Table des matières

Introduction

Dans le cadre de ma deuxième année de master Informatique Pour les Sciences, j'ai eu l'opportunité de réaliser mon stage de fin d'étude au sein de l'entreprise CIL4Sys Engineering.

La majorité des entreprises de l'automobile utilisent des milliers de lignes de texte pour décrire, architecturer et concevoir le comportement d'un système.
CIL4Sys Engineering a défini un processus et créé sa propre chaîne d'outils pour accélérer l'innovation des produits et des services automobiles, ferroviaires et nucléaires.
Un point clé du processus et de la chaîne d'outils est qu'un ingénieur système peut voir le résultat de sa conception, grâce à la possibilité d'exécuter immédiatement des simulations par lui-même, sans attendre que d'autres construisent et exécutent des simulations ou des prototypes. Il délivre les documents d'exigences uniquement lorsque les simulations montrent que le comportement répond aux attentes.
Le processus repose sur une description de base faite en UML, ce qui rend très simple pour un ingénieur système de construire un modèle UML complet : il n'a qu'à concevoir des diagrammes de cas d'utilisation et des diagrammes de séquence.

Mon stage consiste à développer des fonctionnalités de tests automatisés appliqués pour un module additionnel de logiciel de modélisation Papyrus et établir une architecture de test fiable et robuste.
Ces tests permettront d'éviter les régressions et de repérer rapidement l'impact potentiel d'une nouvelle fonctionnalité sur l'ensemble du code par l'intermédiaire de rapports de tests.

Bien que les tests soient souvent négligés par les développeurs, ils sont un axe d'amélioration classique lorsqu'on souhaite renforcer la qualité d'une application.

Parmi ces tests, on trouve les tests unitaires qui ont un aspect pratique pour déboguer un morceau de code précis sans devoir exécuter l'application complète.
Durant ce stage, je vais tester plusieurs fonctionnalités en utilisant le Framework de test JUnit.

Dans ce rapport, je vais vous parler de CIL4Sys Engineering, sa spécialisation, son équipe ainsi que son organisation du travail pour la réalisation de ses projets. Ensuite, je vous parlerai de mes interventions au sein de CIL4Sys, mes formations, mes développements et les différents tests que j'ai réalisé et enfin de ce que j'ai apporté pour le projet sur lequel j'ai travaillé et ce que j'ai acquis comme expérience à titre personnel.

Partie 1

Présentation de CIL4Sys Engineering

Créée en juin 2015 par Philippe Gicquel, la société CIL4Sys Engineering, basée à Paris, s'est donné pour ambition d'accélérer l'innovation des produits et services dans l'automobile, le ferroviaire, le nucléaire par le bais d'une ingénierie agile des systèmes. Et ce en s'appuyant sur une chaîne d'outils innovante Sim4sys qui permet de concevoir visuellement, de vérifier par simulation et de spécifier un projet de systèmes embarqués complexes.

Concevoir une électronique embarquée, notamment dans le domaine automobile, devient une opération de plus en plus complexe. Sim4Sys est développée pour réduire l'utilisation des méthodes traditionnelles et des spécifications textuelles (les cahiers de charge, rédaction des exigences, ...) en automatisant les tâches de conception et de simulation et elle permet de visualiser en permanence les résultats.
Ce processus outillé offre la possibilité de vérifier des exigences par la simulation, avant qu'elles ne soient générées automatiquement. Pour les industriels utilisant un processus classique, l'écriture des éxigencesest souvent coûteuse en termes de temps de travail, et donc d'argent, sans être totalement fiables.

CIL4Sys se compose d'une équipe de développeurs et concepteurs qui possède souvent une double compétence, une équipe qui s'entraide afin d'atteindre les objectifs tracés pour chaque projet.

Partie 2

Projet Sim4Sys

Le projet sur lequel j'ai travaillé durant mon stage de fin d'étude se nomme Sim4sys, une chaine d'outils d'ingénierie système agile développée à partir de Papyrus (logiciel de modélisation UML libre dans l'environnement Eclipse).
Sim4Sys vise à faciliter la conception des systèmes embarqués et permet à CIL4Sys Engineering de livrer à ses clients en boucles courtes :

— Un modèle décrivant la logique comportementale du système étudié en UML,

— Les simulations permettant de visualiser le comportement de ce modèle,

— Les séquences de test pour la validation,

— La documentation générée à partir du modèle selon les besoins exprimés par chaque client.

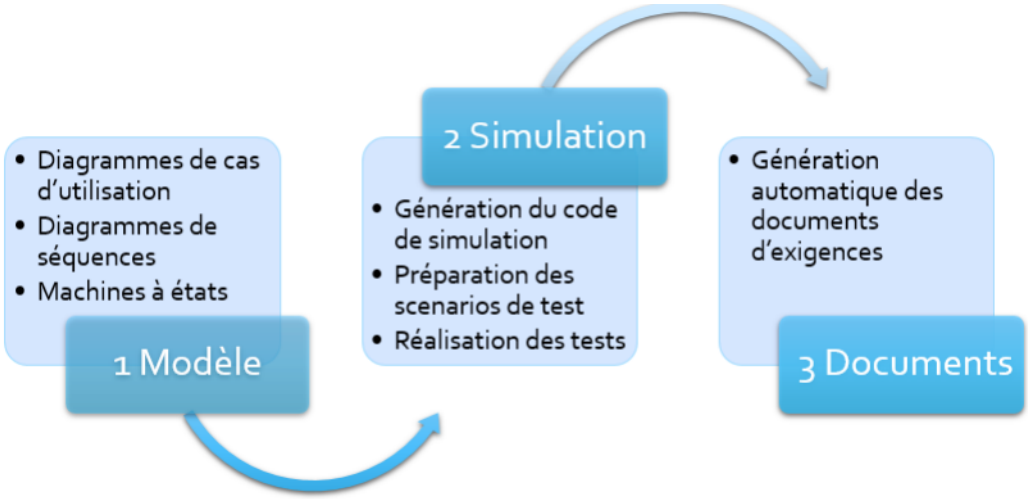


FIGURE 2.1 – Processus de Sim4Sys

2.1 Besoins

Mon stage consiste à développer les fonctionnalités des tests automatisés appliqués à un module additionnel de Papyrus et à établir une architecture de test fiable et robuste en utilisant l'environnement de test JUnit.

Ces tests permettront d'éviter les régressions et de repérer rapidement l'impact potentiel d'une nouvelle fonctionnalité sur l'ensemble du code par l'intermédiaire de rapport de tests ainsi que l'exécution automatique de tous les tests avant les nouvelles releases d'évolution du logiciel Sim4Sys.

2.2 Généralités sur les tests

Le test est une expérience d'exécution pour mettre en évidence un défaut ou une erreur basée sur le diagnostic (quel est le problème ?), le besoin d'un oracle, qui indique si le résultat de l'expérience est conforme aux intentions (où est la cause du problème ?). La figure 2.2 montre le processus d'un test de logiciel.

