ALEXIS FACQUES
M2 CLOUD COMPUTING & SERVICES PROJECT
UNIVERSITÉ DE RENNES 1 – 2017/18

PROJECT'S REPOSITORY AVAILABLE ON GITHUB:
/ALEXISFACQUES/DSL-FEATURE-MODELLING

HACK YOUR DOMAIN-SPECIFIC LANGUAGE

Throughout this project, we propose to create, develop, use and extend a Domain-Specific Language (DSL) expressing basic feature models variability. This includes:

- Creating an abstract syntax and grammar for our DSL, using Eclipse Xtext;
- Developing model transformations in order to compile files written in our custom language into other practical file formats and syntaxes. Transformations include:
    - JSON, which will allow web applications to parse the hierarchical structure of a feature model easily;
    - CNF (DIMACS format, for SAT Solvers compatibility), in order to test the satisfiability and solve the expressed features models;
    - Minizinc format, for interoperating with CSP Solvers;
- Expressing the practical uses of such a DSL through many examples of use, that include:
    - Creating feature models easily, that may allow benchmarking aforementioned solvers;
    - Developing interoperable User Interfaces for feature modelling configuration, using our DSL.

Development stages of our DSL are detailed in this report; sources and codes are available online, through this project's repository.

1 BASIC FEATURE MODEL AND XTEXT

In this first section, we propose a textual syntax for basic feature modelling using Eclipse Xtext. A feature model can be expressed as the definitions given in *Figure 1*.

Definition 2 (Feature Diagram) *A feature diagram* $FD = \langle G, E_{MAND}, G_{XOR}, G_{OR}, I, EX \rangle$ *is defined as follows:* $G = (\mathcal{F}, E, r)$ *is a rooted, labeled tree where* $\mathcal{F}$ *is a finite set of features,* $E \subseteq \mathcal{F} \times \mathcal{F}$ *is a finite set of edges and* $r \in \mathcal{F}$ *is the root feature ;* $E_{MAND} \subseteq E$ *is a set of edges that define mandatory features with their parents ;* $G_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *and* $G_{OR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *define feature groups and are sets of pairs of child features together with their common parent feature ;* I *a set of implies constraints whose form is* $A \Rightarrow B$*,* EX *is a set of excludes constraints whose form is* $A \Rightarrow \neg B$ *(*$A \in \mathcal{F}$ *and* $B \in \mathcal{F}$*).*

Definition 3 (Feature Model) *An FM is a tuple* $\langle FD, \psi \rangle$ *where* FD *is a feature diagram and* ψ *is a propositional formula over the set of features* $\mathcal{F}$*.*

Figure 1 - Feature Diagram and Feature Model definition

1.1 METAMODEL FOR FEATURE MODELING FORMALISM

First step towards building a DSL to express basic feature modelling consist in expressing the definitions given above with a metamodel; as this will help us to express a first syntax for specifying feature models.

We believe the following metamodel, given in *Figure 2*, to be a good formalization for feature modelling:

