Formal Modeling of a Feature Model in VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

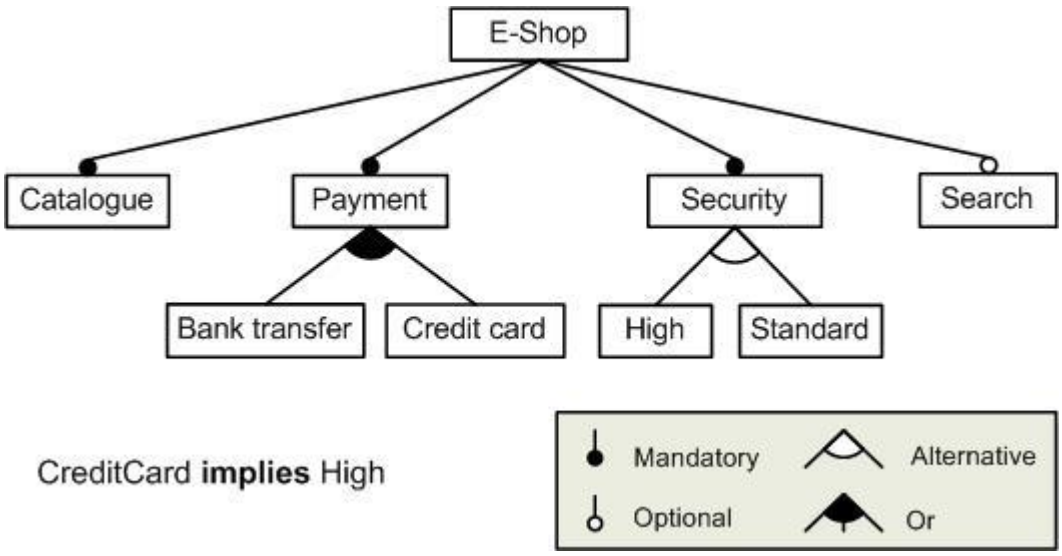
António Pedro Araújo Fraga

1. Informal system description and list of requirements

1.1 Informal System Description

A Feature Model is designed to represent the products of the Software Product Line in a simple, hierarchical fashion. It is structured using a tree model and each feature falls into one of this groups: *mandatory*, *optional*, *alternative* and *or*.

An example of a feature model that we chose to use is that of an e-shop, as presented in the Feature Diagram below:



Img 1: E-Shop Feature Model

1.2 List of Requirements

Id	Priority	Description
R1	Mandatory	The user can build a Feature Model by setting a root
R2	Mandatory	The user can check whether a Model configuration is valid or invalid
R3	Mandatory	The model should generate all valid configurations
R4	Mandatory	The user can create a Feature
R5	Mandatory	The user can create a Parent
R6	Mandatory	The user can set several features as Parent sub features
R7	Mandatory	The user can set a Feature as optional

2. Visual UML Model

2.1 Use Case Model

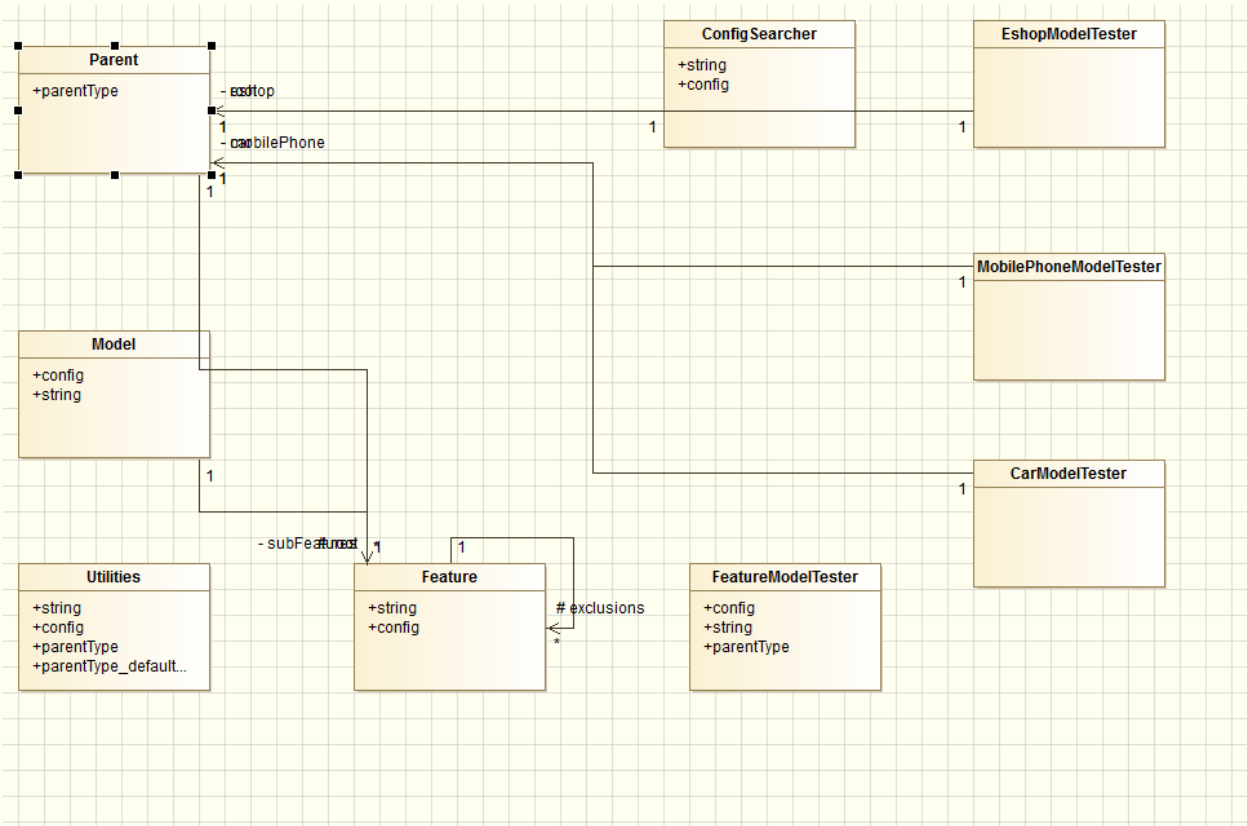
Scenario	**Create Feature Model**
Description	Normal scenario for creating a feature model (with a root as feature)
Pre-conditions	1. The tree should not have features with the same name. 2. A feature should have a string with at least one char.
Post-conditions	1. Root should be initialized
Steps	1. Create Features. 2. Create sub Features. 3. Set some features as Parents. 4. Set a Feature as root.
Exceptions	(none)

