

Rapport de projet PSTL : Génération automatique de variantes d'IDE Eclipse

Julien MARGARIDO – Félix LIMA GORITO

M1 Informatique (STL) – Paris 6 UPMC

2016 / 2017

Table des matières

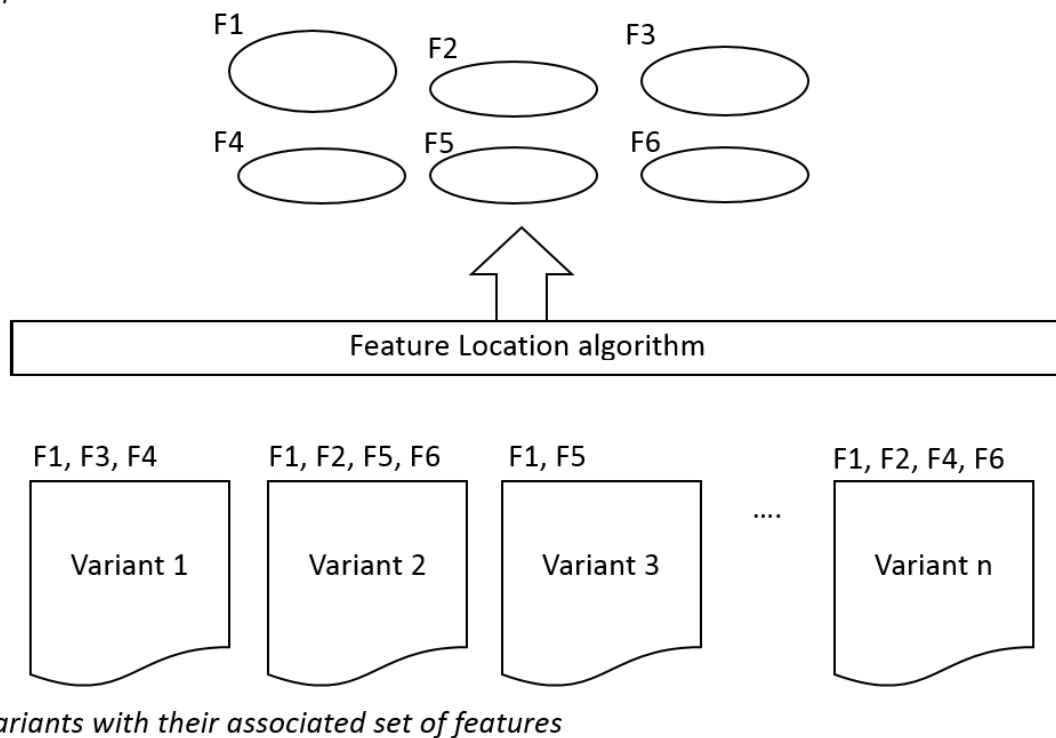
1. Introduction et contexte du projet	3
a. Feature Location	3
b. BUT4Reuse.....	4
c. Besoin d'évaluation des algorithmes	4
2. Les tâches principales : Génération de variantes	6
a. Génération aléatoire.....	6
b. Génération avec PLEDGE	11
3. Tests et mise en application.....	14
4. Objectifs et difficultés rencontrées	17
5. Conclusion.....	18
6. Annexes.....	19
7. Glossaire.....	19
8. Liens utiles et références	20

1. Introduction et contexte du projet

a. Feature Location

Dans le monde du développement logiciel, et plus particulièrement celui utilisant la technique du « Software Product Line Engineering* », une méthode permettant d'analyser et d'isoler chaque fonctionnalité (feature*) est d'utiliser des algorithmes de « Feature Location » (localisation de fonctionnalités). Ces algorithmes correspondent au processus de mise en avant du lien d'une fonctionnalité à son implémentation dans un produit.

Implementation elements associated to each Feature



Malheureusement, peu d'exemples concrets permettent de vérifier l'efficacité et empiriquement leurs forces et limites, d'où l'intérêt de comparer ces algorithmes, notamment à l'aide de benchmarks*.

* voir Glossaire

(x) voir Liens utiles

b. BUT4Reuse

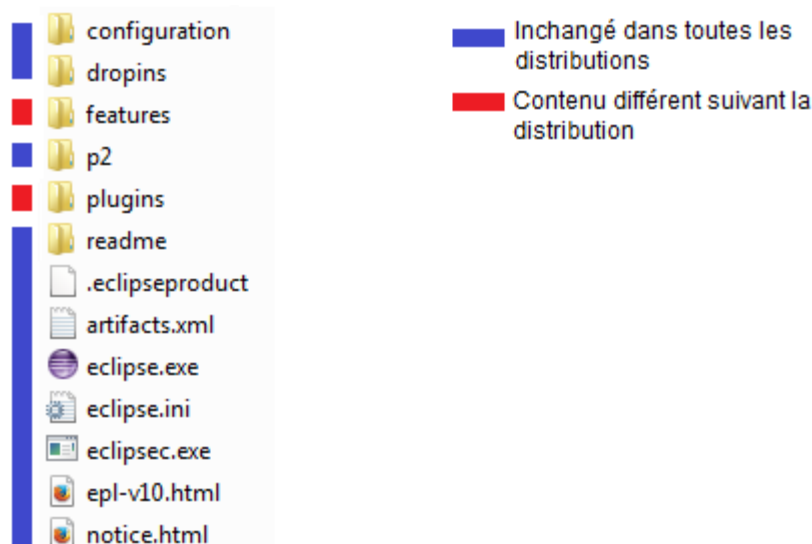
Le projet BUT4Reuse est une application développée conjointement par les départements de recherche du SNT (Security and Trust) de l'Université du Luxembourg et du LIP6 (Laboratoire Informatique de Paris 6) de l'Université Pierre et Marie Curie à Paris.

Elle implémente et intègre des algorithmes de Feature Location, et permet de les utiliser sur différentes structures (projets, textes, images, ...).