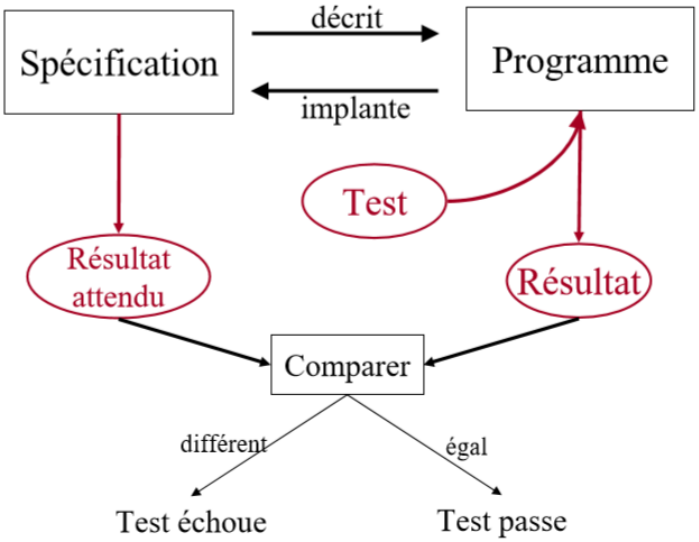


FIGURE 2.2 – Test de logiciel

— **Stratégies de test :**Il existe trois niveaux de test :
  — Le test unitaire,
  — Le test d'intégration est un test basé sur les cas d'utilisation, il intègre l'ensemble des classes nécessaires pour répondre à un cas d'utilisation et il démontre que chaque cas d'utilisation est exécutable par le système (avec ou sans l'interface utilisateur),
  — Le test système est celui qui permet de valider la globalité du système (les fonctions offertes à partir de l'interface).

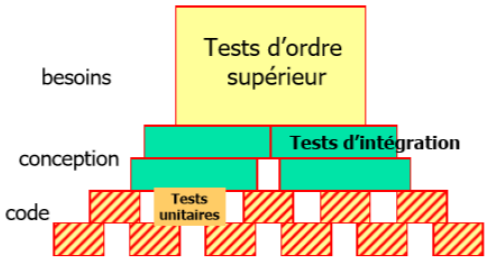


FIGURE 2.3 – Stratégies de test

Dans le cadre de mon stage, je n'ai manipulé que les tests unitaires.

2.3 Environnement technique

Dans le contexte de mes travaux, j'ai utilisé les outils suivants :

➢ **IDE Eclipse :** Environnement de développement intégré Java.

➢ **Langage UML :** Langage UML est la base de toute fonctionnalité à tester. En effet, Sim4Sys est une solution basée sur Papyrus et la modélisation UML.
La modélisation d'un logiciel s'aborde par 3 points de vue sur le système à modéliser :

— axe fonctionnel : ce que fait le système.

— axe structurel : ce que le système utilise pour fonctionner.

— axe comportemental : comment le système fonctionne.

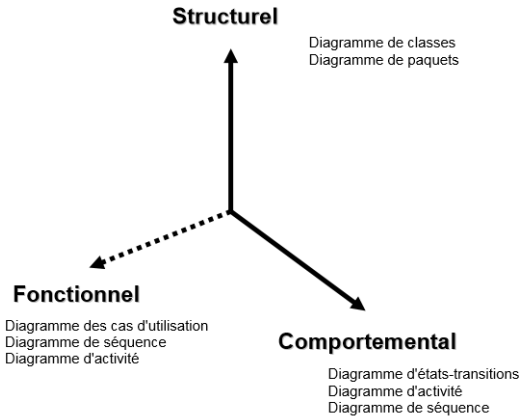


FIGURE 2.4 – Vue complémentaire sur la modélisation

Au cours de mes différents travaux, je n'ai manipulé que l'axe comportemental, en utilisant des diagrammes de séquence ainsi que des diagrammes d'états-transitions.

— **Diagramme de séquence :** Dans un contexte général, le digramme de séquence est un diagramme qui montre les interactions entre les utilisateurs et le système vu de l'extérieur de système. Dans un contexte Sim4Sys, le diagramme de séquence décrit une séquence d'interaction entre des acteurs et des modules internes du système dans un contexte d'exécution donné du système. Les lignes de vie représentent des instances qui peuvent être des utilisateurs, des éléments de l'environnement ou des sous-systèmes. Le diagramme de séquence décrit une activité en se focalisant sur l'enchaînement des messages qui représentent des flux entre les différentes lignes de vie pour réaliser un scénario correspondant à cette activité.

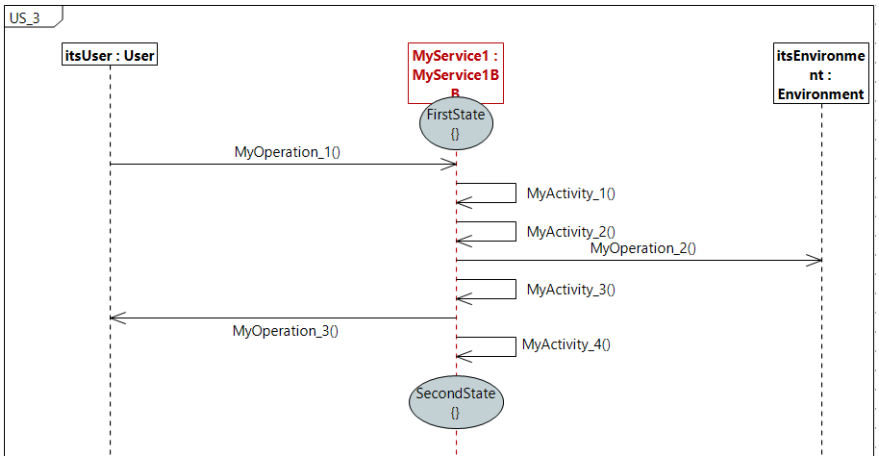


FIGURE 2.5 – Exemple d'un diagramme de séquence

— **Diagramme d'états-transitions :** Il décrit le comportement d'un système automatisé à la réception d'un signal. Il permet de rassembler les états et les transitions dans un système.

  — L'Etat : Représente une période de la vie d'un système, pendant cette période, le système va accomplir des tâches(actions) ou attendre un évènement. Il est représenté par un rectangle à bornes arrondies avec en haut le nom de l'état et à l'intérieur les commandes qui permettent de piloter cet état.

  — La transition : On les trouve entre les états, la transition est un événement (appuyer sur un bouton par exemple) qui va permettre de changer d'état, la transition peut être interne donc l'état de son départ et de son arrivée est le même, ou externe c'est à dire elle lie deux états différents.