Scenario	**Validate configuration of a Model**
Description	Normal scenario for validating a configuration of a Feature Model
Pre-conditions	1. Configuration must have a valid domain, all the features must be part of the Model tree.
Post-conditions	(none)
Steps	1. Create a model 2. Create a map with all the model features as key and the presence status as value. 3. Test whether a configuration is valid or not.
Exceptions	1. The configuration doesn't have all the features as keys (step 2)

Scenario	**Generate valid configurations of a model**
Description	Normal scenario for generating valid configurations of a Feature Model
Pre-conditions	1. A model should be defined
Post-conditions	(none)
Steps	1. Create a model 2. Request all valid configurations

Exceptions	1. A model doesn't have the root (step 1)

2.2 Class Model



Class	Description
Parent	Defines a Parent, subclass of a Feature.
Feature	Defines a feature of a Feature Model.
Model	Defines a Feature Model
Utilities	Contains utility types and functions to work with maps
ConfigSearcher	Contains a method that searchs a model tree to generate valid configurations
FeatureModelTester	Defines the test scenarios and test cases for a Feature Model
CarModelTester	Defines a Feature Model to be tested, described at 4.1

EshopModelTester	Defines a Feature Model to be tested, described at 4.2
MobilePhoneModelTester	Defines a Feature Model to be tested, described at 4.3

3. Formal VDM++ Model

3.1 Class ConfigSearcher

This class is used to search for valid configurations of a given model. It has a set of valid configurations, a map which the key is the name of a feature, and the value is a boolean representing the presence of the same feature in the configuration. We have developed an optimized algorithm to conclude this search:

This searcher has an empty set as valid configurations, and it starts by analyzing the root of the given model. If the root is mandatory then it pushes a new set to the valid configuration set:

```
{{root}}
```

But if the root is optional then it pushes two new sets to this set:

```
{{root}, {}}
```

As the algorithm is analyzing sub features of the given tree it separates the sub features in three types, a feature with an **or parent,** a feature with an **xor parent** and a feature with a **regular/default parent.**

In case a feature has an **or parent,** and supposing that there are two sub features (subFeature1, subFeature2), the algorithm generates the valid configurations related with these sub features while is analyzing their parent. So it pushes the new valid configurations to the related set, making a **union** with the existing sets:

```
{{ root, subFeature1 }, { root, subFeature2 },
{ root, subFeature1, subFeature2}}
```

In case a feature has an **xor parent,** and also supposing that there are two sub features with the same name as above, the algorithm also generates the valid possibilities while is analyzing their parent:

```
{{ root, subFeature1 },
```

```
{ root, subFeature2 }}
```

In case a feature has a **regular parent** then the possibilities are generated while the algorithm is analyzing the feature and not its parent. In this case the algorithm has the same behaviour as the one observed in the root analysis, but since there are configurations in the valid configurations, it's made a **union** between these configurations and the new possibilities. So, if a sub feature with a regular parent is optional then the valid configurations state is:

```
{{root, subFeature}, {root}}
```

If a parent is optional, then the algorithm generates the valid configurations by passing the optional parents in a given subtree. So, if a feature is optional and it has an optional parent, and supposing there is a subSubFeature that is optional too, the steps of the tree analysis is:

1. `{{root}, {}}`

2. `{{root, subFeature}, {root}, {}}`

3. ```
   {{root, subFeature, subSubFeature},
   {root, subFeature},
   {root}, {}}
   ```

When the tree analysis is completed then the algorithm applies the restrictions of implication and exclusion. The valid configurations are now generated.

```
-- ConfigSearcher Class
class ConfigSearcher
types
	public string = Utilities`string;
	public config = Utilities`config;
instance variables
	private root : Parent;
	private nodeCount : nat;
	private restrictedInvalidSubsets : set of config;
	private validConfigs : set of set of string := {{}};
operations

-- ConfigSearcher Constructor
-- 1st arg: Root feature of the model to be tested
public ConfigSearcher: Feature ==> ConfigSearcher
	ConfigSearcher(r) == (root := r; nodeCount := root.nodeCount();
restrictedInvalidSubsets := root.invalidSubsets(); return self;);

-- Gets all valid configurations of the root associated with this ConfigSearcher
-- Return: Sequence of configs as all valid configurations
public getValidConfigs: () ==> set of set of string
	getValidConfigs() == (
		searchFeatureTree(root, true, {});
```
```

```
		applyRestrictions();
		return validConfigs;
	);

-- Applies all restrictions related with the model to be tested
public applyRestrictions: () ==> ()
	applyRestrictions() ==
	(
		validConfigs :=
	{elem | elem in set validConfigs & not exists restriction in set
restrictedInvalidSubsets &
	-- requirements
	(rng restriction = {true, false} and
	(dom (restriction :> {true}) subset elem and not dom (restriction :> {false})
subset elem))
	or
	-- exclusions
	(rng restriction = {true} and
	(dom restriction subset elem)
	)
	};
	);

-- Searchs Feature tree in order to generate valid configurations
-- 1st arg: Feature to be searched
-- 2nd arg: Boolean indicated if the feature parent is a <xorParent> or an <orParent>
-- 3rd arg: Set of optional <orParent> or <xorParent>
public searchFeatureTree: (Feature | Parent) * bool * set of string ==> ()
	searchFeatureTree(feature, hasDefaultParent, optionalParents) ==
	(
	if hasDefaultParent then defaultParentConfigs(feature);
	checkOptionalParents(feature, optionalParents);
	if isofclass(Parent, feature) then (
		dcl newOptionalParents : set of string := optionalParents;
		if feature.isXorParent() then xorParentConfigs(feature)
		else if feature.isOrParent() then orParentConfigs(feature);
		if not feature.isMandatory() then newOptionalParents :=
newOptionalParents union {feature.getName()};
		for all subFeature in set feature.getSubFeatures() do
			searchFeatureTree(subFeature, feature.isDefaultParent(),
newOptionalParents);
	);
	);

-- Generates valid configurations of a feature with a <defaultParent>
-- 1st arg: Feature related with valid configurations
public defaultParentConfigs: (Feature) ==> ()
	defaultParentConfigs(feature)  ==
	(
		dcl possibilities : set of set of string := {};
		if feature.isMandatory() then possibilities := {{feature.getName()}}
		else possibilities := {{feature.getName()},  {}};

		validConfigs := { validConfig union possibility | validConfig in set
validConfigs, possibility in set possibilities};
```

```
	);

-- Generates valid configurations of a feature with a <xorParent>
-- 1st arg: Feature related with valid configurations
public xorParentConfigs: Parent ==> ()
	xorParentConfigs(parent) ==
	(
		dcl subFeatures : set of string := parent.getSubFeaturesNames();
		dcl configsWithoutParent : set of set of string := {validConfig |
validConfig in set validConfigs & parent.getName() not in set validConfig};
		dcl configsWithParent : set of set of string := {validConfig |
validConfig in set validConfigs & parent.getName() in set validConfig};

		validConfigs := {config union possibility | config in set
configsWithParent, possibility in set {elem | elem in set power subFeatures & card
elem = 1}} union configsWithoutParent;
	);

-- Generates valid configurations of a feature with an <orParent>
-- 1st arg: Feature related with valid configurations
public orParentConfigs: Parent ==> ()
	orParentConfigs(parent) ==
	(
		dcl subFeatures : set of string := parent.getSubFeaturesNames();
		dcl configsWithoutParent : set of set of string := {validConfig |
validConfig in set validConfigs & parent.getName() not in set validConfig};
		dcl configsWithParent : set of set of string := {validConfig |
validConfig in set validConfigs & parent.getName() in set validConfig};

		validConfigs := {config union possibility | config in set
configsWithParent, possibility in set {elem | elem in set power subFeatures & elem <>
{}}} union configsWithoutParent;
	);

-- Checks if there are optional parents so far, used to generate valid configurations
with optional parents
-- 1st arg: Feature as the related feature
-- 2nd arg: Sequence of strings as optional parents identified so far
public checkOptionalParents: Feature * set of string ==> ()
	checkOptionalParents(feature, optionalParents) ==
	(
		validConfigs :=
	{elem | elem in set validConfigs & not exists optionalParent in set
optionalParents & ((not optionalParent in set elem) and feature.getName() in set
elem)};
	);

end ConfigSearcher
```