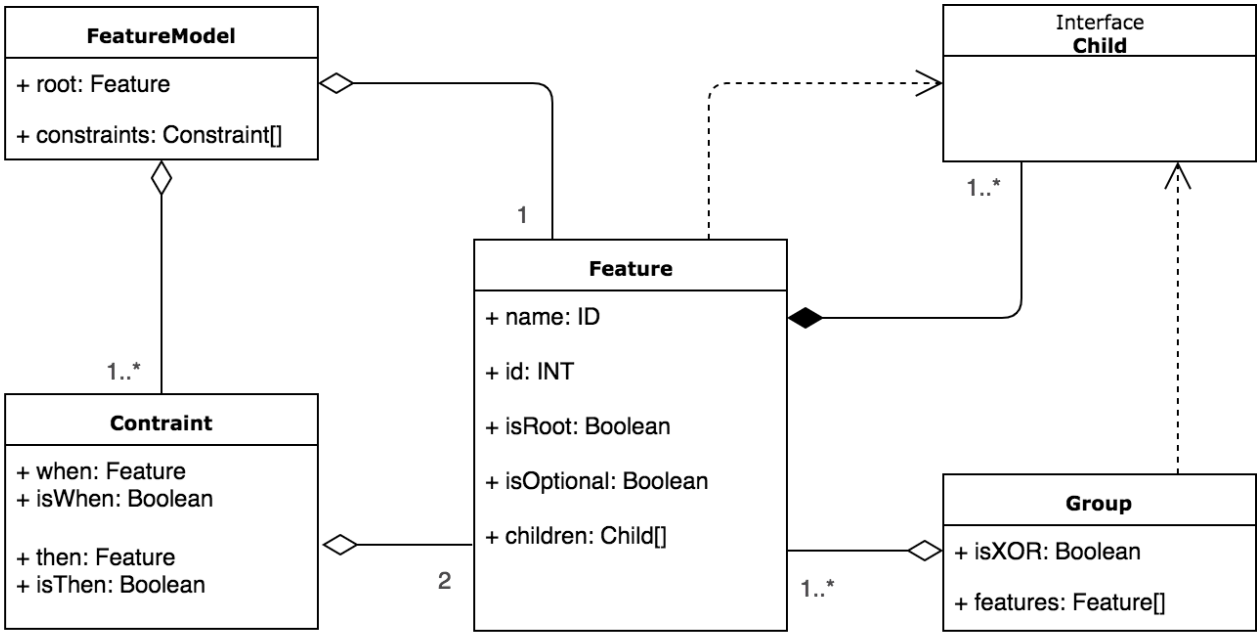

Figure 2 - Metamodel for basic feature modelling

1.2 XTEXT

Xtext is a well-known framework for developing Domain-Specific Languages. It includes a grammar language for describing a model, as well as auto-generated parser generators and class models in order to parse our Xtext-grammar expressed codes.

A. Expressed grammar

The Xtext grammar we came out with for expressing basic feature modelling, is the following:

```
grammar com.alexisfacques.xtext.FeatureDiagram with org.eclipse.xtext.common.Terminals

generate featureDiagram "http://www.alexisfacques.com/xtext/FeatureDiagram"

FeatureDiagramModel: declarations+=Declaration*;

Declaration: Constraint|Feature;

Constraint:
	'constraint' (description=STRING)?
		'when' (notWhen?='not'|'!')? when=[Feature|QualifiedName]
		'then' (notThen?='not'|'!')? then=[Feature|QualifiedName]
;

Feature:
	'feature' name=ID ('as' id=INT)?
		('{'
			children+=FeatureDefinition (',' children+=FeatureDefinition)*
		'}')?
;

QualifiedName: ID ('.' ID)*;

FeatureDefinition: ExtendedFeature|FeatureGroup;

ExtendedFeature: (isOptional?='optional'|'mandatory')? feature=Feature;

FeatureGroup:
	(isExclusive?='one'|'some') 'of'
		('{'
			children+=Feature (',' children+=Feature)*
		'}')
;
```

Figure 3 - Xtext grammar for FeatureDiagramModel (.diagram files)

Note that specifying a Feature's **id** is not mandatory, as this only make sense for certain model transformations we will detail later in this report. Specified **id** values will in all cases, be remapped by the model transformation functions.

B. Example of use

This aforementioned grammar allows us to express basic feature models as in the following examples, in *Figure 4* and *Figure 5*:

```
feature CarEquipment {
        mandatory feature Healthing {
                one of {
                        feature AirConditioningFrontAndRear,
                        feature AirConditioning
                }
        },
        mandatory feature Comfort {
                optional feature AutomaticHeadLights,
        },
        mandatory feature DrivingAndSafety {
                optional feature FrontFogLights
        }
}

constraint 'AutomaticHeadLights requires FrontFogLights'
        when CarEquipment.Comfort.AutomaticHeadLights
        then CarEquipment.DrivingAndSafety.FrontFogLights
```