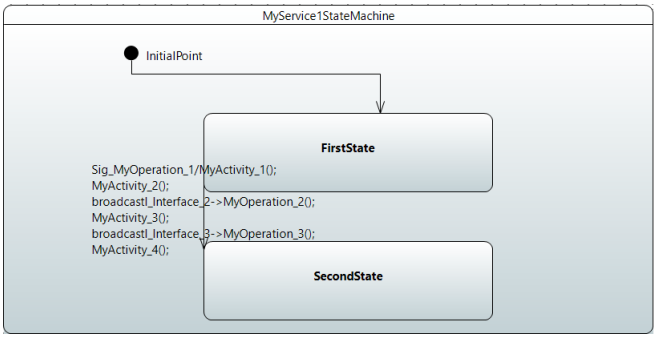


FIGURE 2.6 – Exemple d'un diagramme d'état transition

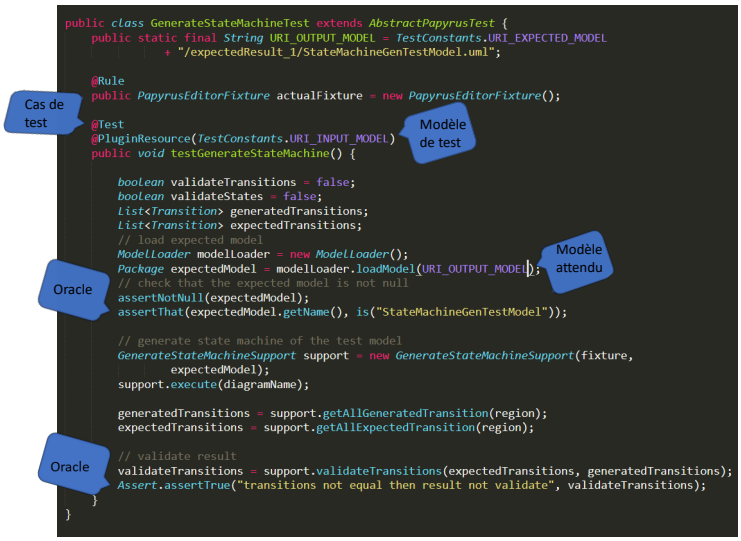
➢ **Papyrus :** Sim4Sys est développé à partir de Papyrus, un outil développé initialement au sein du CEA-List en France et issu du projet PolarSys de la fondation Eclipse. Il s'agit d'un environnement de modélisation de haut niveau qui supporte les métalangages UML 2.5 et SysML 1.4, et autorise la création de langages de modélisation spécifiques à une industrie ou à un domaine applicatif.

➢ **EMF Compare :** Eclipse Modeling Framework est un outil qui permet de comparer des modèles au cadre EMF afin de relever les différences existantes entre eux. Cet outil fournit un support générique pour tout type de métamodèle.
Durant ce stage, j'ai utilisé EMF Compare dans le but de tester le renommage ou l'ajout d'un élément dans un modèle de test et de relever les différences entre ce même modèle avant et après modification.
Dans ce but, j'ai intégré l'API papyrus compare qui permet de comparer des modèles plus spécialisés (UML) car des différences peuvent être présentes en UML et absentes en EMF. EMF compare est utilisé pour d'autres logiciels de modélisation tel que Capella.

➢ **Sourcetree :** Avant de parler de sourcetree, je vais vous décrire briévement Git. Git est un outil qui permet de gérer tout l'historique des modifications apportées à un projet au fur et à mesure de son avancement. A tout moment, on va pouvoir remonter l'ensemble des modifications au projet. Git est un outil pratique pour travailler à plusieurs sur le même projet.
Sourcetree est un logiciel gratuit développé par Atlassian qui possède une interface graphique simple et complète qui offre un processus de développement efficace et cohérent. Il simplifie le contrôle de version distribué avec un client Git et permet d'informer efficacement les utilisateurs.
Dans ce stage, j'ai utilisé SourceTree pour gérer mes développements et le versionnement avec Git.

➢ **JUnit :** Est une librairie Java qui permet de réaliser des tests unitaires. Cette dernière permet de faire des tests simples et efficaces en utilisant des annotations et des assertions.

Les tests unitaires permettent de valider un module indépendamment des autres et de vérifier des fonctionnalités unitaires d'un logiciel.
Dans un contexte orienté objet :
- unité de test = classe
- Cas de test = méthode




```
public class GenerateStateMachineTest extends AbstractPapyrusTest {
    public static final String URI_OUTPUT_MODEL = TestConstants.URI_EXPECTED_MODEL
            + "/expectedResult_1/StateMachineGenTestModel.uml";

    @Rule
    public PapyrusEditorFixture actualFixture = new PapyrusEditorFixture();

    @Test
    @PluginResource(TestConstants.URI_INPUT_MODEL)
    public void testGenerateStateMachine() {

        boolean validateTransitions = false;
        boolean validateStates = false;
        List<Transition> generatedTransitions;
        List<Transition> expectedTransitions;
        // load expected model
        ModelLoader modelLoader = new ModelLoader();
        Package expectedModel = modelLoader.loadModel(URI_OUTPUT_MODEL);
        // check that the expected model is not null
        assertNotNull(expectedModel);
        assertThat(expectedModel.getName(), is("StateMachineGenTestModel"));

        // generate state machine of the test model
        GenerateStateMachineSupport support = new GenerateStateMachineSupport(fixture,
                expectedModel);
        support.execute(diagramName);

        generatedTransitions = support.getAllGeneratedTransition(region);
        expectedTransitions = support.getAllExpectedTransition(region);

        // validate result
        validateTransitions = support.validateTransitions(expectedTransitions, generatedTransitions);
        Assert.assertTrue("transitions not equal then result not validate", validateTransitions);
    }
}
```

FIGURE 2.7 – Cas de test JUnit

Un cas de test JUnit est un ensemble composé de trois objets comme indiqué dans la figure 2.7 :

— Un état (ou contexte) de départ ;
— Un état (ou contexte) d'arrivée ;
— Un oracle, c'est à dire un outil qui va prédire l'état d'arrivée en fonction de l'état de départ et comparer l'état de départ au résultat attendu. Pour se faire, il faut appeler une des nombreuses variantes de méthodes assertXXX() fournies, par exemple :
  — assertTrue(String message, booleantest),
  — assertFalse(...),
  — assertEquals(...) : test d'égalité avec equals,
  — assertNotSame(...), assertSame(...) : tests d'égalité de référence,
  — assertNull(...),
  — assertNotNull(...).

  Il existe aussi Hamcrest qui est un Framework utilisé pour les test unitaires JUnit (toutes les versions) et TestNG, son objectif est de rendre les tests plus lisibles, ainsi que des messages d'erreur plus faciles à lire.
  Il permet de vérifier les conditions dans le code en faisant appel à des classes matchers existantes ou en créant des implémentations personnalisées.

— **Conception des cas de test :** On ne peut pas tester tout le temps ni tous les cas possibles, donc dans le domaine de test, le choix des cas intéressants et la bonne échelle pour le test est une étape stratégique.