3.2 Class Feature

This class defines a Feature in the Features Model.

```
-- Feature Class
class Feature
types
	public string = Utilities`string;
	public config = Utilities`config;
instance variables
	public name: string;
	protected mandatory: bool := true;
	protected requirements: set of Feature := {};
	protected exclusions: set of Feature := {};

	inv name <> "";
operations

-- Feature constructor
-- 1st arg: Name to be associated with the feature
public Feature: string ==> Feature
	Feature(n) == (name := n; return self;)
	post name = n;

-- Sets Feature as mandatory
-- 1st arg: Boolean with mandatory status (True as mandatory, False as optional)
public setMandatory: bool ==> ()
	setMandatory(b) == mandatory := b;

-- Sets Feature requirements
-- 1st arg: Sequence of features as requirements
public setRequirements: set of Feature ==> ()
	setRequirements(fs) == (requirements := fs;)
	post requirements inter exclusions = {};

-- Sets Feature exclusions
-- 1st arg: Sequence of features as exclusions
public setExclusions: set of Feature ==> ()
	setExclusions(fs) == exclusions := fs
	post {} = {element | element in set exclusions & element.mandatory} and
requirements inter exclusions = {};

-- Gets Feature Requirements
-- Return: Sequence of features as requirements
public getRequirements: () ==> set of Feature
	getRequirements() == return requirements;

-- Gets Feature Exclusions
-- Return: Sequence of features as exclusions
public getExclusions: () ==> set of Feature
	getExclusions() == return exclusions;

-- Gets Feature name
-- Return: String as feature name
public getName: () ==> string
	getName() == return name;

-- Gets sub features name (all features of this parent sub tree are included)
```

```
-- Return: Set of strings
public features: () ==> set of string
	features() == return {name};

-- Gets all invalid subsets of configs associated with this feature (all features of
this parent sub tree are included)
-- Return: Set of configs
public invalidSubsets: () ==> set of config
	invalidSubsets() ==
	(
	dcl restrictions : set of config := getReqAndExcRestrictions();
	return restrictions;
	);

-- Gets invalid subsets related with requirements and exclusions
-- Return: Set of config as invalid subsets
public getReqAndExcRestrictions: () ==> set of config
	getReqAndExcRestrictions() ==
	(
	dcl restrictions : set of config := {};
	if (card requirements + card exclusions > 0) then
	(
			for all requirement in set requirements do restrictions :=
restrictions union {{name |-> true} munion {requirement.getName() |-> false}};
			for all exclusion in set exclusions do restrictions :=
restrictions union {{name |-> true} munion {exclusion.getName() |-> true}};
	);
	return restrictions;
	);

-- Gets feature node count (since there is no subfeatures its always 1, itself)
-- Return: Integer as node count
public nodeCount: () ==> int
	nodeCount() == return 1;

-- Checks if a feature is mandatory
-- Return: Boolean with mandatory status (True if mandatory, false if optional)
public isMandatory: () ==> bool
	isMandatory() == return mandatory;

end Feature
```

3.3 Class Model

This class defines a Model.

```
-- Class Model, conatining the root feature
class Model
types
	public config = Utilities`config;
	public string = Utilities`string;
instance variables
```

```
	protected root: Feature;
	private features : set of string := {};

operations

-- Model constructor
-- 1st arg: Root feature
public Model: Feature ==> Model
	Model(r) == (root := r; return self;)
	post root = r;

-- Gets model root
-- Return: Root feature
public getRoot: () ==> Feature
	getRoot() == (return root;);

-- Gets node count of model tree
-- Return: Integer with node count
public nodeCount: () ==> int
	nodeCount() == return root.nodeCount();

-- Gets tree features name
public setFeatures: () ==> ()
	setFeatures() == (
			features := root.features();
	);

-- Generates all valid configurations
-- Return: Sequence of configs
public generateValidConfigs: () ==> set of set of string
		generateValidConfigs() == (setFeatures(); return new
ConfigSearcher(root).getValidConfigs());

end Model
```