C'est un projet activement développé depuis fin 2013, entièrement en Java, et disponible sur Github⁽¹⁾.

c. Besoin d'évaluation des algorithmes

Un logiciel en libre distribution permettant de tester ces algorithmes est le logiciel Eclipse. De par sa modularité et ses différences/similitudes entre versions, c'est un outil intéressant à étudier, puisque sa structure orientée features (et basée sur des plugins*) correspond parfaitement aux critères du Software Product Line (SPL).



Architecture d'une distribution d'Eclipse

Cependant, ce logiciel ne possède (pour une version et un OS donné) que 12 distributions différentes, fournies par la fondation Eclipse⁽²⁾, comme la distribution Java, Team, Parallel, Automotive ou encore C/C++, ce qui est insuffisant pour une évaluation intensive.

D'où le projet qui nous a été proposé par nos encadrants, Tewfik Ziadi et Jabier Martinez, qui consiste à ajouter un composant dans le projet BUT4Reuse, permettant d'effectuer une génération automatique et paramétrable de distributions d'Eclipse, tout en respectant des exigences de contraintes.

Avant de pouvoir débiter la réalisation de ce projet, il fut nécessaire d'assimiler de nouvelles connaissances comme l'architecture d'Eclipse, le fonctionnement des features (les liens, les dépendances, ...), mais également les relations entre plugins et features.

2. Les tâches principales : Génération de variantes

Une fois l'environnement de travail prêt (dossier Google Drive et projet Github), le contexte et le sujet compris, nous avons pu commencer à intégrer directement les nouvelles fonctionnalités à BUT4Reuse.

a. Génération aléatoire

A la suite d'une réunion avec nos encadrants, nous avons décidé des différents paramètres concernant le premier type de génération de variantes.

Ainsi, nous devons créer une nouvelle option (sous forme d'un nouveau choix dans une liste apparaissant lors d'un clic droit), nommée "Generate Eclipse variants", qui ouvrirait une fenêtre permettant le choix des différents paramètres, qui sont :

- Le dossier d'entrée où se trouve la distribution d'Eclipse à utiliser
- Le dossier de sortie où les variantes seront générées
- Le nombre de variantes à créer
- Le pourcentage de choix d'une feature
- La possibilité de conserver uniquement les métadonnées des plugins

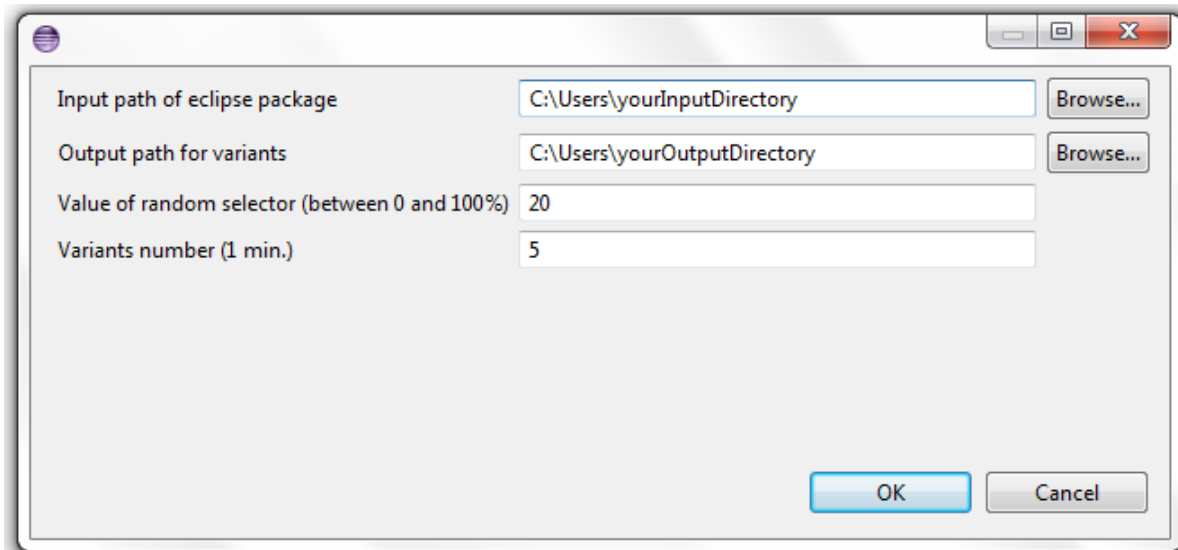
Le dossier d'entrée correspond à la distribution d'Eclipse sur laquelle vont se baser nos variantes.

L'option du pourcentage du choix d'une feature est important, car c'est sur cela que s'appuie une partie de notre projet. En effet, pour constituer une variante d'Eclipse, nous allons nous baser sur le fait que ce pourcentage servira à choisir ou non des features, de la distribution donnée en input, de manière aléatoire.

Concernant l'option des métadonnées, elle devait nous permettre de ne copier que les propriétés des plugins, sans conserver le reste (fichiers class, images, etc ...) ce qui allégeait considérablement les variantes générées (environ 90%).

Cependant, ces variantes d'Eclipse générées n'étaient plus exécutables, par conséquent nous avons décidé d'abandonner cette option.

Ainsi, nous obtenons ce menu :



A screenshot of a Windows-style dialog box titled "Eclipse package settings". It contains four input fields and two buttons. The first field, "Input path of eclipse package", has the value "C:\Users\yourInputDirectory" and a "Browse..." button. The second field, "Output path for variants", has the value "C:\Users\yourOutputDirectory" and a "Browse..." button. The third field, "Value of random selector (between 0 and 100%)", has the value "20". The fourth field, "Variants number (1 min.)", has the value "5". At the bottom right are "OK" and "Cancel" buttons.