Pour bien concevoir des cas de test utiles :

— Chaque cas de test doit être identifié de manière unique et explicitement associé à la classe à tester,

— L'objectif du test doit être décrit,

— Une liste des étapes du test doit être définie pour chaque test et doit contenir : La liste de tous les états spécifiés pour l'objet à tester, la liste des messages/opérations qui seront utilisés par ce test, la liste des exceptions qui peuvent être levées lors du test de l'objet, les informations complémentaires qui pourront faciliter la compréhension ou l'implémentation du test.

Partie 3

Gestion du projet

Afin de comprendre l'organisation suivie par l'équipe CIL4Sys pour gérer ses projets et planifier les étapes à suivre pour satisfaire leurs clients, je vais vous expliquer dans cette partie de mon rapport, la composition détaillée de l'équipe, la méthode agile adaptée ainsi que ma propre organisation de travail pour atteindre l'objectif de mon stage.

3.1 L'équipe

Voici la composition de l'équipe du projet Sim4Sys :

— Le projet est géré par le fondateur de la start-up qui supervise l'avancement, tout en étant en contact avec son équipe et les clients,

— Une équipe technique composée de neuf personnes : un responsable technique modélisation, des ingénieurs plutôt orienté développement des outils et des ingénieurs orientés conception, modélisation, simulation,

— Une directrice commerciale qui assure les relations avec les différents clients. CIL4Sys n'opte pas pour le portage c'est à dire que même si elle a un contrat avec un client son équipe travaille dans ses locaux, ce qui permet de mieux s'entraider

3.2 Les méthodes Agile

Les méthodes Agiles sont des pratiques de réalisation de projets, basées sur des développements itératifs telles que Exterme Programming(XP) et SCRUM.

La méthode SCRUM :
C'est un cadre méthodologique agile qui provient du monde informatique m'étant à se généraliser y compris dans le marketing. Scrum a été appliqué et adapté par CIL4Sys pour ses besoins. Cette méthode permet de délivrer et modifier un projet très rapidement. La méthode Scrum est basé sur des rôles. On trouve l'équipe (développeurs, testeurs et tout métiers nécessaires à la réalisation du projet) et le chef de produit (Product Owner), c'est le client qui va définir les spécifications fonctionnelles de ce qu'il faut développer ainsi que la validation des fonctionnalités.

Le processus d'application de Scrum est basé sur la définition de :

— User Story : il s'agit de décrire toutes les fonctionnalités nécessaires à la réalisation de produit, l'importance de chaque fonctionnalité, l'estimation du travail nécessaire ainsi qu'une démonstration simple de la fonctionnalité.

— Product BackLog : à partir des user story, il va émaner des exigences, elles seront hiérarchiser avec le client par rapport à leurs importances. Le Product BackLog est comme un miroir de ce qu'il faut faire pour réaliser le besoin du client.

— Sprints : Une fois que le client est d'accord sur les exigences, il est temps de se lancer dans la réalisation du projet. Le BackLog sera découpé en plusieurs parties qui seront développées dans les sprints. Chaque sprint dure de 2 à 4 semaines, y compris un développement, un test et une livraison. CIL4Sys travaille en boucle courte de 2 semaines.

— Réunions Scrum : Des réunions organisées chaque jours permettent à l'équipe de mesurer l'avancement du projet et de s'assurer de la qualité des livrables ainsi que le respect des délais. A la fin des deux semaines, une solution est présentée au client sous forme d'une démonstration pouravoir son retour. Ses retours sont analysés lors des rétrospectives que CIL4Sys organise au début de chaque semaine et des changements peuvent être intégrés au BackLog. Un schéma explicatif de la méthode Scrum est donnée sur la Figure 3.1.

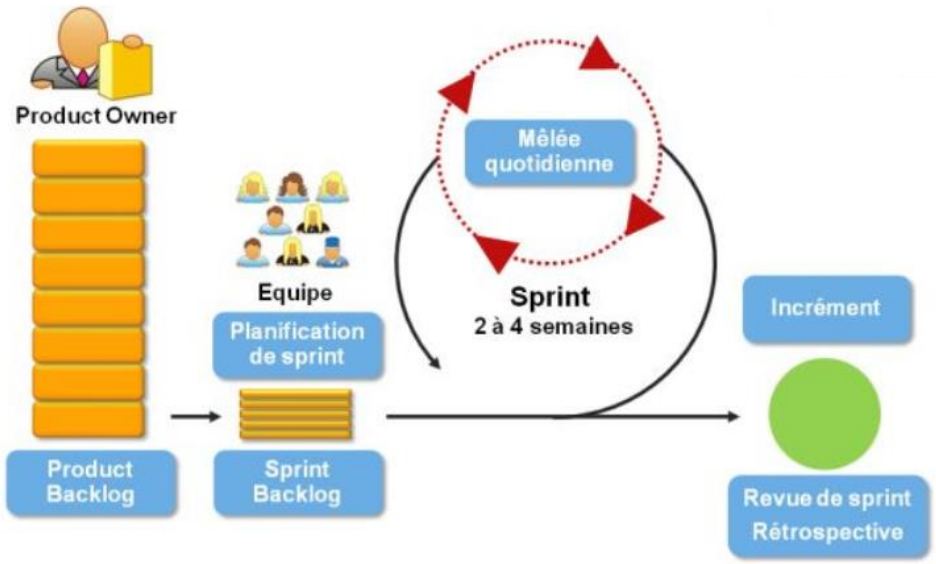


FIGURE 3.1 – La méthode SCRUM

3.3 L'organisation de mon travail

Tout au long de mon stage, j'étais guidé par mon tuteur pour le choix des API et des fonctionnalités les plus importantes à tester mais j'ai aussi fait un travail plus exploratoire. En effet, j'étais libre d'explorer d'autres API qui permettent d'atteindre l'objectif de test. Tout au long de ce projet, j'étais guidée par mon encadrant. Des réunions régulières avec F. Yoann ont eu lieu afin d'assurer le suivi de mon avancement. Lors de la période de confinement suite à l'épidémie de Covid-19, CIL4Sys à opter pour le télétravail. Ceci n'a pas impacter l'organisation du travail initialement définie.
En effet, des réunions étaient organisées chaque jour, pour discuter de l'avancement de chacun sur la tâche qui lui a été confiée. De plus, des revues permettent le partage et la présentation des résultats obtenus y compris mes résultats de test des explications concernant les Frameworks utilisés ainsi que les résultats obtenus.