3.4 Class Parent

This class defines a Parent (a subclass of Feature).

```
-- Class Parent, a subclass of Feature
class Parent is subclass of Feature
types
	public parentType = Utilities`parentType;
instance variables
	private subFeatures: set of Feature := {};
	private type: parentType := <defaultParent>;

operations

-- Parent constructor
```

```
-- 1st arg: Name to be associated with the feature
public Parent: string ==> Parent
	Parent(n) ==
	(name := n; return self;)
	post name = n;

-- Sets Parent type
-- 1st arg: Custom type (<defaultParent>, <orParent>, <xorParent>)
public setParentType: parentType ==> ()
	setParentType(t) ==
	(type := t; if t = <orParent> or t = <xorParent> then
setSubFeaturesOptional())
	pre type <> <orParent> and type <> <xorParent>;

-- Gets Parent type
-- Return: Current Parent type
public getParentType: () ==> parentType
	getParentType() == return type;

-- Checks if Parent is a xorParent
-- Return: Boolean with status
public isXorParent: () ==> bool
	isXorParent() == return type = <xorParent>;

-- Checks if Parent is an orParent
-- Return: Boolean with status
public isOrParent: () ==> bool
	isOrParent() == return type = <orParent>;

-- Checks if Parent is a defaultParent
-- Return: Boolean with status
public isDefaultParent: () ==> bool
	isDefaultParent() == return type = <defaultParent>;

-- Sets sub features of parent
-- 1st arg: Set of Features
public setSubFeatures: set of Feature ==> ()
setSubFeatures(s) ==
(
	subFeatures := s;
	if (type = <orParent> or type = <xorParent>) then
		setSubFeaturesOptional();
);

-- Sets sub features as optional features
public setSubFeaturesOptional: () ==> ()
	setSubFeaturesOptional() ==
	for all subFeature in set subFeatures do subFeature.setMandatory(false);

-- Gets sub features of a Parent
-- Return: Set of Features
public getSubFeatures: () ==> set of Feature
	getSubFeatures() == return subFeatures;
```

```
-- Gets sub features name
-- Return: Set of strings
public getSubFeaturesNames: () ==> set of string
	getSubFeaturesNames() == return {el.getName() | el in set subFeatures};

-- Gets sub features name (all features of this parent sub tree are included)
-- Return: Set of strings
public features: () ==> set of string
	features() == (dcl parentFeatures : set of string := {name};
	for all subFeature in set subFeatures do
			parentFeatures := parentFeatures union subFeature.features();
	return parentFeatures;);

-- Gets all invalid subsets of configs associated with this feature (all features of
this parent sub tree are included)
-- Return: Set of configs
public invalidSubsets: () ==> set of config
	invalidSubsets() ==
	(
	dcl restrictions : set of config := getReqAndExcRestrictions();
	for all subFeature in set subFeatures do
			restrictions := restrictions union
subFeature.invalidSubsets();
	return restrictions;
);

-- Gets sub all invalid subsets of configs associated with this feature (all features
of this parent sub tree are included)
-- Return: Integer as child nodes count
public nodeCount: () ==> int
	nodeCount() == (dcl childNodeCount : int := 1;
			for all subFeature in set subFeatures do
				childNodeCount := childNodeCount +
subFeature.nodeCount();
	return childNodeCount;);

end Parent
```

3.5 Class Utilities

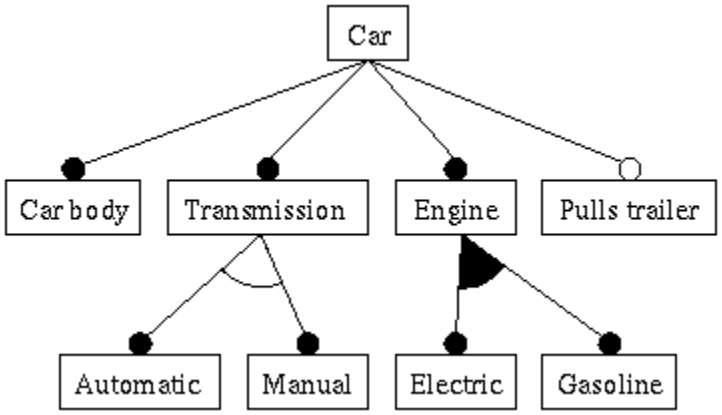
Contains utility types.

```
-- Class containing utilities
class Utilities
types
	public string = seq1 of char;
	public config = map string to bool;
	public parentType = <orParent> | <xorParent> | <defaultParent>;
end Utilities
```