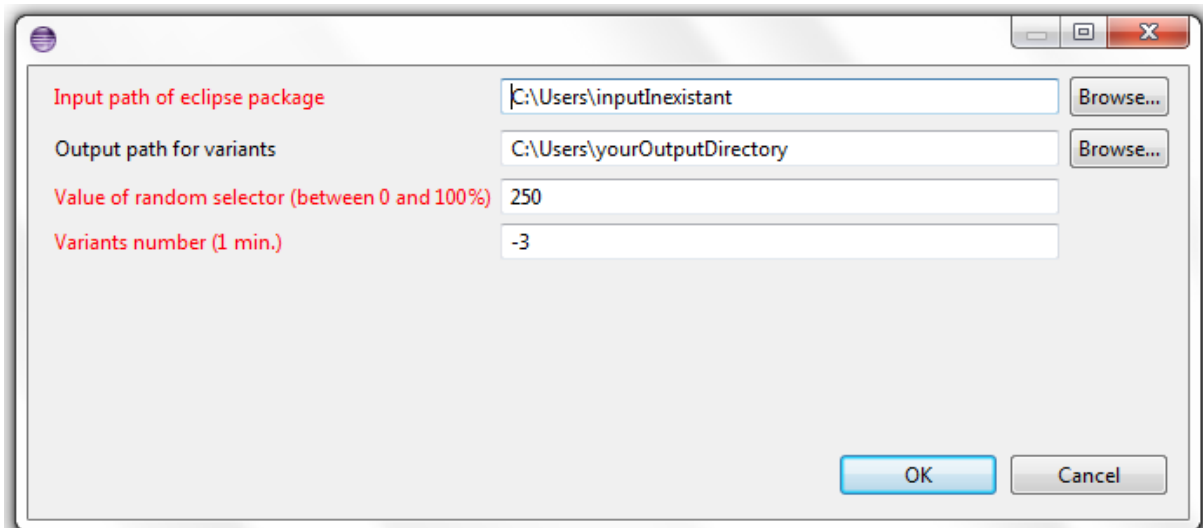
Input path of eclipse package	C:\Users\yourInputDirectory	Browse...
Output path for variants	C:\Users\yourOutputDirectory	Browse...
Value of random selector (between 0 and 100%)	20	
Variants number (1 min.)	5	

OK Cancel

Elle a été créée à l'aide de la bibliothèque SWT, en utilisant des éléments comme des Label, des Text, ou encore des GridLayout.

Deux boutons "Browse", faisant apparaître des DirectoryDialog, servent à parcourir les dossiers de l'utilisateur, dans le but de trouver aisément le input ou output recherché.

La validation, par le bouton "OK" de ce menu, entraîne une vérification des paramètres. Si cette vérification échoue, les paramètres erronés apparaissent en rouge :



A screenshot of the same dialog box as above, but with validation errors. The labels for "Input path of eclipse package", "Value of random selector (between 0 and 100%)", and "Variants number (1 min.)" are in red. The input values are "C:\Users\inputInexistant", "250", and "-3" respectively. The "Output path for variants" label and value remain black. The "Browse..." buttons and "OK/Cancel" buttons are still present.

Input path of eclipse package	C:\Users\inputInexistant	Browse...
Output path for variants	C:\Users\yourOutputDirectory	Browse...
Value of random selector (between 0 and 100%)	250	
Variants number (1 min.)	-3	

OK Cancel

Une fois tous les paramètres conformes, nous avons décidé (avec l'accord de nos encadrants) que ceux-ci seraient enregistrés dans un fichier de préférence, dans le but de gagner du temps à chaque nouvelle relance du menu.

Avec ces 4 paramètres, nous pouvons démarrer le premier type de génération : La génération aléatoire.

```
VariantsGenerator varGen = new VariantsGenerator(paramDialog.getInputPath(),  
    paramDialog.getOutputPath(), nbVariantsForThread, valRandForThread);  
  
varGen.addListener(context);  
  
varGen.generate();
```

Nous avons créé une classe VariantsGenerator. Celle-ci est instanciée en lui donnant en paramètres ceux définis dans le menu : le path* d'un input, le path d'un output, un nombre entier de variantes à générer, et un nombre entier aléatoire.

Cette classe, lors de son exécution, renvoie des informations sous forme de chaînes de caractères, que nous affichons sur une page de résumé, d'où l'importance de lier cette affichage avec VariantsGenerator, à l'aide d'une méthode addListener (utilisation du Design Pattern Require/Provide).

Lorsque notre générateur est instancié, et optionnellement lié à un affichage, alors l'appel à la fonction generate démarre le processus de création.

Celui-ci utilise principalement les path, URI de java.net et File de java.io.

Tout d'abord, nous récupérons toutes les features du Eclipse donné en input, à l'aide d'une classe existante de BUT4Reuse, FeatureHelper.

De la même manière, nous récupérons également une liste de tous les plugins de cet input.

Avec ces deux listes, nous lançons une procédure d'analyse des dépendances, grâce à notre composant DependenciesAnalyzer.

Celle-ci prend en paramètres la liste de tous les plugins et la liste de toutes les features d'un input, ainsi que l'URI de celui-ci.

Elle va s'occuper d'analyser et de créer le graphe des dépendances entre toutes ces features, et des plugins associés.

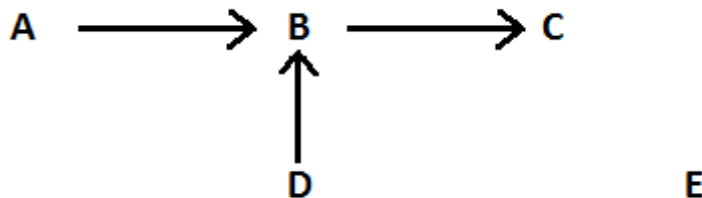
Il faut savoir qu'une feature est représentée dans le code de BUT4Reuse par la classe ActualFeature, et est composée d'un identifiant, d'un nom, d'une description et d'une liste de plugins (représentés par la classe PluginElement) dépendant de cette fonctionnalité.

De plus, elle est composée d'une éventuelle liste de features requis (required features), et une autre de features l'incluant (included features).