3.4 Diagramme de GANTT

Comme je l'ai expliqué dans la section précédente, Mon travail est un travail exploratoire, je n'étais pas liée à des créneaux précis pour la réalisation de mes tâches. Dans cette section, je résume le calendrier de mes tâches à posteriori dans un diagramme de gantt.(Figure 3.2)

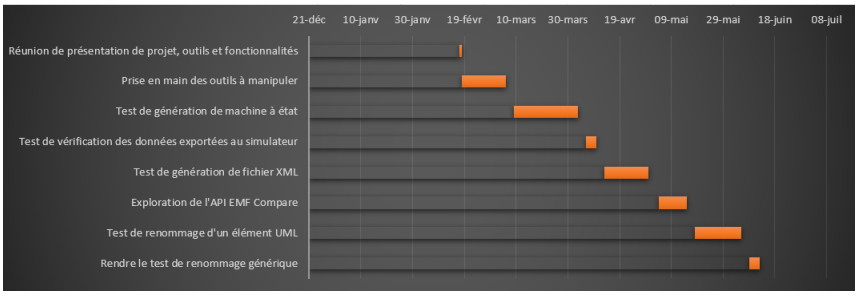


FIGURE 3.2 – Diagramme de Gantt

Partie 4

Les tests JUnit

La mission principale de mon stage consiste à réaliser des plans de tests afin de vérifier le bon fonctionnement de certaines fonctionnalités principales dans le projet Sim4Sys. Ce travail a été réalisé en utilisant des outils de développement Java classiques et des Frameworks spécifiques pour les tests dans un environnement Eclipse. Dans cette partie de mon rapport, je vais vous parler de toutes les étapes que j'ai suivi dès le début de mon stage pour réaliser le travail demandé.

4.1 Découverte du projet

Les premiers jours de mon stage étaient consacrés à la réalisation de petites formations sur le fonctionnement de la chaine outillée Sim4Sys ainsi que sur les différents Framework à utiliser pour atteindre l'objectif de mon stage. Lors de ces premiers jours, j'ai réalisé : :

- Une formation approfondie sur les éléments UML manipulés,
- Une formation sur l'environnement Eclipse/Maven avec intégration de test,
- Une formation sur JUnit 5,
- Des investigations sur les plugins de test de Papyrus afin de me familiariser avec les tests portant sur des éléments UML.

4.2 Les développements

4.2.1 Test de génération de la machine à état à partir d'un diagramme de séquence

Dans un premier temps, il me semble pertinent d'expliquer la fonctionnalité testée dans un modèle d'exemple portant sur la conception d'un lave-linge.
Dans la figure 4.1, je présente un diagramme qui décrit les séquences entre l'utilisateur de l'appareil et le système. Le premier message "the user requests to select a program" est le déclencheur de la première transition lors de la génération de la machine à état. Dans le contexte Sim4Sys, il peut y avoir des conditions à vérifier avant de déclencher le comportement du système.

Le diagramme de séquence représenté dans la figure 4.1 contient :

— Trois lignes de vie : deux correspondent à des acteurs pouvant interagir avec le système, la dernière correspond à un service dont on souhaite spécifier le comportement.

— La représentation d'un état en haut de la ligne de vie du service « WashingMachineControlBB ».

— Un message entre la ligne de vie « User » et la ligne de vie du service « WashingMachineControlBB » . Un second message entre la ligne de vie service et la ligne de vie « Environment ».

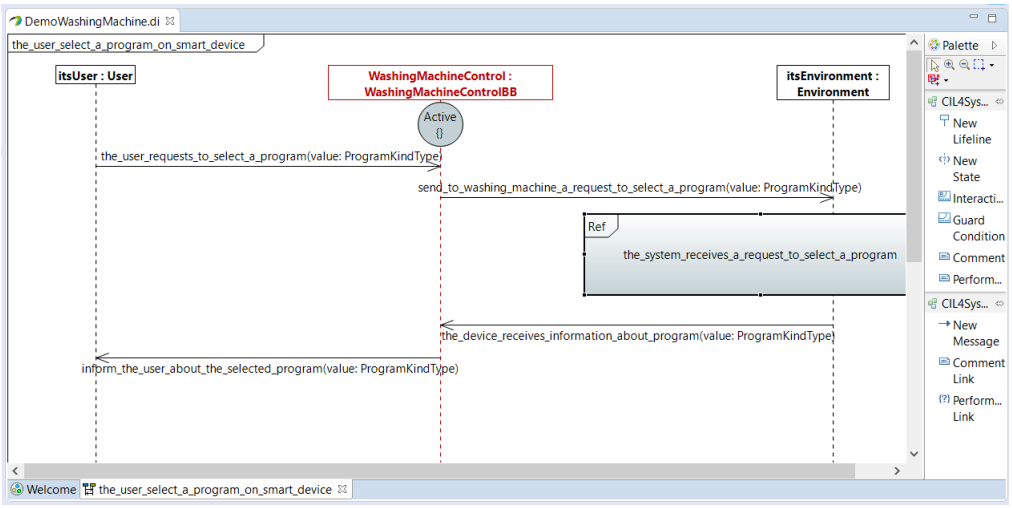


FIGURE 4.1 – Diagramme de séquence de l'appareil intelligent

Le choix de la fonctionnalité qui permet de générer la machine à état est fait dans une vue (Overview) de Sim4Sys comme indiqué sur la Figure 4.2.

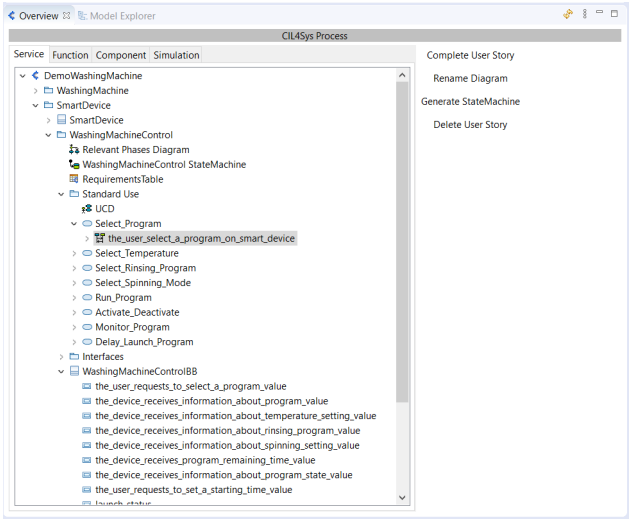


FIGURE 4.2 – Fonctionnalité de la génération de la machine à état

La transcription d'un tel diagramme en transition dans le diagramme de machine à état se fera en suivant les règles suivantes :

— Le premier état placé sur la ligne de vie sera l'état initial.

— Un message reçu par une ligne de vie sera interprété comme un évènement déclencheur.

— Les messages sortant d'une ligne de vie en dessous d'un message reçu seront les flux à envoyer (vers les acteurs ou d'autres services du produit par exemple).

— Si un message a ses deux extrémités sur la même ligne de vie, il s'agira d'une activité interne, d'une action à réaliser.

— Si un état est placé sur la ligne de vie, cela indique que la transition générée occasionnera un changement d'état, et que cet état sera l'état final.

— Une condition pourra être apposée à un message déclencheur de manière à contraindre le déclenchement de la transition.