4. Model Validation

4.1 Class CarModelTester

In this model we've added two sub features into the **pulls trailer** node, and one of those features has a sub feature too. In this case we are testing an optional sub tree.




```
-- Class defined to test the Car Model
class CarModelTester is subclass of FeatureModelTester
instance variables
	car : Parent := new Parent("car");
	body : Feature := new Feature("body");
	transmission : Parent := new Parent("transmission");
	automatic : Feature := new Feature("automatic");
	manual : Feature := new Feature("manual");
	pullsTrailler : Parent := new Parent("pullsTrailler");
	heavyTrailler : Parent := new Parent("heavyTrailler");
	armor : Feature := new Feature("armor");
	lightTrailler : Feature := new Feature("lightTrailler");
	engine : Parent := new Parent("engine");
	gasoline : Feature := new Feature("gasoline");
	electric : Feature := new Feature("electric");
```

```
operations

-- Creates model and defines restrictions
-- Return: Model as new model
public createModel: () ==> Model
       createModel() ==
       (
       dcl model : Model := new Model(car);

       pullsTrailler.setParentType(<xorParent>);
       transmission.setParentType(<xorParent>);
       engine.setParentType(<orParent>);

       armor.setMandatory(false);
       pullsTrailler.setMandatory(false);
       lightTrailler.setMandatory(false);
       heavyTrailler.setMandatory(false);

       car.setSubFeatures({body, transmission, pullsTrailler, engine});
       transmission.setSubFeatures({automatic, manual});
       engine.setSubFeatures({gasoline, electric});
       pullsTrailler.setSubFeatures({lightTrailler, heavyTrailler});
       heavyTrailler.setSubFeatures({armor});


       model.setFeatures();

       return model;
       );

-- Tests the Car Model
-- Note: The Car Model scheme is part of the report
public testModel: () ==> ()
  testModel() ==
  (

       dcl carModel : Model := createModel();

       -- Parent type tests

       testParentType(pullsTrailler, <xorParent>);
       testParentType(transmission, <xorParent>);
       testParentType(engine, <orParent>);
       testParentType(car, <defaultParent>);
       testParentType(heavyTrailler, <defaultParent>);

       -- Sub features tests

       testSubFeatures(car, {body, transmission, pullsTrailler, engine});
       testSubFeatures(transmission, {automatic, manual});
       testSubFeatures(engine, {gasoline, electric});
       testSubFeatures(pullsTrailler, {lightTrailler, heavyTrailler});
       testSubFeatures(heavyTrailler, {armor});

       -- Mandatory and optional features tests
```

```
        -- Note: In order to build strong test cases we're gonna consider the "screen"
feature as optional

        testMandatoryFeature(car, true);
        testMandatoryFeature(body, true);
        testMandatoryFeature(transmission, true);
        testMandatoryFeature(automatic, false);
        testMandatoryFeature(manual, false);
        testMandatoryFeature(pullsTrailler, false);
        testMandatoryFeature(heavyTrailler, false);
        testMandatoryFeature(lightTrailler, false);
        testMandatoryFeature(armor, false);
        testMandatoryFeature(engine, true);
        testMandatoryFeature(gasoline, false);
        testMandatoryFeature(electric, false);

        -- Requirements tests

        testRequirements(car, {});
        testRequirements(body, {});
        testRequirements(transmission, {});
        testRequirements(automatic, {});
        testRequirements(manual, {});
        testRequirements(pullsTrailler, {});
        testRequirements(heavyTrailler, {});
        testRequirements(lightTrailler, {});
        testRequirements(armor, {});
        testRequirements(engine, {});
        testRequirements(gasoline, {});
        testRequirements(electric, {});

        -- Exclusions tests

        testExclusions(car, {});
        testExclusions(body, {});
        testExclusions(transmission, {});
        testExclusions(automatic, {});
        testExclusions(manual, {});
        testExclusions(pullsTrailler, {});
        testExclusions(heavyTrailler, {});
        testExclusions(lightTrailler, {});
        testExclusions(armor, {});
        testExclusions(engine, {});
        testExclusions(gasoline, {});
        testExclusions(electric, {});

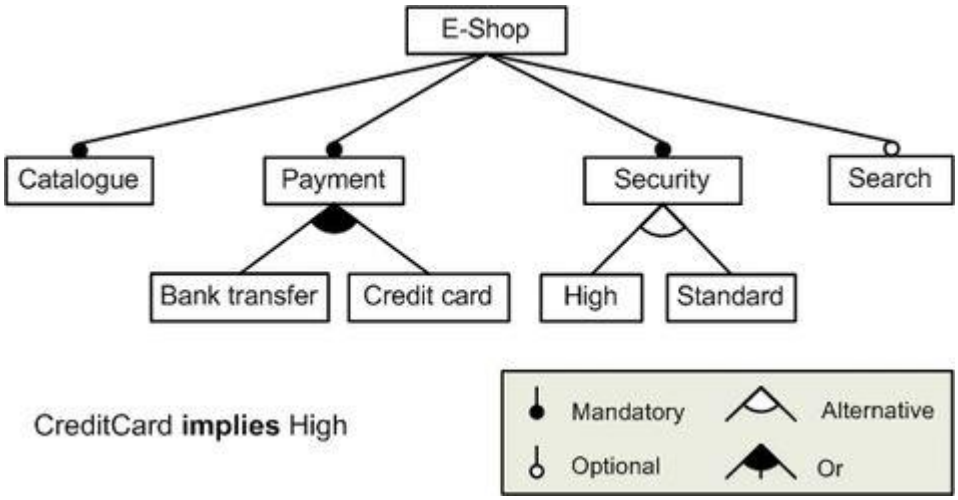
        -- Model tests

        testModelRoot(carModel, car);
        testModelFeaturesCount(carModel, 12);
        validModelConfig(carModel, {"armor", "automatic", "body", "car", "electric",
"engine", "gasoline", "heavyTrailler", "pullsTrailler", "transmission"});
  invalidModelConfig(carModel, {"car"});
        testGeneratedValidConfigs(carModel, carValidConfigs());
);
```

```
functions
-- All Car Model valid configurations
-- Return: Set of Car Model valid configurations
public carValidConfigs: () -> set of set of string
       carValidConfigs() ==       {{"armor", "automatic", "body", "car", "electric",
"engine", "gasoline", "heavyTrailler", "pullsTrailler", "transmission"},
{"armor", "automatic", "body", "car", "electric", "engine", "heavyTrailler",
"pullsTrailler", "transmission"},
{"armor", "automatic", "body", "car", "engine", "gasoline", "heavyTrailler",
"pullsTrailler", "transmission"},
{"armor", "body", "car", "electric", "engine", "gasoline", "heavyTrailler", "manual",
"pullsTrailler", "transmission"},
{"armor", "body", "car", "electric", "engine", "heavyTrailler", "manual",
"pullsTrailler", "transmission"},
{"armor", "body", "car", "engine", "gasoline", "heavyTrailler", "manual",
"pullsTrailler", "transmission"},
{"automatic", "body", "car", "electric", "engine", "gasoline", "heavyTrailler",
"pullsTrailler", "transmission"},
{"automatic", "body", "car", "electric", "engine", "gasoline", "lightTrailler",
"pullsTrailler", "transmission"},
{"automatic", "body", "car", "electric", "engine", "gasoline", "transmission"},
{"automatic", "body", "car", "electric", "engine", "heavyTrailler", "pullsTrailler",
"transmission"},
{"automatic", "body", "car", "electric", "engine", "lightTrailler", "pullsTrailler",
"transmission"},
{"automatic", "body", "car", "electric", "engine", "transmission"},
{"automatic", "body", "car", "engine", "gasoline", "heavyTrailler", "pullsTrailler",
"transmission"},
{"automatic", "body", "car", "engine", "gasoline", "lightTrailler", "pullsTrailler",
"transmission"},
{"automatic", "body", "car", "engine", "gasoline", "transmission"},
{"body", "car", "electric", "engine", "gasoline", "heavyTrailler", "manual",
"pullsTrailler", "transmission"},
{"body", "car", "electric", "engine", "gasoline", "lightTrailler", "manual",
"pullsTrailler", "transmission"},
{"body", "car", "electric", "engine", "gasoline", "manual", "transmission"},
{"body", "car", "electric", "engine", "heavyTrailler", "manual", "pullsTrailler",
"transmission"},
{"body", "car", "electric", "engine", "lightTrailler", "manual", "pullsTrailler",
"transmission"},
{"body", "car", "electric", "engine", "manual", "transmission"},
{"body", "car", "engine", "gasoline", "heavyTrailler", "manual", "pullsTrailler",
"transmission"},
{"body", "car", "engine", "gasoline", "lightTrailler", "manual", "pullsTrailler",
"transmission"},
{"body", "car", "engine", "gasoline", "manual", "transmission"}}
end CarModelTester
```