Figure 4 - CarEquipment feature model expressed with our language

```
feature MobilePhone {
    mandatory feature Calls,
    optional feature GPS,
    mandatory feature Screen {
        one of {
            feature ColourScreen,
            feature HighResolutionScreen
        }
    },
    optional feature Media {
        some of {
            feature Camera,
            feature MP3
        }
    }
}

constraint 'Camera requires HighResolutionScreen'
    when MobilePhone.Media.Camera
    then MobilePhone.Screen.HighResolutionScreen
```

Figure 5 - Mobile phone feature model expressed with our language

1.3 GENERATED METAMODEL

The metamodel corresponding to the specified Xtext grammar above does not strictly correspond to the metamodel we established previously. On artifact generation, Xtext provides us with an auto-generated metamodel of our grammar. This metamodel is given below:

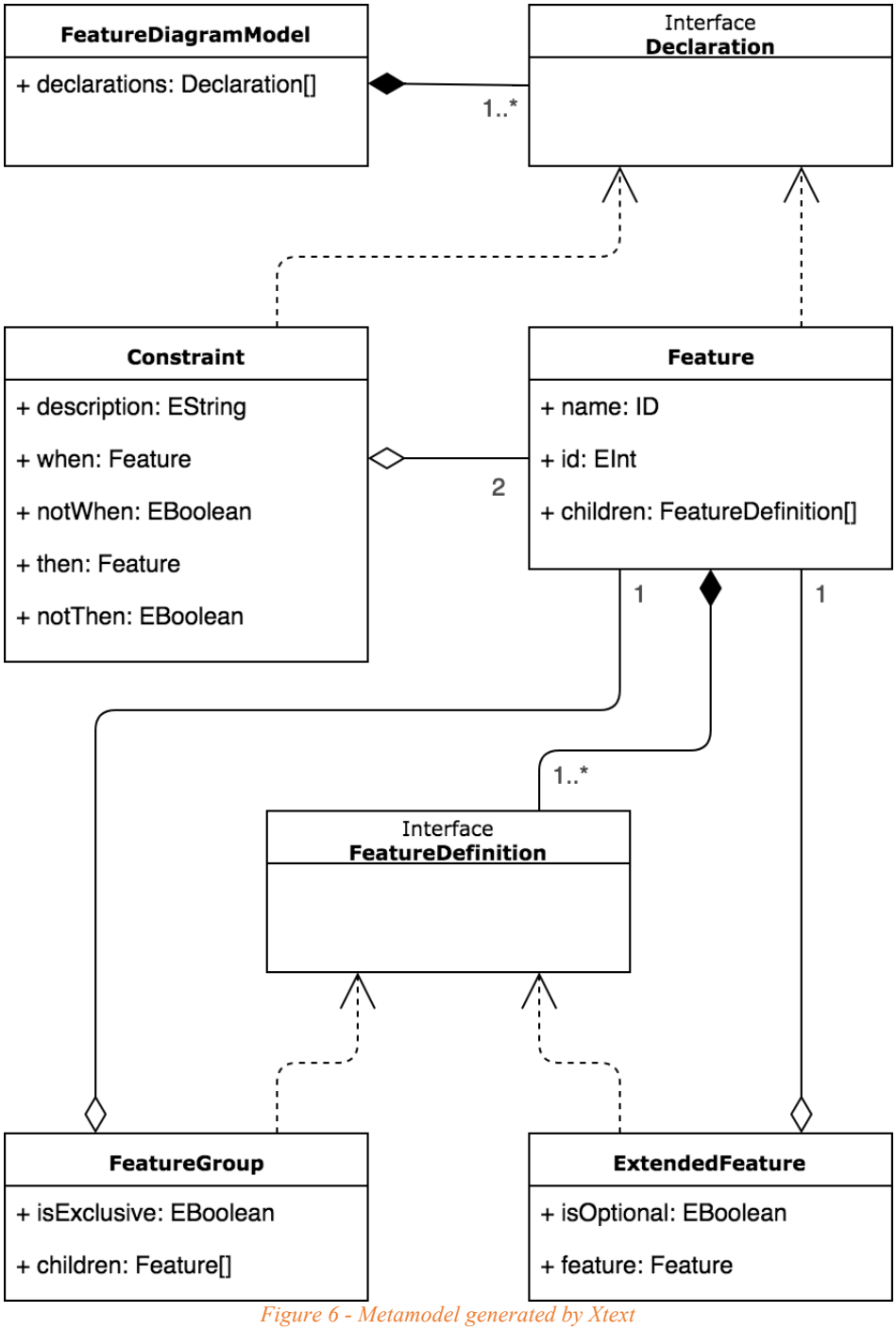

Figure 6 - Metamodel generated by Xtext

Both metamodels are however pretty similar. Declaring both Constraints and Features to be Declarations allows us to parse a FeatureDiagramModel more easily with repetitive structures.

