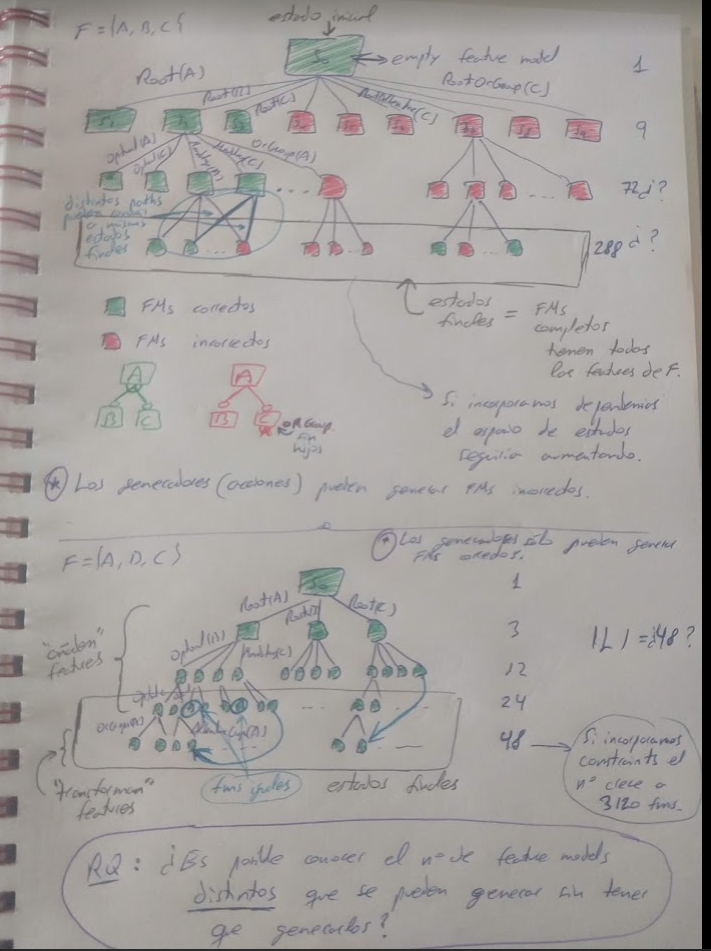
❌ **Tareas:** analizar la expresividad de los lenguajes.

* **Objectives:** Estudiar si es posible analizar la expresividad de un lenguaje de forma eficiente basándonos en los constructores del lenguaje.
* **Method:** El problema puede verse como un problema de búsqueda en espacio de estados (ver apuntes a mano en las siguientes páginas).  
  La idea es generar todos los posibles feature models dado un conjunto de constructores y un conjunto de features, calcular sus productos y comparar con el dominio de todos las posibles SPLs.





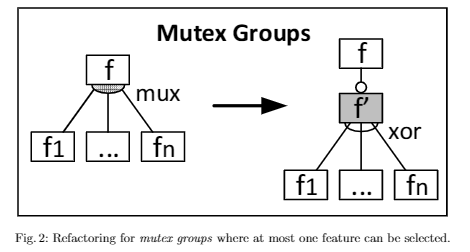




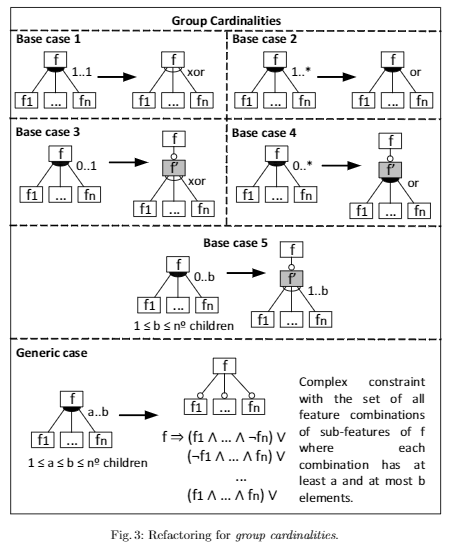
# rhea.transformaciones

Implementación de los refactorings, language generators,… (cualquier transformación de modelo) en Henshin. Incluye también las clases necesarias para ejecutar las transformaciones por código (el Henshin Engine) así como las clases para generar transformaciones (ficheros .henshin) por código (en el caso del group cardinalities).