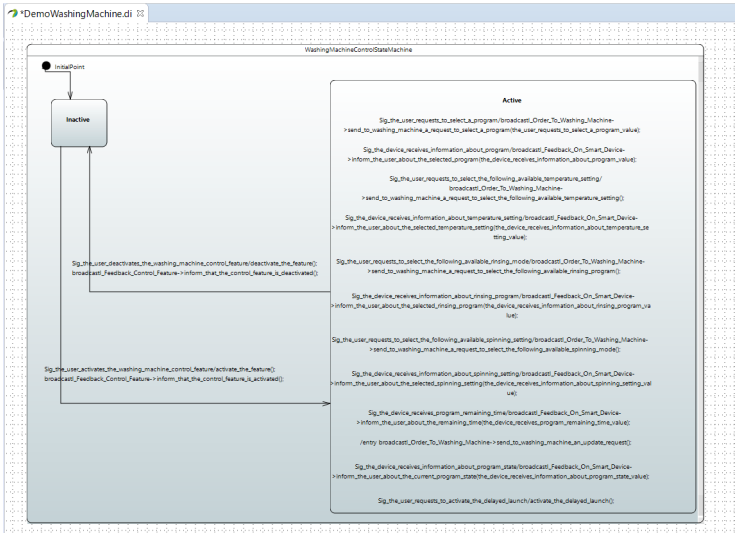


FIGURE 4.3 – Diagramme d'état transition

Les opérations déclenchées lors de la génération de diagramme de machine à état peuvent être (comme indiqué sur la Figure 4.4) :

— Des opérations instantanées non interrompues (dites actions) associées à un état ou une transition, ces opérations peuvent intervenir soit en entrée de l'état avec le préfix "entry/", ou en sortie de l'état avec le préfixe "exit/". Pour Sim4Sys, les entry sont les seules opérations utilisées.

— Des opérations d'une certaine durée qui sont exécutées tant que l'objet se trouve dans l'état (dites activités), elles sont représentées dans l'état précédées par la notation "do/".

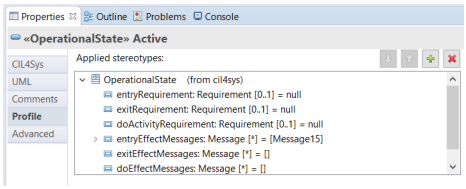


FIGURE 4.4 – Les opérations déclenchées dans un état

Pour ce test, j'ai vérifié le bon fonctionnement de la fonctionnalité ainsi que les différentes conditions que cette fonctionnalité doit respecter. Dans ce test, j'utilise un modèle de départ (test model) que je vais indiquer dans le contexte de test JUnit avec l'annotation "@PluginRessource(chemin vers le modèle de test)". Un environnement logiciel pour exécuter un test peut être initialisé en utilisant l'annotation @Rule, ceci permet d'indiquer la « fixture » utilisée tout au long de ce test. Un exemple de fixture utilisé dans l'un de mes tests est présenté dans le Listing 4.1.

```
1
2  import org.eclipse.papyrus.junit.utils.rules.PapyrusEditorFixture;
3  import org.eclipse.papyrus.junit.utils.rules.PluginResource;
4  import org.junit.Rule;
5  import org.junit.Test;
6
7  /**
8   * generate state machine of test model "StateMachineGenTestModel"
9   * from a sequence diagram "US_1" with output operation and one
10  * state stereotyped Operational state, with trigger and
11  * without Guard condition
12  */
13 public class GenerateStateMachineTest1 extends AbstractPapyrusTest {
14     public static final String OUTPUT_MODEL = TestConstants.
       URI_EXPECTED_MODEL
15         + "/expectedResult_1/StateMachineGenTestModel.uml";
16     @Rule
17     public PapyrusEditorFixture actualFixture = new PapyrusEditorFixture();
18     @Test
19     @PluginResource(TestConstants.URI_INPUT_MODEL)
20     public void testGenerateStateMachine() {
21         StateMachineGenerationTestUtils.launchTest(actualFixture,
22             OUTPUT_MODEL, TestConstants.DIAGRAM_NAME_1,
23                 TestConstants.REGION);
24     }
25 }
```

Listing 4.1 – Exemple de fixture

Un autre modèle contenant le résultat attendu est chargé via une méthode utilitaire "loadModel". Ce modèle comprend tous les changements attendus par la fonctionnalité à tester. Dans ce test, je compare les éléments UML (transitions et états) pour chaque modèle, dans un premier temps, je génère le diagramme d'état transition de modèle de test, je récupère les transitions et les états, ensuite j'utilise des oracles pour affirmer les différentes comparaisons entre les différents objets (un exemple montrant la comparaison de propriété "effect" des deux transitions est présenté dans le Listing 4.2).
Parmi les cas de test réalisés :

— Comparer les messages des déclencheurs d'états.

— S'il existe des conditions, je vérifie le corps « body » de la condition (il faut avoir le langage C++ par exemple).

— Comparer l'état de départ et d'arrivés (source et target) d'une transition générée.

— Vérifier qu'un stéréotype (une spécialisation appliquée à un élément UML) est bien appliquée ainsi qu'il est de type "RequirementTransition", type spécifique à des éléments UML dans le contexte Sim4Sys.

— Vérifier que ces stéréotypes stockent des messages qui produisent des entrée (entry) qui sont ainsi stockés dans les états sous forme d'un Opaque Behavior (élément UML qui permet de stocker du texte).

```
1  protected boolean compareEffect(Behavior firstEffect, Behavior
       secondEffect) {
2          boolean checkEffect = false;
3          boolean compareEffect = false;
4
5          // check that the effects are opaque behavior
6          checkEffect = firstEffect instanceof OpaqueBehavior &&
       secondEffect instanceof OpaqueBehavior;
7
8          // areSameTransition return true if and only if the both
       transitions are //equal
9          if ((firstEffect == null && secondEffect != null) || (
       firstEffect != null && secondEffect == null)) {
10             compareEffect = false;
11         } else if (firstEffect == null && secondEffect == null) {
12             compareEffect = true;
13         } else if (firstEffect != null && secondEffect != null) {
14             if (checkEffect) {
15                 OpaqueBehavior myFirstEffect = (OpaqueBehavior)
       firstEffect;
16                 OpaqueBehavior mySecondEffect = (OpaqueBehavior)
       secondEffect;
17                 compareEffect = equalBody(myFirstEffect,
       mySecondEffect);
18             }
19         }
20         return compareEffect;
21     }
```