Other specificities, as Feature's likeliness to be the root feature of a tree, or to be optional; finally made more sense to be expressed on the metamodel level, rather than the meta-metamodel level (as Class arguments).

2 TRANSFORMATION OF FEATURE MODELS

We will now focus on writing model transformation functions for our established Xtext grammar, using Java as well as Eclipse Xtend. Xtend makes programs written in our own language easily parsable, as previously generated Xtext artifacts include and Xtend Class for each declared Entity (or ParserRule), which include getters and setters for each or its arguments.

Transforming our language into other formats thus mainly consist of repetitive structures over arguments of declared Entities; mapping each of them to their equivalent declaration in any other language. In the context of feature modelling, model transformation does include many use cases as followed:

- Ease of structural representation of a feature model in another language;
- Checking for satisfiability of the expressed basic feature model by transforming it into a solver-interpretable form (Boolean algebra, etc…);

All developed model transformation functions have been gathered into an executable jar file, for ease of use, deployability, and interoperability.

2.1 JSON FORMAT

The first transformation we want to handle is model transformation towards JSON format. This model transformation is pretty straightforward, as our language structure is partly inspired by it.
We chose the resulting JSON file to be **only** reflecting the hierarchical structure of the Feature Model (as Child-parent relations), and not the functions that may or may not affect these relations (XOR, OR, Mandatory, Optional…).

Resulting structure would thus consist in a "stringified" array of the following structure (expressed in TypeScript):

```
export interface Feature {
    name: string,
    id: number,
    children: Array<Feature>
}
```

Figure 7 - Feature interface for JSON format transformation

Even though that transformation was not explicitly required, we believe it to be mandatory, as it allows JS-based clients (having Milestone 3 in mind) to parse our Feature Model but also allows us to keep track and map feature names to custom ids, for transformations which requires features to be expressed as such.

A. Format limitations

As we already mentioned though, this model transformation does not reflect any relation between features, apart from "child-parent" relations. In addition, constraints between entities are also purposely ignored as our interest only goes for the tree structure. This leads to theoretical inverse transformation not being possible.

B. Example of use

JSON transformation can be performed through command lines using the appropriate jar file as follows. Source files are available on the project's repository.

```
java -jar bin/transform.jar json data/test.diagram data/output.json
```

carequipment.diagram

```
feature CarEquipment {
        mandatory feature Healthing {
                one of {
                        feature ACFrontAndRear,
                        feature AC
                }
        }
}
```

carequipment.json

```
[
  {
    "children": [
      {
        "children": [
          {
            "children": [],
            "name":"ACFrontAndRear",
            "id":3
          },
          {
            "children": [],
            "name":"AC",
            "id":4
          }
        ],
        "name":"Healthing",
        "id":2
      }
    ],
    "name":"CarEquipment",
    "id":1
  }
]
```

Figure 8 - A feature model and its JSON transformation counterpart

2.2 DIMACS FORMAT

Second transformation we are interested in is towards DIMACS format (.cnf file). One original purpose of the DIMACS format was to ease the effort required to test and compare boolean satisfiability solving algorithms by providing a standard format for the problems addressed.
Expressing a feature model into this file format will allow us to be interoperable with many SAT algorithms, and will allow us to check the satisfiability of a configuration of the aforementioned feature model (as well as checking for deadlock features, core features and false optional features).

A. DIMACS Format

The DIMACS CNF file format is used to define a boolean expression / problem written in **conjunctive normal form**, which basically consist in:

- Multiple *clauses* joined by AND statements;
- Each clause, in turn, consists of *literals* joined by OR statements;
- Each literal is either the variable it represents, or its negation (NOT).