Feature	
<i>id:</i> org.eclipse.cvs	
<i>name:</i> Eclipse CVS Client	
<i>description:</i> Eclipse CVS Client (binary runtime and user documentation).	
Plugin id	Plugin name
org.eclipse.cvs	Eclipse CVS Client
org.eclipse.team.cvs.core	CVS Team Provider Core
org.eclipse.team.cvs.ssh2	CVS SSH2
org.eclipse.team.cvs.ui	CVS Team Provider UI

Exemple de la feature « CVS », implémentée par 4 plugins

Par exemple : Soient A, B, C, D et E 5 features.



Pour fonctionner, la feature A a besoin de la feature B : B est donc dans la liste des required de A, et A est dans la liste des included de B. De même pour B avec C, B avec D. Cependant, la feature E n'ayant aucune dépendance avec une autre feature, elle ne possède ni de required, ni de included.

De ce fait, lorsque l'on voudra choisir une feature, il faudra récupérer également toutes celles qui lui sont associées : Si l'on choisit C, nous devrons également récupérer A, B et D.

Ainsi, ce sont ces dépendances dont s'occupe la classe DependenciesAnalyzer. Au début du processus de génération des variantes, elle crée les graphes de toutes les features et plugins d'une distribution.

De plus, elle offre des fonctions utiles durant la génération :

- List<ActualFeature> **getFeaturesDependencies**(ActualFeature actual) : retourne la liste des features associées à une feature
- List<ActualFeature> **getFeaturesMandatoriesByInput**() : retourne la liste des features obligatoires pour qu'une variante d'Eclipse soit exécutable.
- List<PluginElement> **getPluginsDependencies**(ActualFeature actual) : retourne la liste des plugins d'une feature.
- List<PluginElement> **getPluginsWithoutAnyFeaturesDependencies**() : retourne la liste des plugins associés à aucune feature.

Une fois ces graphes générés, VariantsGenerator va créer les X variantes (avec X le paramètre correspondant au nombre de variantes).

Le premier traitement effectué est de supprimer les features (parmi la liste récupérée avec FeatureHelper) qui ont obtenu un score supérieur au nombre aléatoire indiqué en paramètre.

Ce score est calculé pour chaque feature, et est un nombre aléatoire entre 0 et 100.

Si ce score est supérieur au paramètre, alors cette feature ne sera pas prise en compte pour cette variante. Par exemple, si une feature obtient un score de 38, mais le paramètre aléatoire est de 20, alors celle-ci n'est pas conservée.

Si au contraire, elle l'est, nous récupérons toutes ses dépendances, à l'aide de la fonction getFeaturesDependencies de DependenciesAnalyzer, en vérifiant de ne pas récupérer des features déjà choisies ou récupérées (pour éviter les doublons). Une fois toutes les features obtenues, nous récupérons les plugins de chaque une (tout en refusant les doublons).

De ce fait, pour chaque variante, nous obtenons une partie des features et des plugins de la distribution donnée en input.

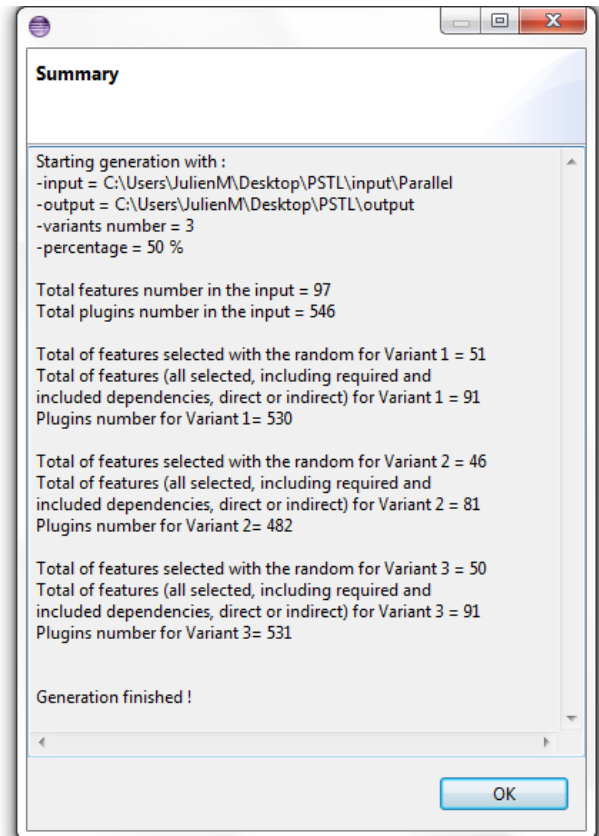
La composition de ces variantes n'est pas totalement proportionnelle au pourcentage choisi. En effet, si on lance une génération, basée sur une distribution possédant 600 features, avec un paramètre aléatoire de 50%, les variantes générées auront peu de chances de ne posséder que 300 features, ce nombre sera forcément supérieur, du fait qu'il faut prendre en compte les dépendances entre features (le paramètre aléatoire mis à 100% donnera toujours une variante identique à la distribution en input).

En utilisant la bibliothèque Commons IO, d'Apache, nous créons les variantes dans le output spécifié, nous y copions les fichiers/dossiers de configuration inchangé, et nous y copions les features et plugins sélectionnés lors de la génération.

Finalement, nous utilisons une fonction existante de BUT4Reuse :

```
adapter.construct(outputUri, allElements, new NullProgressMonitor());
```

Celle-ci va prendre en paramètre le output, la liste des plugins+fichiers de configuration, et va correctement paramétrer le fichier de configuration de la variante d'Eclipse, bundles.info, ce qui permettra que celle-ci soit exécutable.



Génération de 3 variantes, avec un pourcentage de 50%

De cette manière, nous avons créé un composant qui analyse les fonctionnalités et plugins d'une distribution, et crée une ou plusieurs variantes, en se basant sur les paramètres fournis.

b. Génération avec PLEDGE

La première partie de notre travail terminée, nos encadrants nous ont fourni une nouvelle mission qui consiste à utiliser une nouvelle stratégie pour la génération automatique. Celle-ci est l'utilisation d'une nouvelle librairie, développée par Monsieur Christopher Henard de l'Université du Luxembourg, qui est PLEDGE (Product Line Editor and tests GEneration tool).