Listing 4.2 – Exemple de cas de comparaison des transitions

Après vérification de tous les cas de test, j'utilise une méthode qui permet de valider le test de la génération des transitions (Listing 4.3).

```
1  public boolean validateTransitions(List<Transition> expectedTransition,
       List<Transition> generatedTransition) {
2          boolean equal = true;
3          Map<Transition, Transition> map = new HashMap<>();
4
5          // Check that the list of expected transitions and generated
       transitions //have the same size
6          boolean haveSameSize = (expectedTransition.size() ==
       generatedTransition.size());
7          Assert.assertTrue("the list of transitions have not the same size",
        haveSameSize);
8
9          // affect all expected transitions to key map
10         for (Transition transition : expectedTransition) {
11             map.put(transition, null);
12             assertNotNull(map);
13         }
14
15         // check that the map and the list of transition have same size
16         assertEquals(map.size(), expectedTransition.size());
17
18         // check if compare is true, if yes affect the generated transition
        to the //value map.
19         for (Entry<Transition, Transition> transitionMap : map.entrySet())
       {
20             for (Transition transition : generatedTransition) {
21                 if (compareTransition(transitionMap.getKey(), transition))
       {
22                     map.put(transitionMap.getKey(), transition);
23                 }
24             }
25         }
26         // check that each key(expected transition) has an associated //
       value(generated transition)
27         for (Entry<Transition, Transition> transitionMap : map.entrySet())
       {
28             if (transitionMap.getValue() == null) {
29                 equal = false;
30                 assertFalse("there is a transition that is not generated",
       equal);
31             }
32         }
33         return equal;
34     }
```

Listing 4.3 – Validation de test

J'ai fait ainsi plusieurs cas de test pour valider le test de comparaison des états.
La seule difficulté que j'ai rencontré dans ce premier test était de trouver une méthode JUnit qui permet de récupérer deux modèles : de test et attendu dans le même test. Après plusieurs recherches, j'ai créé une méthode qui charge le modèle attendu car JUnit ne peut pas charger deux modèles en même temps.

4.2.2 Test de génération de fichier XML

La seconde fonctionnalité que j'ai testée est celle qui permet d'exporter un ensemble de données dans un fichier XML. Ce fichier XML contient les données nécessaires aux échanges d'information entre le simulateur et le modèle : la description des types de données et la description des flux.

— Le modèle : Est le projet papyrus qui contient l'ensemble des diagrammes de cas d'utilisation, diagrammes de séquence et les diagrammes d'état transition qui expliquent la conception détaillée d'un système donné. (Figure 4.5).

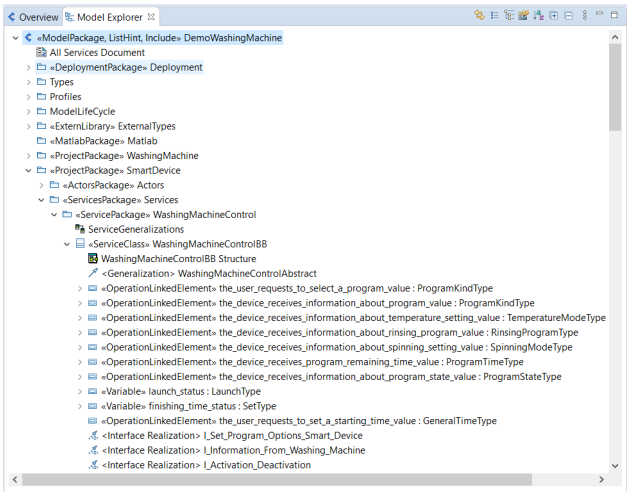


FIGURE 4.5 – Exemple d'un modèle papyrus

— Le simulateur : Est l'un des outils de la solution Sim4sys, il permet la préparation des scénarios de test à partir des données importées de modèle ainsi que la réalisation de ces tests. Il donne une vision plus claire et réelle d'un système complexe, l'outil est disponible sur le site sim4Sys.com. Un exemple de simulation d'un lave-linge est donnée sur la Figure 4.6.

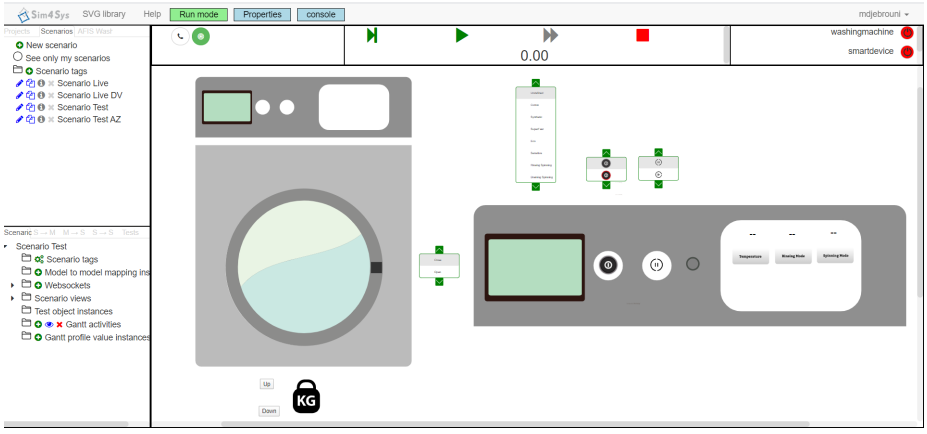


FIGURE 4.6 – Exemple d'une simulation

Les données exportées dans le fichier XML doivent être correctes et conformes aux données attendues par un outil de sim4sys. Le fichier est créé en portant le nom de projet avec l'extension ".xml" qui sera stocker dans un dossier "export". Ce dossier est ajouté à l'ensemble des fichiers de modèle papyrus.
Dans un premier temps, j'ai fait quelques cas de test qui permettent de vérifier la conformité des données exportées entre un modèle de test et un ensemble de données construites et qui sont les données attendues.
Parmi les cas de test réalisés :

- Vérifier que les directions des données pour chaque opération de modèle stéréotypée "ConversionToModelOperation", sont de simulateur au modèle : "simulateur -> modèle". Les opérations de modèle stéréotypée "FlowOperation", doivent avoir la direction "modèle -> simulateur".
- Vérifier que le type des données exportées est primitiveType, DataType ou Enumeration.

Par la suite, j'ai généré le fichier XML de modèle de test et j'ai récupérer ce fichier et le fichier XML qui contient les données attendues construites. Ensuite, j'ai comparé les deux fichiers XML en utilisant deux alternatives :

- En comparant toutes les chaines de caractère ligne par ligne, Dans un premier temps cette méthode n'a pas permis de valider le test. Après investigation, j'ai pu mettre en évidence que les données contenues dans le fichier étaient correctes mais l'ordre des données générées différait d'une génération à l'autre. Le test échouait alors à comparer ligne à ligne deux fichiers qui étaient donc différents. Il s'agit d'un cas où l'écriture de test a permis de mettre en lumière un défaut dans l'implémentation, qui a ensuite été corrigé.