The following represents an example of a boolean satisfiability problem in 3 variables and 2 clauses, written in conjunctive normal form:

```
( x(1) OR ( NOT x(3) ) )
AND
( x(2) OR x(3) OR ( NOT x(1) ) ).
```

Figure 9 - A boolean satisfiability problem in CNF

CNF files are formatted as such:

- Files may start with comments, which are lines beginning with the character *c*;
- The number of variables and clauses of the problem must be declared through a line as following:

```
p cnf <NB_VAR> <NB_CLAUSES>
```

- Boolean problem declaration follows.
  - A line generally represents a *clause*;
  - *Clauses* must end with a *0* value;
  - *Literals* of each *clause* are represented by positive integers, and their negation by their negative counterpart.

An example of the boolean satisfiability problem expressed in Figure 9, in CNF format is given below:

```
p cnf 3 2
1 -3
2 3 -1
```

Figure 10 - Same boolean satisfiability problem expressed in CNF format

B. Feature Models to Boolean Algebra

Let's now focus in finding a way to express Feature models as boolean satisfiability problems. The following paper *[Krzysztof Czarnecki, Andrzej Wasowski "Feature Diagrams and Logics: There and Back Again." SPLC 2007]* offers an easy way to express relation between features for feature modelling, using **logical implications (=> operator)**, we could summarize in the following table:

Let *p* be a parent feature and *q* and *s* children features of *p*.

Relation	Resulting relation in boolean algebra
Child-parent	CHILD => PARENT
Mandatory	PARENT => MANDATORY
Or-group	PARENT => CHILD_1 OR CHILD_2 OR …
Xor-group	PARENT => CHILD_1 XOR CHILD_2 XOR …
Constraint	WHEN => THEN

Figure 11 – Expression of feature relations in boolean algebra

Keeping in mind that the conjunctive normal form of a logical implication is:

$$CNF(A => B) = -A\ OR\ B$$

…expressing a feature model into a boolean satisfiability problem can thus be possible by recursively parsing its feature diagram (and constraints), to express each of its features relations as shown above.

C. Limitations

Expression of a *n*-uple XOR group in conjunctive normal form is far from trivial; thus and XOR group is limited to 2 features only.

D. Example of use

CNF transformation can be performed through command lines using the appropriate jar file as follows. Source files are available on the project's repository.

As CNF files require variables to be integers, ids will be mapped automatically to each feature, and a JSON transformation will also be performed; with resulting JSON notation being shown though *stdout*, for ease of interpretation.

```
java -jar bin/transform.jar cnf data/test.diagram data/output.cnf
```

carequipment.diagram

```
feature CarEquipment as 1 {
   mandatory feature Healthing as 2 {
      one of {
         feature ACFrontAndRear as 3,
         feature AirConditioning as 4
      }
   },
   mandatory feature Comfort as 5 {
      optional feature AutoHeadLights as 6,
      mandatory feature LEDHeadLights as 7
   },
   mandatory feature DrivingAndSafety as 8 {
      optional feature FogLights as 9
   }
}

constraint 'AutoHeadLights requires
FrontFogLights'
   when CarEquipment.Comfort.AutoHeadLights
   then CarEquipment.DrivingAndSafety.FogLights
```

carequipment.cnf

```
p cnf 9 16
1 0
-1 2 0
-2 1 0
-2 -3 -4 0
-2 3 4 0
-3 2 0
-4 2 0
-1 5 0
-5 1 0
-6 5 0
-5 7 0
-7 5 0
-1 8 0
-8 1 0
-9 8 0
-6 9 0
```

Figure 12 – A serialized feature model and its CNF transformation counterpart

2.3 MINIZINC FORMAT

Last model transformation we will perform is expressing feature models as a **Constraint Satisfaction Problem** (CSP), for interoperability with CSP solvers such as Choco-Solver or Gecode.
CSP solving consist in solving boolean equation expressed as constraints between variable. In addition CSP-formatted feature models will be expressed in the Minizinc format (.mzn file), which include the support of operators such as disjunction (`\/`), exclusive disjunction (`xor`), negation (`not`) as well as implication (`->`); making the transformation from feature models a little bit more straight-forward.