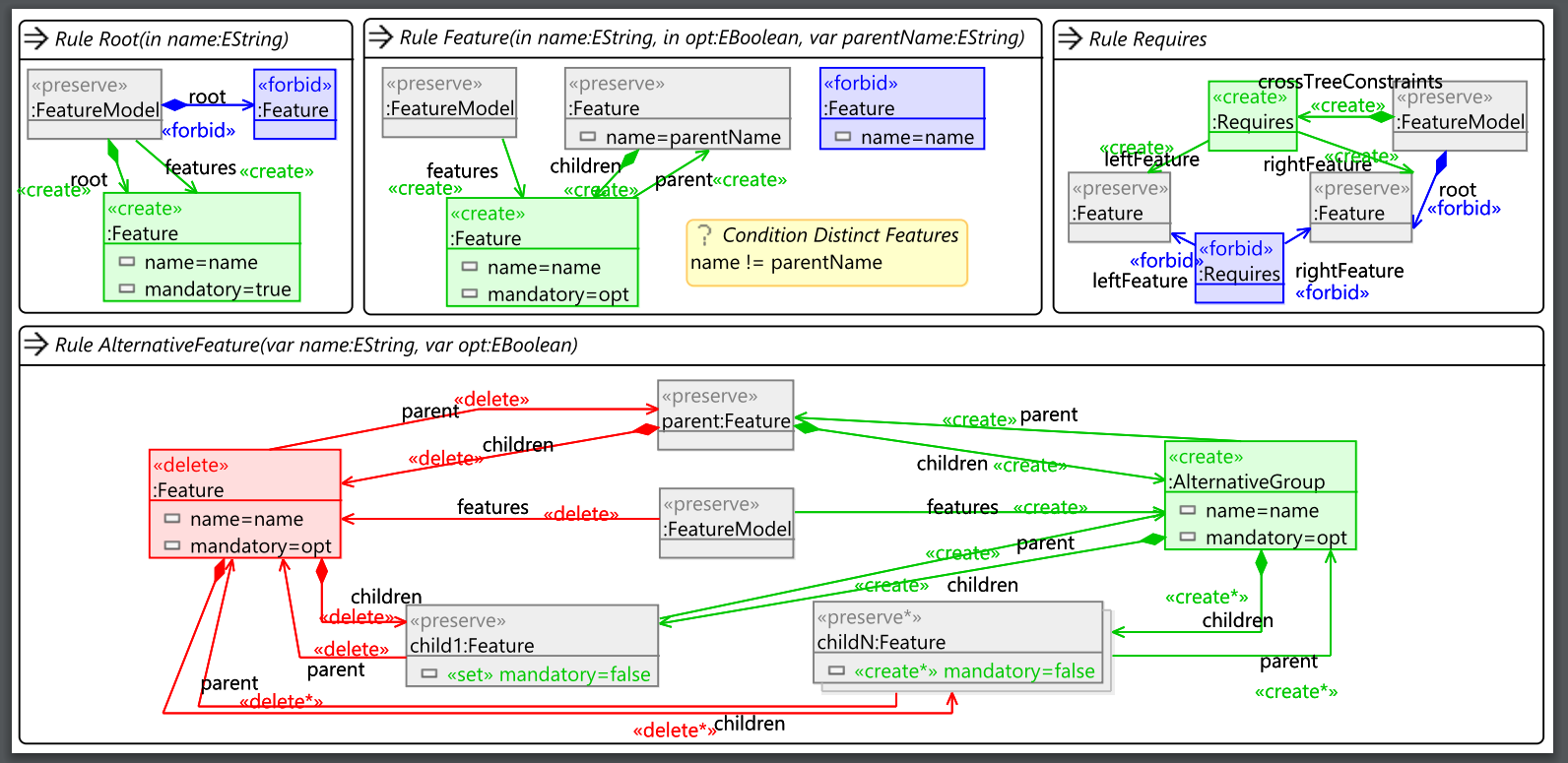
* Refactorings:
  + Mutex-Group.