4.2 Class EshopModelTester

Class defined to test the E-shop model, described below.




```
-- Class defined to test the E-shop Model
class EshopModelTester is subclass of FeatureModelTester
instance variables
	eshop : Parent := new Parent("e-shop");
	catalogue : Feature := new Feature("catalogue");
	payment : Parent := new Parent("payment");
	security : Parent := new Parent("security");
	search : Feature := new Feature("search");
	bankTransfer : Feature := new Feature("bank transfer");
	creditCard : Feature := new Feature("credit card");
	high : Feature := new Feature("high");
	standard : Feature := new Feature("standard");
operations

-- Creates model and defines restrictions
-- Return: Model as new model
public createModel: () ==> Model
	createModel() ==
	(
		dcl model : Model := new Model(eshop);

		payment.setParentType(<orParent>);
		security.setParentType(<xorParent>);

		payment.setSubFeatures({bankTransfer, creditCard});
		security.setSubFeatures({high, standard});
		eshop.setSubFeatures({catalogue, payment, security, search});

		search.setMandatory(false);

		creditCard.setRequirements({high});

		model.setFeatures();

		return model;
```

```
      );

-- Tests the E-shop Model
-- Note: The E-shop Model scheme is part of the report
public testModel: () ==> ()
  testModel() ==
  (

    dcl eshopModel : Model := createModel();

                                        -- Parent type tests
      -- description: Since the default operation is to set the parent as a
<defaultParent>, eshop should have the <defaultParent> type
      -- note: Only these 3 Features are Parents (a subclass of Feature)

      testParentType(payment, <orParent>);
      testParentType(security, <xorParent>);
      testParentType(eshop, <defaultParent>);

      -- Sub features tests
      -- description: Each parent should have the respective features as subFeatures

      testSubFeatures(payment, {bankTransfer, creditCard});
      testSubFeatures(security, {high, standard});
      testSubFeatures(eshop, {catalogue, payment, security, search});

      -- Mandatory and optional features tests
      -- description: The default operation is to set any feature as mandatory, the
"search" and features with xor and or parents should be optional

      testMandatoryFeature(eshop, true);
      testMandatoryFeature(catalogue, true);
      testMandatoryFeature(payment, true);
      testMandatoryFeature(security, true);
      testMandatoryFeature(search, false);
      testMandatoryFeature(bankTransfer, false);
      testMandatoryFeature(creditCard, false);
      testMandatoryFeature(high, false);
      testMandatoryFeature(standard, false);

      -- Requirements tests
      -- description: By default a Feature doesn't have requirements, so only the
"creditCard" should have the "high" feature as requirement

      testRequirements(eshop, {});
      testRequirements(catalogue, {});
      testRequirements(payment, {});
      testRequirements(security, {});
      testRequirements(search, {});
      testRequirements(bankTransfer, {});
      testRequirements(creditCard, {high});
      testRequirements(high, {});
      testRequirements(standard, {});

      -- Exclusions tests
```

```
        -- description: By default a Feature doesn't have exclusions

        testExclusions(eshop, {});
        testExclusions(catalogue, {});
        testExclusions(payment, {});
        testExclusions(security, {});
        testExclusions(search, {});
        testExclusions(bankTransfer, {});
        testExclusions(creditCard, {});
        testExclusions(high, {});
        testExclusions(standard, {});

        -- Model tests
        -- description: By default a Feature doesn't have requirements, so only the
"creditCard" should have the "high" feature as requirement

        testModelRoot(eshopModel, eshop);
        testModelFeaturesCount(eshopModel, 9);
        validModelConfig(eshopModel, {"bank transfer", "catalogue", "credit card", "e-
shop", "high", "payment", "search", "security"});
    invalidModelConfig(eshopModel, {"e-shop"});
        testGeneratedValidConfigs(eshopModel, eshopValidConfigs());
);
functions
-- All Eshop Model valid configurations
-- Return: Set of Eshop Model valid configurations
public eshopValidConfigs: () -> set of set of string
        eshopValidConfigs() == {{"bank transfer", "catalogue", "credit card", "e-
shop", "high", "payment", "search", "security"},
{"bank transfer", "catalogue", "credit card", "e-shop", "high", "payment",
"security"},
{"bank transfer", "catalogue", "e-shop", "high", "payment", "search", "security"},
{"bank transfer", "catalogue", "e-shop", "high", "payment", "security"},
{"bank transfer", "catalogue", "e-shop", "payment", "search", "security",
"standard"},
{"bank transfer", "catalogue", "e-shop", "payment", "security", "standard"},
{"catalogue", "credit card", "e-shop", "high", "payment", "search", "security"},
{"catalogue", "credit card", "e-shop", "high", "payment", "security"}};
end EshopModelTester
```

4.3 Class MobilePhoneModelTester

Class to test the Mobile Phone Model described below.