Our strategy towards this transformation will consist in expressing each relation between features in our feature models, using the pre-established table given in *Figure 11*.

A. Minizinc format

MZN files are formatted as such:

- Declaration of each variable of the constraint satisfaction problem, as following:

```
var bool: VariableA;
var bool: VariableB;
var bool: VariableC;
```

- Declaration of constraints between variables, as following:

```
constraint VariableA -> VariableB xor VariableC;
```

- Solver function call:

```
solve satisfy;
```

B. Example of use

MZN transformation can be performed through command lines using the appropriate jar file as follows. Source files are available on the project's repository.

```
java -jar bin/transform.jar mzn data/test.diagram data/output.mzn
```

carequipment.diagram

```
feature CarEquipment {
   mandatory feature Healthing {
      one of {
         feature ACFrontAndRear,
         feature AirConditioning
      }
   },
   mandatory feature Comfort {
      optional feature AutoHeadLights,
      mandatory feature LEDHeadLights
   },
   mandatory feature DrivingAndSafety {
      optional feature FogLights
   }
}

constraint 'AutoHeadLights requires
FrontFogLights'
   when CarEquipment.Comfort.AutoHeadLights
   then CarEquipment.DrivingAndSafety.FogLights
```

carequipment.mzn

```
var bool: CarEquipment;
var bool: Healthing;
var bool: ACFrontAndRear;
var bool: AirConditioning;
var bool: Comfort;
var bool: AutoHeadLights;
var bool: LEDHeadLights;
var bool: DrivingAndSafety;
var bool: FogLights;

constraint CarEquipment;
constraint CarEquipment -> Healthing;
constraint Healthing -> CarEquipment;
constraint ACFrontAndRear -> Healthing;
constraint AirConditioning -> Healthing;
constraint Healthing -> ACFrontAndRear xor
AirConditioning;
constraint CarEquipment -> Comfort;
constraint Comfort -> CarEquipment;
constraint AutoHeadLights -> Comfort;
constraint Comfort -> LEDHeadLights;
constraint LEDHeadLights -> Comfort;
constraint CarEquipment -> DrivingAndSafety;
constraint DrivingAndSafety -> CarEquipment;
constraint FogLights -> DrivingAndSafety;
constraint AutoHeadLights -> FogLights;

solve satisfy;
```

Figure 13 – A serialized feature model and its MZN transformation counterpart

3 SOLVERS BENCHMARKING

With the aforementioned transformations made possible, the use a DSL for expressing basic feature models may start to make sense; as we can now easily express any feature model, and compile it in multiple concurrent formats, which could be of use for benchmarking differents families of boolean problems solvers as depicted previously.
This is what we focus on in this section. After generating random feature models of all size, we will present a very brief comparison of execution time between a SAT Solver and a CSP Solver implementation, using our model transformations functions.

3.1 RANDOM FEATURE MODEL GENERATION

First step will consist in writing a function for generating feature models of different sizes, on the following pattern:
Considering a submodel consisting of the conjunction of:

- An optional feature;
- The choice of one of (XOR) two features;
- The choice of some of (OR) two features;
- A mandatory feature.

Given a numerical input *i* the function will return a feature model composed of this submodel repeated *i* times; where the submodel *i + 1* is the child of the mandatory feature of *i*.

In addition, given a numerical input *j*; the feature model will generate *j* additional constraints between two randomly picked features of the feature tree.

FeatureModels, Features, and Constraints may easily be created using the pre-generated classes of the Xtend API we already used to parse entities of our language, for model transformation.