* + Group cardinalities (casos bases, template + generador por Código de la transformación).



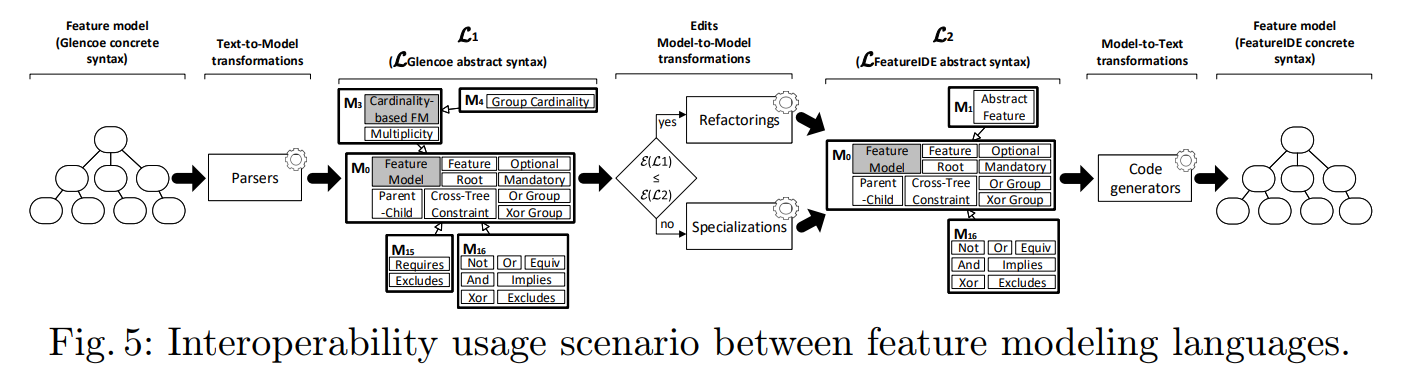
* Language generators: todos los definidos por Miguel en la carpeta “languagegenerators”. La siguiente imagen es solo una referencia para saber de qué estamos hablando, las versiones finales “buenas” son las que ha definido Miguel y están en el repositorio.



# rhea.evaluation

Este proyecto sirve para testear la funcionalidad del resto de proyectos y para la evaluación de los diferentes aspectos del proyecto.

## Escenario principal:



Contiene clases de tests que dado un FM en Clafer, lo parsea y convierte a la sintaxis abstract. Para comprobar que la transformación es correcta se calculan las configuraciones de ambos modelos. A continuación se aplicarán los refactorings y se volverá a comprobar las configuraciones. Por último se pasará el modelo a la sintaxis concreta de Clafer y se volverá a comprobar las configuraciones.

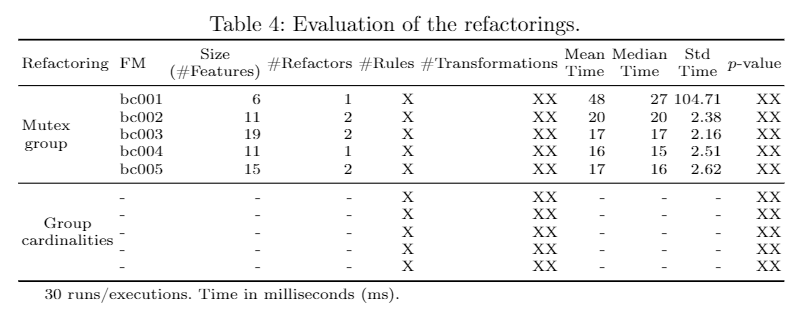
## Métricas para evaluar

1. **Soundness (correctness).** Si las transformaciones son correctas o no. Son correctas si mantienen el número de productos.
2. **Completeness.** Si las transformaciones son completas. Son completas si se aplican a todos los constructores complejos a tener en cuenta (e.g., se transforman todos los group cardinalities).
3. **Performance.** Tiempo de ejecución.
4. **Scalability.** Aplicación a feature models gigantes.