PLEDGE est un outil java, permettant de créer un ensemble de configurations de features aléatoires qui sont dissimilaires entre elles, mais respectant toujours les contraintes entre les features.

Pour pouvoir fonctionner de manière optimale, PLEDGE a besoin de temps. En effet son algorithme étant long à s'exécuter, il a deux stratégies différentes à adopter :

- la première consiste à sélectionner au hasard les features qu'il va sélectionner, reprenant le même principe que notre première méthode de génération où nous sélectionnons les features selon un pourcentage, sauf qu'ici il est inconnu.
- la seconde méthode, quant à elle, va tenter de sélectionner les features les plus dissimilaires entre eux.

Afin d'utiliser ce nouvel outil, nous avons appris à créer un fichier XML au format SPLOT (Software Product Lines Online Tools).

Le format SPLOT est un format standardisé pour représenter un FeatureModel*. Le fichier XML crée permet de représenter les contraintes qu'il y a entre les différentes features, comme expliqué dans la partie précédente.

```
<feature_model name="Mobile phone ">
<meta>
<data name="description">Example Mobile Phone</data>
</meta>
<feature_tree>
:r Mobile_phone(_r)
  :m Calls(_r_1)
  :o GPS(_r_4)
  :m Screen(_r_5)
    :g (_r_5_6) [1,1]
      : Basic(_r_5_6_7)
      : Colour(_r_5_6_8)
      : High Resolution(_r_5_6_9)
  :o Media(_r_10)
    :g (_r_10_11) [1,*]
      : Camera(_r_10_11_12)
      : MP3(_r_10_11_13)
</feature_tree>
<constraints>
constraint_1:~_r_4 or ~_r_5_6_7
constraint_2:_r_5_6_9 or ~_r_10_11_12
</constraints>
</feature_model>
```

Exemple de fichier SPLOT pour un téléphone mobile

PLEDGE fourni par défaut une interface graphique, mais dans notre cas, nous n'avons pas besoin de celle-ci mais uniquement des méthodes offertes par l'outil, tel que le chargement d'un FeatureModel (sous format SPLOT) et de paramètres (comme le temps maximum alloué, ou la technique de hiérarchisation), et la récupération des résultats du traitement.

Comme dans la partie précédente où nous générions les variantes d'Eclipse avec un paramètre aléatoire, nous avons réutilisé le même principe du menu et de la fenêtre de dialogue, mais en remplaçant le pourcentage de sélection d'une feature par le temps à allouer à PLEDGE pour sa génération.

Un temps faible donnera une configuration proche d'une génération aléatoire, alors qu'à l'inverse, la génération donnera un choix de features le plus dissimilaire possible.

Ainsi, nous avons créé une fonction `exportToSPLOT(List<ActualFeature> allFeatures)`, dans une classe `SpotUtils`, qui crée et retourne le fichier SPLOT d'après une liste de features, tout en respectant les contraintes et dépendances. Le fichier XML sera stocké dans le dossier output, dans le but d'éventuellement vérifier si le fichier généré ne contient pas d'erreurs ou d'incohérences.

Le lancement de cette génération, par la classe `VariantsPledgeGenerator`, récupère donc ce fichier SPLOT, et le fourni en paramètre à PLEDGE.

```
ModelPLEDGE mp = new ModelPLEDGE();

try {
    mp.loadFeatureModel(f.getAbsolutePath(), FeatureModelFormat.SPLOT);
} catch (Exception e1) {
    sendToAll("Error in generator : Error to load FeatureModel.");
    e1.printStackTrace();
    return;
}

mp.setNbProductsToGenerate(nbVariants);
mp.setGenerationTimeMSAllowed(time * 1000L);
mp.setPrioritizationTechniqueByName("SimilarityGreedy");
try {
    mp.generateProducts();
} catch (Exception e1) {
    sendToAll("Error in generator : Error to generate the product.");
    e1.printStackTrace();
    return;
}
```

Les résultats du traitement de PLEDGE est retournée sous forme d'une liste de chiffre, où chaque chiffre représente l'identifiant de la feature dans l'objet `ModelPLEDGE` créé.

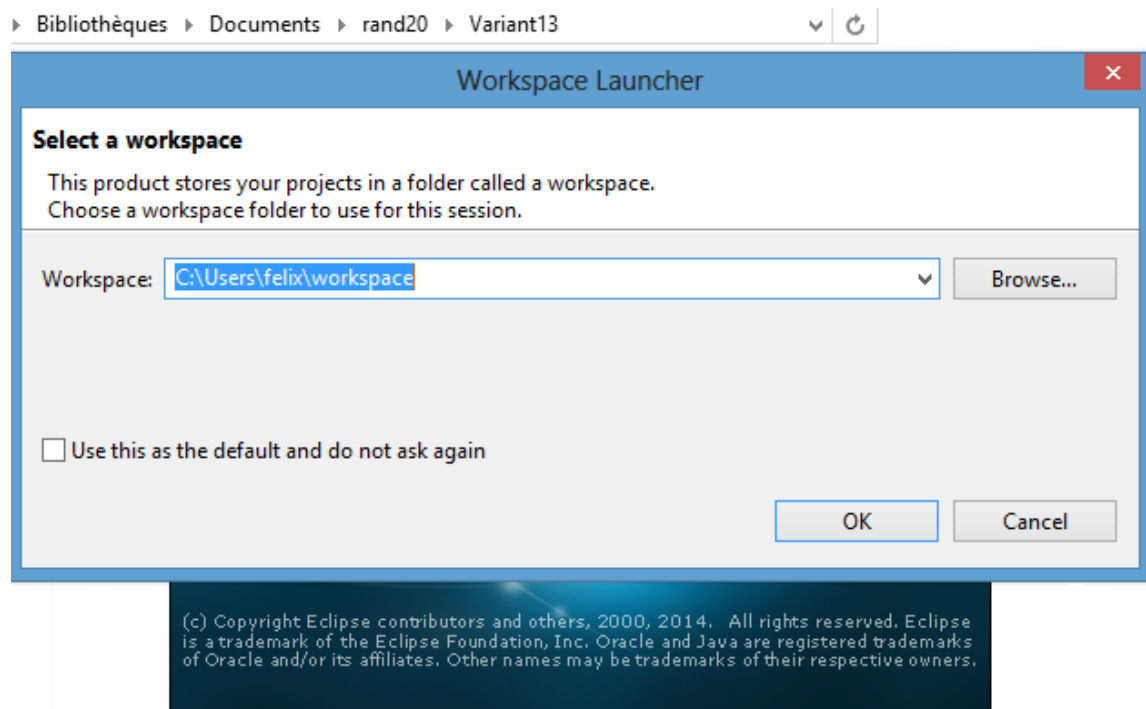
La liste ne représente pas uniquement son identifiant, mais indique, si la feature ayant cet identifiant doit être conservée (les identifiants de features ignorées sont retournés sous forme négative).