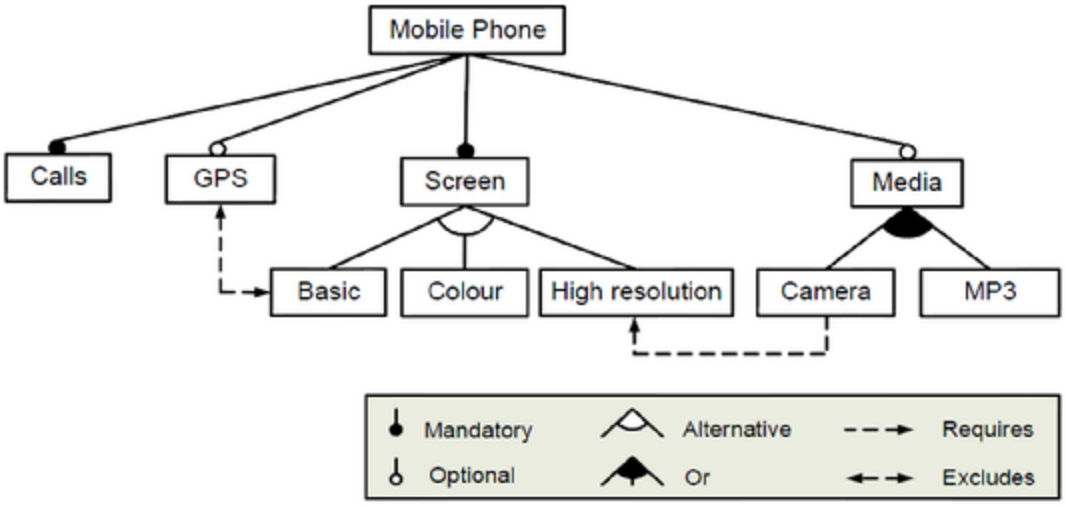


Figure 1: A sample feature model

```
-- Class defined to test the Mobile Phone Model
class MobilePhoneModelTester is subclass of FeatureModelTester
instance variables
	mobilePhone : Parent := new Parent("mobile phone");
	calls : Feature := new Feature("calls");
	gps : Feature := new Feature("gps");
	screen : Parent := new Parent("screen");
	media : Parent := new Parent("media");
	basic : Feature := new Feature("basic");
	colour : Feature := new Feature("colour");
	highResolution : Feature := new Feature("high resolution");
	camera : Feature := new Feature("camera");
	mp3 : Feature := new Feature("mp3");
operations

-- Creates model and defines restrictions
-- Return: Model as new model
public createModel: () ==> Model
	createModel() ==
	(
	dcl model : Model := new Model(mobilePhone);

	media.setParentType(<orParent>);
	screen.setParentType(<xorParent>);

	screen.setSubFeatures({basic, colour, highResolution});
	media.setSubFeatures({camera, mp3});
	mobilePhone.setSubFeatures({media, calls, gps, screen});

	gps.setMandatory(false);
	media.setMandatory(false);

	camera.setRequirements({highResolution});
```

```
        gps.setExclusions({basic});

        model.setFeatures();

        return model;
        );

-- Tests the Mobile Phone Model
-- Note: The Mobile Phone Model scheme is part of the report
public testModel: () ==> ()
  testModel() ==
  (

        dcl mobilePhoneModel: Model := createModel();

        -- Parent type tests

        testParentType(media, <orParent>);
        testParentType(screen, <xorParent>);
        testParentType(mobilePhone, <defaultParent>);

        -- Sub features tests

        testSubFeatures(screen, {basic, colour, highResolution});
        testSubFeatures(media, {camera, mp3});
        testSubFeatures(mobilePhone, {media, calls, gps, screen});

        -- Mandatory and optional features tests
        -- Note: In order to build strong test cases we're gonna consider the "screen"
feature as optional

        testMandatoryFeature(mobilePhone, true);
        testMandatoryFeature(calls, true);
        testMandatoryFeature(gps, false);
        testMandatoryFeature(screen, true);
        testMandatoryFeature(media, false);
        testMandatoryFeature(basic, false);
        testMandatoryFeature(colour, false);
        testMandatoryFeature(highResolution, false);
        testMandatoryFeature(camera, false);
        testMandatoryFeature(mp3, false);

        -- Requirements tests

        testRequirements(mobilePhone, {});
        testRequirements(calls, {});
        testRequirements(gps, {});
        testRequirements(screen, {});
        testRequirements(media, {});
        testRequirements(basic, {});
        testRequirements(colour, {});
        testRequirements(highResolution, {});
        testRequirements(camera, {highResolution});
        testRequirements(mp3, {});
```

```
        -- Exclusions tests

        testExclusions(mobilePhone, {});
        testExclusions(calls, {});
        testExclusions(gps, {basic});
        testExclusions(screen, {});
        testExclusions(media, {});
        testExclusions(basic, {});
        testExclusions(colour, {});
        testExclusions(highResolution, {});
        testExclusions(camera, {});
        testExclusions(mp3, {});

        -- Model tests

        testModelRoot(mobilePhoneModel, mobilePhone);
        testModelFeaturesCount(mobilePhoneModel, 10);
        validModelConfig(mobilePhoneModel, {"basic", "calls", "mobile phone",
"screen"});
  invalidModelConfig(mobilePhoneModel, {"mobile phone"});
        testGeneratedValidConfigs(mobilePhoneModel, mobilePhoneValidConfigs());
        );

functions
-- All Mobile Phone Model valid configurations
-- Return: Set of Mobile Phone Model valid configurations
public mobilePhoneValidConfigs: () -> set of set of string
        mobilePhoneValidConfigs() == {{"basic", "calls", "media", "mobile phone",
"mp3", "screen"},
{"basic", "calls", "mobile phone", "screen"},
{"calls", "camera", "gps", "high resolution", "media", "mobile phone", "mp3",
"screen"},
{"calls", "camera", "gps", "high resolution", "media", "mobile phone", "screen"},
{"calls", "camera", "high resolution", "media", "mobile phone", "mp3", "screen"},
{"calls", "camera", "high resolution", "media", "mobile phone", "screen"},
{"calls", "colour", "gps", "media", "mobile phone", "mp3", "screen"},
{"calls", "colour", "gps", "mobile phone", "screen"},
{"calls", "colour", "media", "mobile phone", "mp3", "screen"},
{"calls", "colour", "mobile phone", "screen"},
{"calls", "gps", "high resolution", "media", "mobile phone", "mp3", "screen"},
{"calls", "gps", "high resolution", "mobile phone", "screen"},
{"calls", "high resolution", "media", "mobile phone", "mp3", "screen"},
{"calls", "high resolution", "mobile phone", "screen"}};
end MobilePhoneModelTester
```

4.4 Class FeatureModelTester

```
-- Main class of Test related functions
class FeatureModelTester
types
        public config = Utilities`config;
```

```
	public string = Utilities`string;
	public parentType = Utilities`parentType;
operations

-- tests true conditions
private assertTrue: bool ==> ()
  assertTrue(cond) == return
  pre cond;

-- tests false conditions
private assertFalse: bool ==> ()
  assertFalse(cond) == return
  pre not cond;

-- tests parent type
-- 1st arg: Parent to be tested
-- 2nd arg: Type to be tested
protected testParentType: Parent * parentType ==> ()
	testParentType(parent, type) == assertTrue(parent.getParentType() = type);

-- tests mandatory and optional features
-- 1st arg: Feature to be tested
-- 2nd arg: Mandatory status
protected testMandatoryFeature: Feature * bool ==> ()
	testMandatoryFeature(feature, isMandatory) == assertTrue(feature.isMandatory()
= isMandatory);

-- tests parents sub features
-- 1st arg: Parent to be tested
-- 2nd arg: Expected sub features
protected testSubFeatures: Parent * set of Feature ==> ()
	testSubFeatures(parent, subFeatures) == assertTrue(parent.getSubFeatures() =
subFeatures);

-- tests feature requirements
-- 1st arg: Feature to be tested
-- 2nd arg: Expected requirements
protected testRequirements: Feature * set of Feature ==> ()
	testRequirements(feature, requirements) ==
assertTrue(feature.getRequirements() = requirements);

-- tests feature exclusions
-- 1st arg: Feature to be tested
-- 2nd arg: Expected exclusions
protected testExclusions: Feature * set of Feature ==> ()
	testExclusions(feature, exclusions) == assertTrue(feature.getExclusions() =
exclusions);

-- tests model features count
-- 1st arg: Model to be tested
-- 2nd arg: Expected features count
protected testModelFeaturesCount: Model * nat ==> ()
	testModelFeaturesCount(model, count) == assertTrue(model.nodeCount() = count);

-- tests a model valid config
```

```
-- 1st arg: Model to be tested
-- 2nd arg: Valid config to be tested
protected validModelConfig: Model * set of string ==> ()
	validModelConfig(model, config) == assertTrue(config in set
model.generateValidConfigs());

-- tests a model invalid config
-- 1st arg: Model to be tested
-- 2nd arg: Invalid config to be tested
protected invalidModelConfig: Model * set of string ==> ()
	invalidModelConfig(model, config) == assertFalse(config in set
model.generateValidConfigs());

-- tests the valid configs generation of a given model
-- 1st arg: Model to be tested
-- 2nd arg: Set of possible configs
protected testGeneratedValidConfigs: Model * set of set of string ==> ()
 testGeneratedValidConfigs(model, validSet) ==
 (
	dcl generatedSet : set of set of string := model.generateValidConfigs();
	assertTrue(validSet union generatedSet = validSet);
 );

-- tests model root
-- 1st arg: Model to be tested
-- 2nd arg: Expected root
  protected testModelRoot: Model * Parent ==> ()
		testModelRoot(model, root) == assertTrue(model.getRoot() = root);

 -- INVALID invariants and pre/post conditions
 -- warning: Run one function at a time

 -- tests the same requirements and exclusions post condition
protected testRequirementsBeforeExclusions: () ==> ()
	testRequirementsBeforeExclusions() ==
	(
	let feature1 = new Feature("feature1"),
		feature2 = new Feature("feature2") in
		(feature1.setRequirements({feature2});
		feature1.setExclusions({feature2});
		);
	);

-- tests the same exclusions and requirements post condition
protected testExclusionsBeforeRequirements: () ==> ()
	testExclusionsBeforeRequirements() ==
	(
		let feature1 = new Feature("feature1"),
			feature2 = new Feature("feature2") in
			(feature1.setExclusions({feature2});
			feature1.setRequirements({feature2});
			);
	);

public static main: () ==> ()
```

```
  main() ==
  (
            dcl eshopModelTester : EshopModelTester := new EshopModelTester();
            dcl mobilePhoneModelTester: MobilePhoneModelTester := new
MobilePhoneModelTester();
            dcl carModelTester : CarModelTester := new CarModelTester();
      eshopModelTester.testModel();
      mobilePhoneModelTester.testModel();
      carModelTester.testModel();
  );
end FeatureModelTester
```