## Feature models a evaluar

Tres conjuntos de modelos:

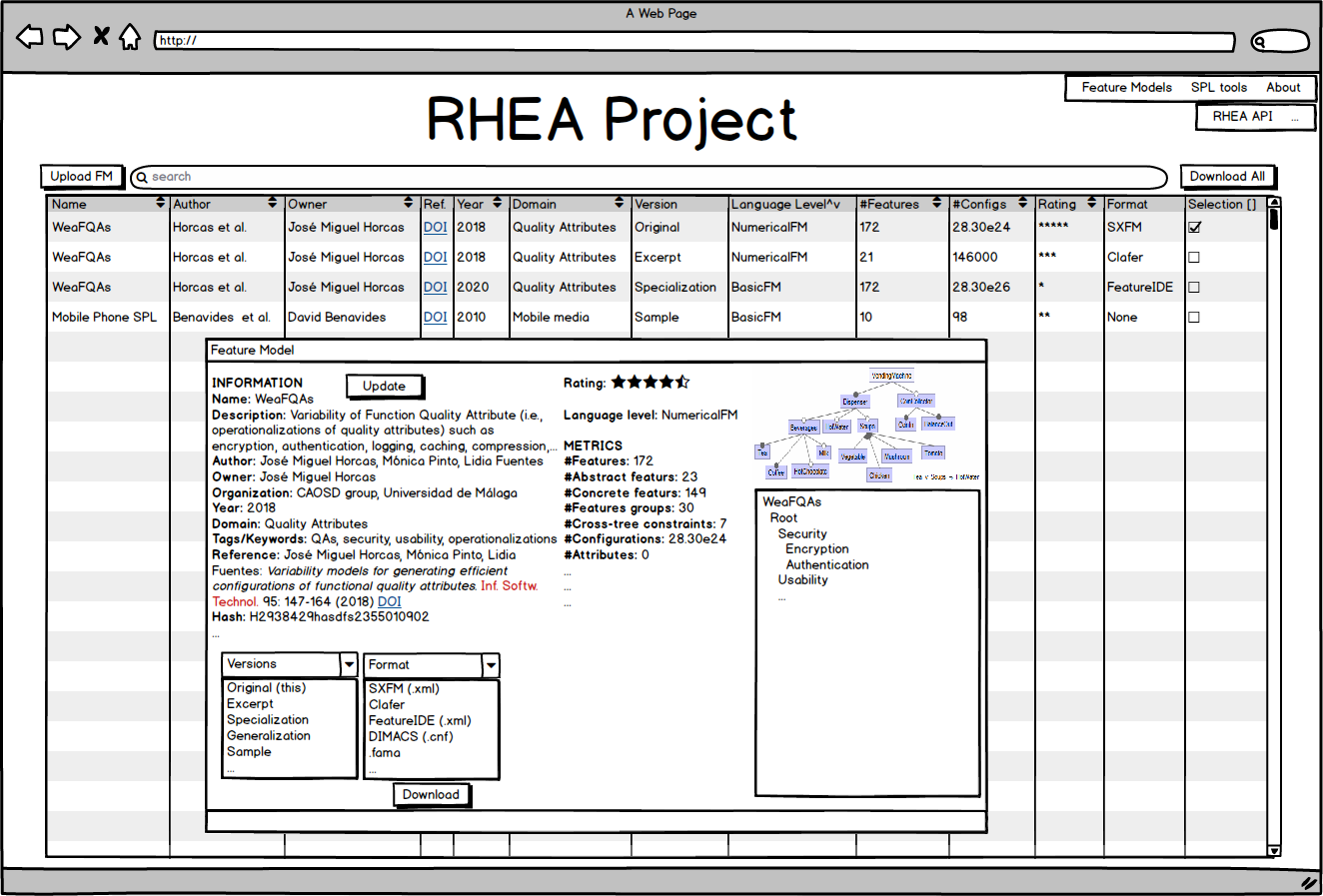
1. **Casos bases.** Definidos a mano, se usan para evaluar la correctitud y la completitud de las reglas.
2. **Feature models generados aleatoriamente.** Para X features y Y% de features de diferentes tipos. Se usan para el completness, el performance y la scalabilidad.
3. **Feature models reales.** Linux, automotive,…. Se usan para el completness, performance y scabilidad.