Avec tous ses éléments mis en place, nous avons pu procéder de la même manière que précédemment pour générer des variantes d'Eclipse fonctionnelles, en utilisant la classe `DependenciesAnalyzer`, en récupérant les plugins associés à chaque feature "conservée", et en transformant chaque variante créée en une distribution exécutable.

3. Tests et mise en application

Pour vérifier que notre travail est correct, nous avons décidé de générer de nombreuses variantes d'Eclipse en utilisant les 3 techniques, développées précédemment. Bien sûr, nous avons changé les paramètres à chaque génération, dans le but de couvrir un maximum de cas possibles.

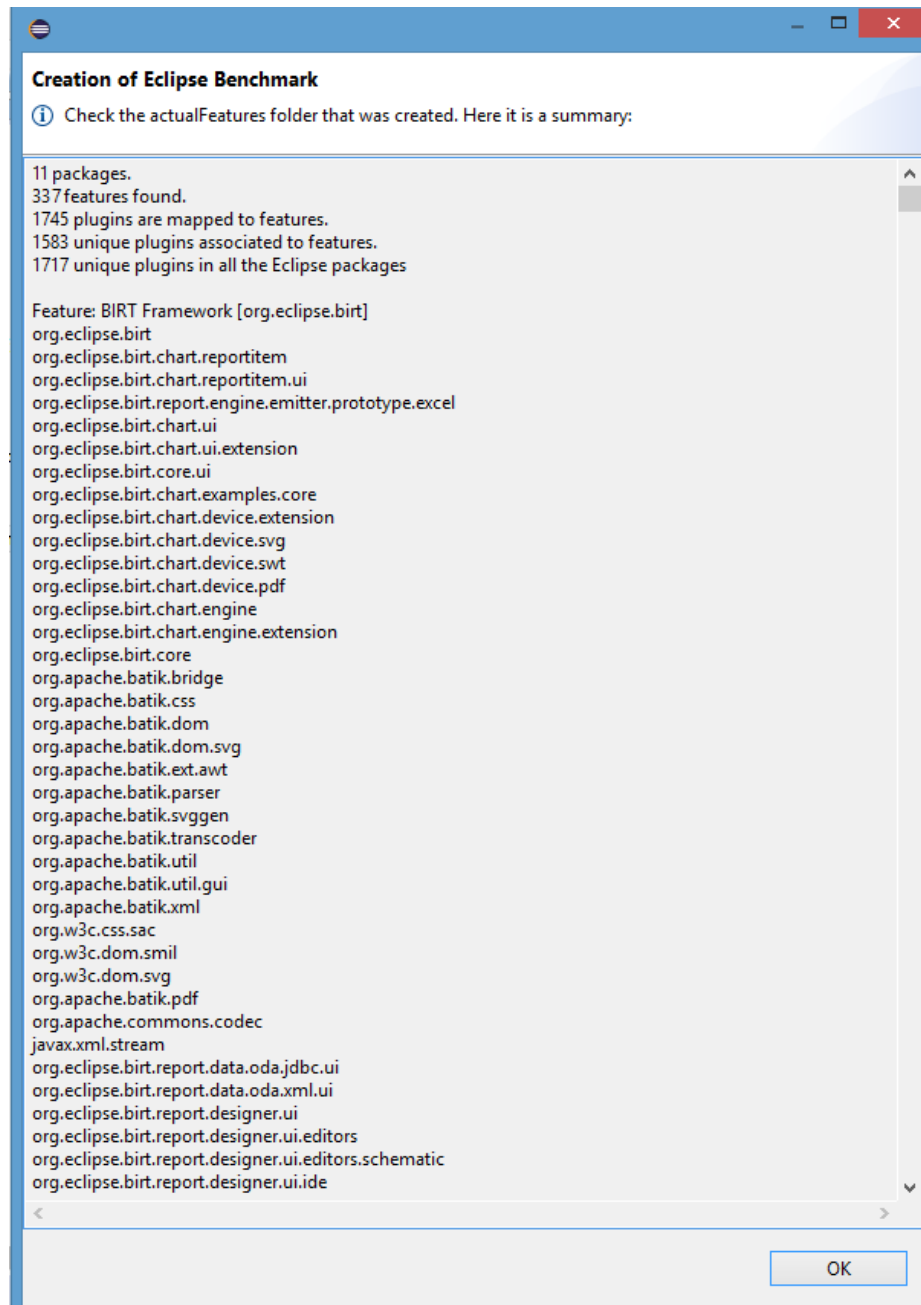
Nous avons pu observer que chaque variante était exécutable, montrant ainsi que la gestion des dépendances, ainsi que la configuration et la copie de la structure de base (la distribution en input) est correcte.



Lancement d'une variante générée avec un nombre aléatoire de 10%

Une fois la partie conception de notre projet terminée et validée, nous avons pu vérifier l'utilité et la conformité de celui-ci en lançant des tests, comme des benchmarks ou un algorithme de Feature Location, intégrés à BUT4Reuse.