The developed feature model generator is available for testing through an executable jar, as follows; and sources may be find on the project's repository.

```
java -jar bin/transform.jar generate <NB_OF_HIERARCHICAL_LEVELS> <NB_OF_CONSTRAINTS> <OUTPUT_FILE_PATH>
```

3.2 SAT4J SOLVER IMPLEMENTATION

In this first section, we propose an implementation of the SAT4J solver in Java. SAR4J is a java library for solving boolean satisfaction and optimization problems, as depicted previously. We give a solution on how to use this library for either:

- Checking satisfiability of a feature model (at least one configuration possible);
- Detecting core features, dead features and false optional features (described as locked features through this report);
- Enumerating all possible configurations;

In addition, all functions will be able to take under account some assumptions on the variable configuration, thus in extent will respectively offer the possibility to:

- Check the validity of a given configuration;
- Return in “pseudo-real-time” the possible values of each variable the problem;
- Give all possible configurations including the assumptions.

Examples of implemented functions are given below, and runnable through a jar file; sources are available online on the project’s repository.
Functions take in input the preformatted feature model file in CNF format.

A. Satisfiability solving & configuration checking

Satisfiability solving function developed only consist of the solver's implementation example available in SAT4J's documentation, to which we added the possibility to add assumptions (boolean values of specified variables) preformatted in CNF syntax.
Note that assumptions on variable values actually only consist in additional standalone clauses, as they were part of the actual input file.

This implementation of the solver consists in finding at least one valid configuration for which the feature model is valid.

test.diagram	console

feature A as 1 {
  one of {
  feature B as 2,
  feature C as 3
  },
  mandatory feature D as 4
}
 	
constraint 'D requires C'
  when A.D
  then A.C
	$ java -jar bin/sat4j.jar satisfy test.cnf
	Satisfiable
	Execution time : 112ms

feature A as 1{
  one of {
  feature B as 2,
  feature C as 3
  },
}
 	
constraint 'B -> C' when A.B then A.C
constraint 'C -> B' when A.C then A.B
	$ java -jar bin/sat4j.jar all test.cnf
	Unsatisfiable
	Execution time : 107ms

Figure 14 – Satisfiability testing with SAT4J

By adding the possibility to formulate assumptions over the variables values, this function can be used as a mean to either validate or not a given feature model configuration.

test.diagram

```
feature A as 1 {
   one of {
      feature B as 2,
      feature C as 3
   },
   mandatory feature D as 4
}

constraint 'D requires C'
   when A.D
   then A.C
```

console

```
$ java -jar bin/sat4j.jar satisfy test.cnf 1,4,3
Satisfiable
Execution time : 94ms

$ java -jar bin/sat4j.jar satisfy test.cnf 1,2,4
Unsatisfiable
Execution time : 95ms

$ java -jar bin/sat4j.jar satisfy test.cnf -1
Unsatisfiable
Execution time : 95ms
```

Figure 15 - Configuration testing with SAT4J

B. Enumeration of all solutions

Yet again, this function is the implementation of the one given in SAT4J's documentation, to which the possibility of formulation assumptions through an input has been added.
This function parses all possible solutions of a given SAT problem, if the problem is valid.

test.diagram

```
feature A as 1 {
   one of {
      feature B as 2,
      feature C as 3
   },
   optional feature D as 4
}

constraint 'D requires C'
   when A.D
   then A.C
```

console

```
$ java -jar bin/sat4j.jar all test.cnf
1,2,-3,-4
1,-2,3,-4
1,-2,3,4
Execution time : 104ms

$ java -jar bin/sat4j.jar all test.cnf 4
1,-2,3,4
Execution time : 108ms
```

Figure 16 - Possible configurations of a feature model over assumptions with SAT4J

Formulating assumptions over the variables values will return all the possible configurations which including this particular set of values (see above).

C. *Locked features detection*

At last, a locked feature detection function has been developed. This consist of testing every variable of a given SAT problem to check if they are either core features (cannot be false), deadlock features (cannot be true) or still undetermined. The function returns the list of locked variable and their value, formatted in the CNF syntax.

test.diagram	console

feature A as 1 {
  one of {
  feature B as 2,
  feature C as 3
  },
  optional feature D as 4
}
 	
constraint 'D requires C'
  when A.D
  then A.C
	$ java -jar bin/sat4j.jar locked test.cnf
	1
	Execution time : 98ms
	
	$ java -jar bin/sat4j.jar locked test.cnf 1,4
	1,-2,3,4
	Execution time : 94ms

In extent, this can be used for real-time configurator validation, by updating the list of assumptions made over the variables' values each time the value of a feature changes (see above). This example will be implemented and described later on.