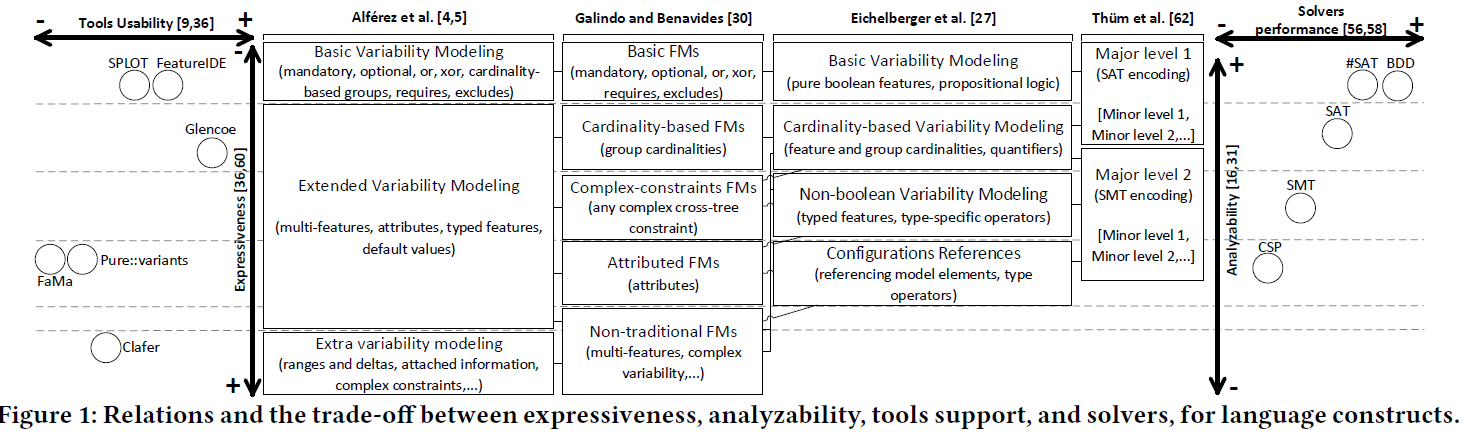


# rhea.interfaces

Idea de servicio web donde es integre toda la funcionalidad del proyecto. A modo de repositorio de FMs con diferentes versiones de los mismos (specializations, generalizations,…) y la posibilidad de descargarlos en cualquier formato.

[Ver artículo de Galindo y Benavides 2019 sobre repositorio de FMs.]





# Otros

## Herramientas de feature models

**(todas gratis, algunas como Glencoe y SPLOT son online disponibles desde el navegador):**

* Glencoe: <https://glencoe.hochschule-trier.de/webapp>
* SPLOT: <http://www.splot-research.org/>
* Clafer: <https://www.clafer.org/>
* FeatureIDE: <http://www.featureide.com/>

## Artículos

* Horcas2020 [SPLC] – Sobre la sintaxis abstracta y los metamodelos modulares.  
  <https://dl.acm.org/doi/10.1145/3382025.3414959>
* Knuppel2018 [FSE] – Sobre refactorings (restricciones complejas) y los Relaxed FMs (formalización.

<https://dl.acm.org/doi/10.1145/3106237.3106252>

* Thum2009 [ICSE] – Sobre los edits (refactorings, specializations, generalizations…)

<https://ieeexplore.ieee.org/document/5070526>

* Tanhaei2016 [IST] - Transformaciones (edits, no refactorings) implementados con ATL.

<https://doi.org/10.1016/j.infsof.2016.08.011>