Le benchmark va nous permettre de retrouver tous les détails relatifs à notre génération, tels que le nombre de features et de plugins contenu dans les X variantes générées (X étant le nombre de variantes à créer, indiqué dans la fenêtre de paramètres), ou encore les features de la même catégorie.



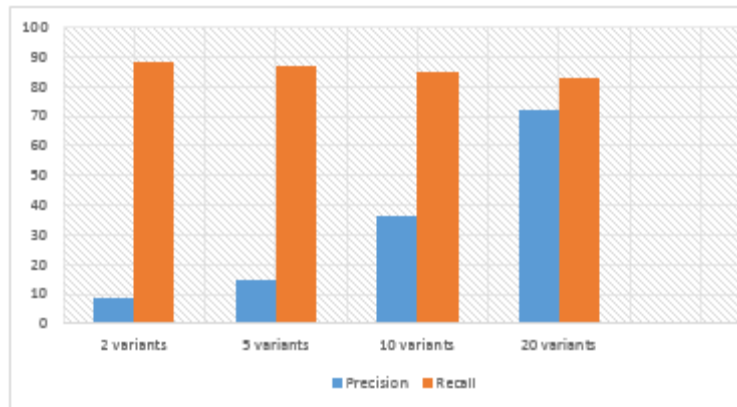
Benchmark effectué après la génération de 11 variantes.

Nous avons également lancé plusieurs tests de l'algorithme de Feature Location, présent dans EFLBench. Ces tests nous ont permis d'obtenir un graphique montrant les résultats de la "precision" et du "recall" de cet algorithme.

La "precision" correspond au pourcentage de plugins correctement récupérés via l'algorithme à partir de l'ensemble des plugins liés aux features.

Le "recall" correspond quant à lui au pourcentage du nombre de plugins correctement récupérés par rapport à l'ensemble des plugins non liés aux features. Pour plus d'informations, se référer au document (6).

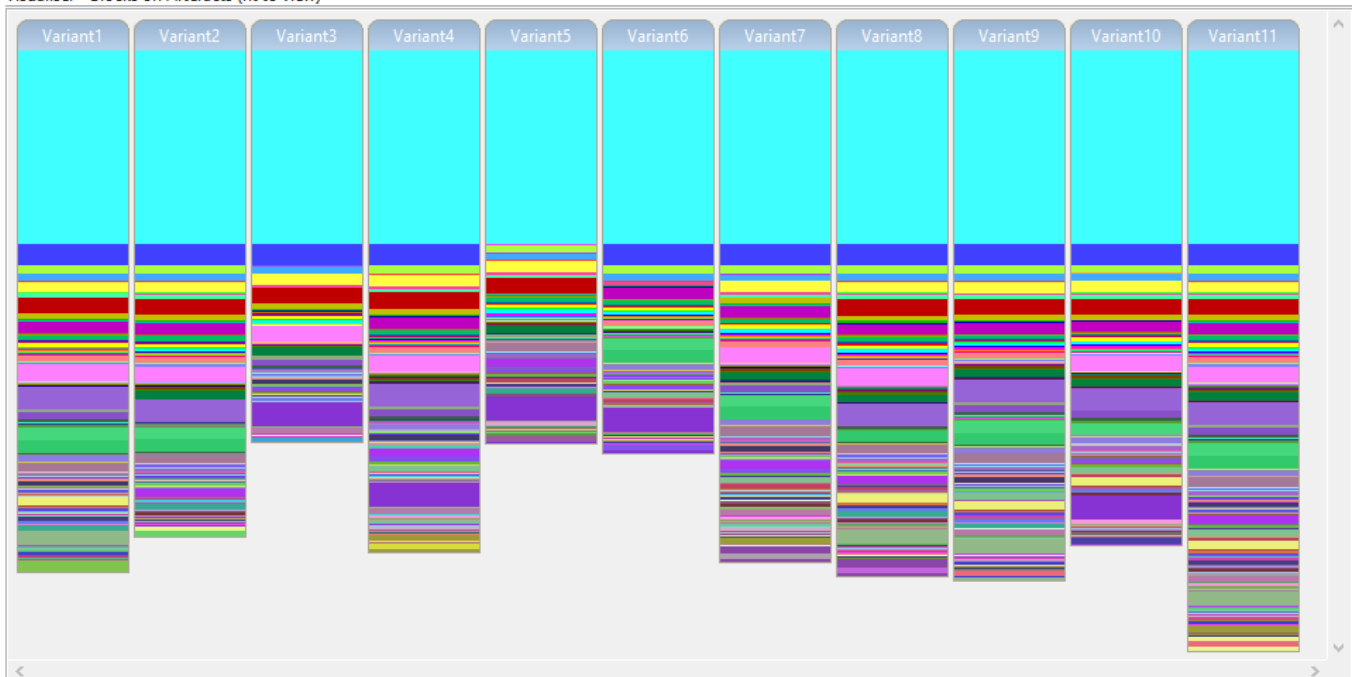
	2 variants	5 variants	10 variants	20 variants
Precision	9	15	36	72
Recall	88	87	85	83



Settings :
 Feature location technique -> Formal Concept Analysis
 Generation Techniques -> Percentage of features
 - Percentage: 10%
 - Input Eclipse -> Eclipse standard (x features)
 - Number of executions to calculate average: 10

Grphe modélisant plusieurs tests de Feature Location

Visualiser - Blocks on Artefacts (fit to view)



Processus de « Feature Identification » appliqué sur 11 variantes, chaque couleur représentant un ensemble de features

4. Objectifs et difficultés rencontrées

Nous nous étions fixé dès le départ des buts et des objectifs, primordiaux pour la réussite de notre projet :

- Réussir les tâches qui nous seraient assignées, dans les délais souhaités.
- Être un maximum autonome et efficace dans notre travail.
- Satisfaire les encadrants sur le plan humain, en apportant des réponses rapides, un travail rigoureux, une attitude concernée et irréprochable.
- Améliorer nos compétences techniques et engranger de nouvelles connaissances (technologies, « best practices », etc...).
- Gérer convenablement notre temps pour mener à bien tous nos projets universitaires.