3.3 SAT 4J IMPLEMENTATION VS. CSP GECODE (MINIZINC)

At last, we propose a comparison of performances of two different solvers for solving the satisfiability of feature models, written in our DSL:

- Our SAT4J Java implementation;
- The CSP solver used in Minizinc IDE, knowned as Gecode.

Execution conditions not repeatable.

4 FROM FEATURE MODEL TO WIZARDS AND CONFIGURATOR

In this last section, we will conclude by developing a small web app exploiting the full potential of our DSL and its possible transformations.
This application consists in a dynamic feature model configurator (in *compilation* mode), written in TypeScript using NodeJS, which allows to interface **any** given feature model written with our language. The user will then be able to test configurations of the given feature model through an Angular-based User Interface.

Sources are available online through the project's repository. The application is provided containerized for easy testing.

4.1 NODE.JS SOLVER ENCAPSULATION

The NodeJS application consists in an Express web server, in which you can input a feature model written in our DSL through command line arguments.
The feature model is then compiled to both CNF, in order to provide solvability capacities to the application, and JSON format; which we be send to the client-side of our application, for dynamically creating the user interface.

The web client is capable of formulating assumptions over some variables values, depending on the user's choices through a REST API, and receives locked variables back, to dynamically, and visually change the features values on the UI. These is made possible by executing our SAT4J solver implementation as stated previously, through child processes.

In addition, the application trivially serves the Angular-generated web application for people to connect to.

4.2 ANGULAR APPLICATION

The developed UI is a configurator in *compilation* mode, through which the user can pick a configuration of a given feature model. It is developed with Angular.
The application consists in a custom "three-state" custom checkbox component, which allows the selection or deselection of a feature.

- On connection, the feature model's JSON structure is required through and HTTP (Ajax-based) call towards our REST API, in order to recursively generate the UI, binding a custom checkbox to each feature;
- Once the UI is generated, locked features may now be required through the REST API. These locked features consist in either core features, deadlock features, or consequences of choices of the user. Values of locked features will be set as fixed, and the user won't be able to change them;
- The user may now select or deselect features as he sees fit. Every change triggers a REST API call, updating the "locked features" consequently.

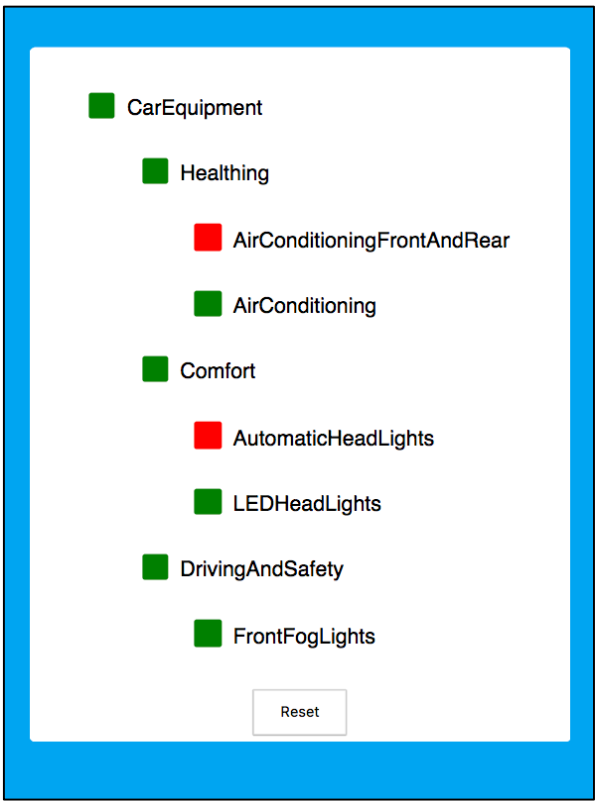

Figure 17 - The feature model configurator