5. Model Verification

5.1 Invariant Verification

No.	PO Name	Type
61	Parent`nodeCount	state invariant holds

the code under analysis (with the relevant words underlined)

```
-- Gets sub all invalid subsets of configs associated with this feature (all features
of this parent sub tree are included)
-- Return: Integer as child nodes count
public nodeCount: () ==> int
nodeCount() == (dcl childNodeCount : int := 1;
	for all subFeature in set subFeatures do
		childNodeCount := childNodeCount + subFeature.nodeCount();
return childNodeCount;);
```

Since the field subFeatures is a field of the Parent class, it implies that those invariant conditions maintain.

5.2 Class Verification

No.	PO Name	Type
53	Model`Model	class is verified

the code under analysis (with the relevant words underlined)

```
-- Model constructor
-- 1st arg: Root feature
public Model: Feature ==> Model
	Model(r) == (root := r; configGenerator := new ConfigGenerator(self); return
self;)
	post root = r;
```

```
The received argument is defined as being a Parent, so we know that all roots
are defined as Features.
```

6. Code Generation

While we were developing this project, we were able to generate **Java** code from **Overture** by right clicking on the project folder, choose the **Code Generation** submenu and left click on **Generate Java**. We have also developed a **CLI**, to test all the project **functionalities**. Some images of the the **Java** interface:

```
************************************
*                                  *
*          Features Model          *
*                                  *
************************************

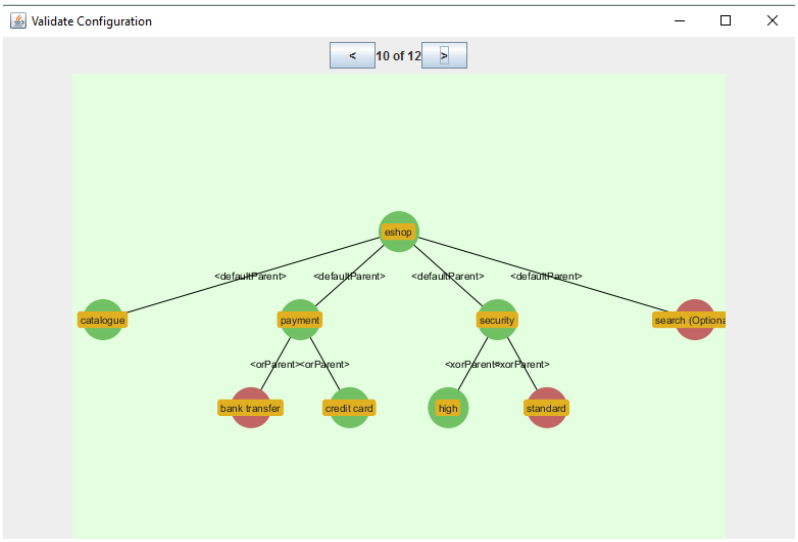
by Antonio Pedro Fraga, Francisco Rodrigues and Filipa Barroso


1. Load Feature Model
2. Generate valid configs
3. Validate config
4. Exit

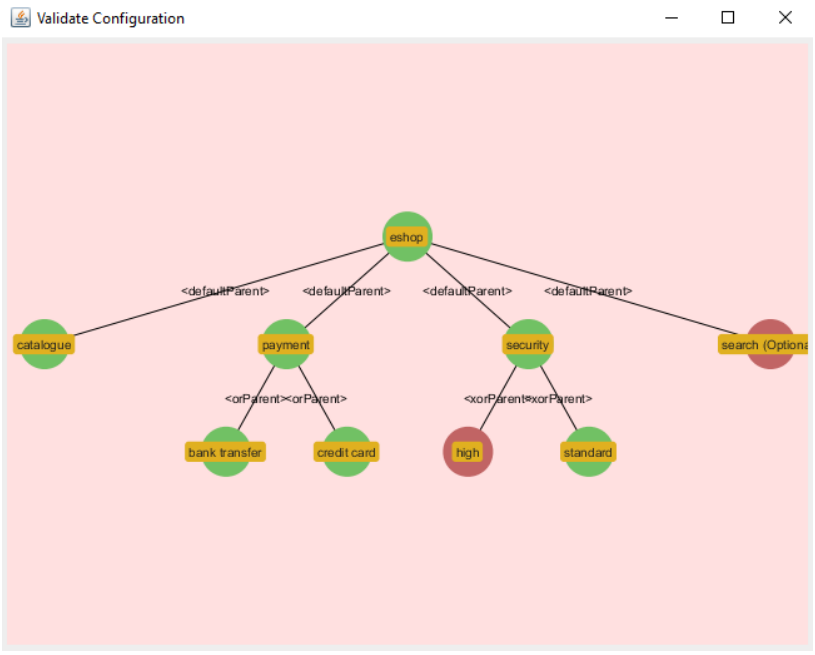
Insert option followed by enter:
```

There are four options available on the Java menu, each of these options can prove that one requirement is done.

When the “Load Feature Model” option is chosen, the user is able to choose a **json** file containing the information of a valid feature model. A **JFrame** is then prompted with a graph that represents the loaded model.



By choosing the "Generate Valid Configurations" option, the user is also able to choose a valid Features Model from a **json** file. After that, is possible to iterate through all the valid configurations by clicking the **left** and **right** arrow.



The option "Valid Configuration" is very similar to the previous option, in this case the user is able to choose the **json** file that represents a model, and a **json** file that represents a configuration. After that, a JFrame is prompted, the user is able to check whether the loaded configuration is valid or not. A red background corresponds to an invalid status, and a green one corresponds to a valid status.

7. Conclusions

The developed project covers all the requirements. Unit tests coverage is 100%, the Post / Pre conditions and invariants must be tested one at a time, so the coverage information related with those situations are not represented in this report. As future work we should improve the CLI, although we think that the developed interface has the required level.

This project took approximately 50 hours of development time. We think that we were supposed to take less time, but one of our goals was to develop optimized code.

To conclude, we are glad to have had the opportunity of developing this project.

8. References

- https://www.pure-systems.com/mediapool/tutorial_splwithfeaturemodelling.pdf
- http://overturetool.org/documentation/ (The documentation of the overture tool)
- https://en.wikipedia.org/wiki/Feature_model (Feature Model definition)
- https://moodle.up.pt/course/view.php?id=1678 (MFES moodle course page)