La première difficulté rencontrée dans la réalisation de notre projet fut de comprendre le fonctionnement de l'application BUT4Reuse (application d'environ 34KloC*, composée de 45 plugins), et quel serait la place de notre travail dans celui-ci. Ce n'est qu'après une réunion avec nos encadrants et avec des démonstrations que nous avons saisi l'importance de notre tâche.

Un autre problème rencontré fut l'assimilation de nouvelles technologies. Certaines, comme SWT, étaient bien documentées, mais d'autres ne contenaient que peu de documentation, comme PLEDGE ou SPLOT. Pour comprendre le fonctionnement de celles-ci, nous avons eu à effectuer plusieurs tests sur chacune d'elles, quelque fois en repérant des contraintes non spécifiées (certains caractères interdits en SPLOT par exemple).

En plus de ces nouvelles technologies, il nous a fallu étudier et comprendre de nouvelles notions, dans un laps de temps court. L'une d'entre elles fut les features et leurs rôles primordiales pour la suite de notre projet, nous permettant ensuite de mieux comprendre ce que sont des FeaturesModel ou encore le principe des algorithmes de Feature Location.

Un point non-négligeable à aborder est la langue. En effet, Jabier Martinez étant un doctorant étranger, et tout le projet BUT4Reuse étant en anglais (code et documentations), nous avons eu à parler, lire et rédiger en anglais. Ce fut une difficulté au début, mais rapidement, cela nous est apparu comme un avantage, car cela nous exerçait, nous poussait à nous corriger et à vérifier mutuellement nos travaux et messages.

5. Conclusion

Ce projet en équipe fût très enrichissant, nous apportant beaucoup de connaissances, de rigueur, de pratique et de perspectives d'études. C'est un travail que nous avons pris fortement à cœur, en effectuant une grande partie des tâches demandées dès le début du projet, nous évitant une pression trop accrue (retards, erreurs, négligences, ...), et nous permettant d'effectuer d'autres travaux et améliorations supplémentaires.

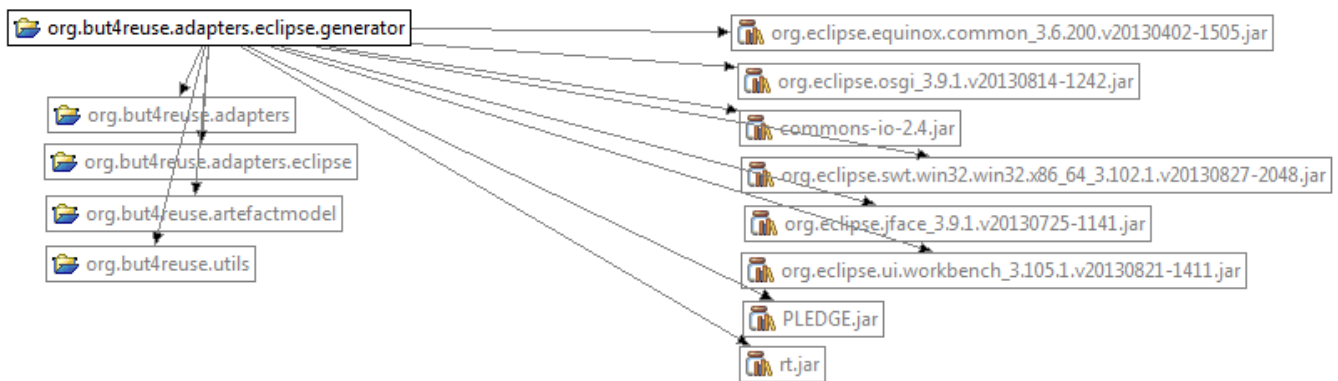
Concernant l'encadrement du projet, nous tenons fortement à remercier Tewfik Ziadi et Jabier Martinez, pour leur irréprochable suivi du projet, et leurs précieux conseils.

Nous avons pu échanger beaucoup avec eux (plus de 90 mails, et 6 réunions), souvent en anglais, et toujours dans un cadre chaleureux, accueillant, rigoureux et professionnel.

Notre projet, disponible sur Github⁽³⁾, est composé de plus de 2KLoC.

Nous avons effectué un « Pull Request* », qui a été accepté, signifiant que notre travail est actuellement intégré à la version officielle de BUT4Reuse, et plus concrètement une extension de EFLBench⁽⁴⁾.

6. Annexes



Plugins de BUT4Reuse et librairies utilisés par notre plugin eclipse.generator

7. Glossaire

- **Software Product Line Engineering** : En reprenant l'explication de Mr Tewfik Ziadi : "approche de développement dans le génie logiciel (transposé des chaînes de production industrielles) qui consiste à ne plus réaliser chaque logiciel indépendamment mais à partir d'éléments réutilisables." ⁽⁵⁾
- **Benchmark** : analyse permettant de mesurer les performances d'un produit
- **Feature** : fonctionnalité
- **KLoC** : 1,000 lignes de code
- **IDE** : environnement de développement (Integrated Development Environment)
- **Plugin** : module d'extension
- **Path** : chemin d'accès
- **Dissimilarité** : couvrir la majorité des features dans l'ensemble des variantes générées
- **FeatureModel** : technique de modélisation de Software Product Line

8. Liens utiles et références

(1) Github de BUT4Reuse : <https://github.com/but4reuse/but4reuse>

(2) Liste des distributions d'Eclipse Kepler :
<https://eclipse.org/downloads/packages/release/Kepler/SR2>

(3) Github de notre projet : <https://github.com/JulienMrgrd/but4reuse>

(4) EFLBench : <https://github.com/but4reuse/but4reuse/wiki/Benchmarks>

(5) Thèse de Tewfik Ziadi : <http://www.irisa.fr/triskell/publis/2004/Ziadi04c.pdf>

(6) Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon : Feature Location Benchmark for Software Families Using Eclipse Community Releases, ICSR 2016, Limassol, Cyprus, 5-7 June