- En utilisant la librairie XMLUnit qui permet de tester et de vérifier le contenu XML. Elle est utile lorsque nous savons exactement ce que doit contenir le fichier XML. Elle est indépendante de JUnit.
  La classe Diff de XMLUnit permet d'identifier les différences entre deux documents. XMLUnit est capable de comprendre toutes les balises d'élément dans le contexte XML, par exemple (une balise d'élément vide et un élément avec une balise de début et de fin sans aucun contenu ont la même représentation).
  Grace aux méthodes similar() et identical() de la classe Diff, nous pouvons comparer deux fichiers XML facilement. En effet, Ils sont réputés identiques lorsque le contenu et la séquence des nœuds dans les documents sont exactement identiques et ils sont similaires mais non identiques si leurs nœuds apparaissent dans une séquence différente.(Listing 4.4)

```
1 public void assertXMLEqual() throws Exception{
2     getFolderAndFileExist();
3     Diff diff = new Diff(expectedFileContent, generatedFileContent);
4     Assert.assertTrue("the xml files are not similar" + diff.toString
      (), diff.similar());
5     Assert.assertTrue("the xml files are not identical" + diff.
      toString(), diff.identical());
6 }
```

Listing 4.4 – Cas de test XMLUnit

4.2.3 Test de renommage, modification des données de modèle

le projet Eclipse Modeling Framework (EMF) est un Framework de modélisation qui permet la génération de code Java à partir des modèles de données structurées.
EMF Compare que j'ai utilisé comme API pour réaliser ce test est un sous projet d'EMF destiné à la comparaison de modèles réalisés avec EMF. Il fournit des options de personnalisation pour prendre en compte des languages de modélisation spécifique tel que UML que j'ai manipulé au cours de ce test.
Le composant principale d'EMF Compare est "Comparison" qui est un composant de comparaison qui prend en charge les comparaisons bidirectionnels (deux modèles) ou tripartites (trois modèles).

La comparaison des modèles est réalisé en quatre phases :

— Phase de résolution : Après avoir défini dans le scope les deux modèles à comparer, EMF Compare va faire une résolution de modèle c'est à dire qu'il va trouver tous les éléments de modèle nécessaires à la comparaison et donc obtenir deux modèles logiques qu'il va par la suite comparer.

— La phase de correspondance : Ensuite, il va itérer sur les deux modèles logiques afin de mapper les éléments deux par deux dans une comparaison de deux modèles ou trois à trois dans une comparaison de trois modèles. Il va déterminer quels éléments correspondaient ensemble et il va les mettre dans un modèle de comparaison.

— La phase de différenciation : Ici il va parcourir les mapping trouvés à la phase précédente et il va déterminer si les deux éléments sont égaux ou s'ils présentent des differences. Pour chaque différence relevée, un élément dit "diff" est créé qui décrit avec précision chaque variation entre les éléments. Cet élément est par la suite stocké dans le modèle de comparaison à côté des correspondances.

— Phase d'analyse : dans cette phase, EMF Compare va classer les différences relevées en passant par trois étapes :

  — Equivalence : Deux differences distinctes pourraient présenter le même changement, cette phase parcourra toutes les differences et les reliera lorsqu'elles peuvent être considérées comme équivalente (par exemple, les differences sur les références opposées).

  — Exigence : Cette phase a pour but de fusionner des differences, par exemple l'ajout d'une classe C1 dans un package P1, cet ajout dépend de l'ajout du package lui-même donc cette différence est fusionnée en une seule.

  — Conflits : cette étape existe dans la comparaison de trois versions de modèle. Les conflits seront entre les modifications apportées aux modèles localement et celles apportées aux modèles sur le référentiel distant.

Le processus de comparaison d'EMF Compare est résumé dans la Figure (Figure4.7).

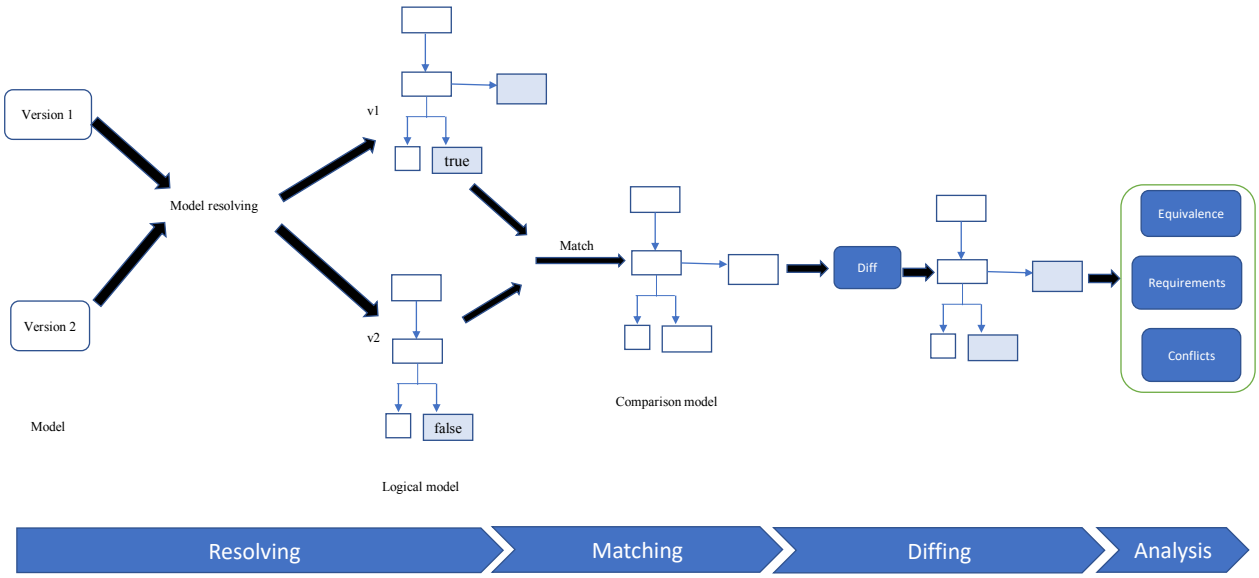


FIGURE 4.7 – Processus de comparaison EMF Compare

Dans ce test, j'ai vérifié le bon fonctionnement de la fonctionnalité de renommage d'un élément UML et pu détecté tous les changements qui suivent cette modification.
J'ai renommé un état de départ d'une machine à état. En conséquence, les noms de tous les états présents dans le diagramme de séquence sont mis à jour.
J'utilise l'API EMF Compare pour relever les différences entre le modèle de test avant et après modification. L'initialisation de la classe "PostProcessor" pour créer une nouvelle extension est nécessaire pour gérer des modèles UML correctement car lors de la comparaison de modèles UML, un filtre supplémentaire est disponible, je l'indique avec l'expression régulière comme indiqué dans le code suivant (Listing 4.5) :

```
1
2  // initialization of the PostProcessor uml is necessary to manage uml
       correctly
3  // because When comparing UML models, an additional filter is available,
       this is
4  //indicated with the regular expression
5          final IPostProcessor.Descriptor.Registry<String>
       postProcessorRegistry = new PostProcessorDescriptorRegistryImpl<String
       >();
6          BasicPostProcessorDescriptorImpl post = new
       BasicPostProcessorDescriptorImpl(new UMLPostProcessor(),
7                  Pattern.compile("http://www.eclipse.org/uml2/\\d\\.\\d\\.\\
       d/uml"), null);
8          postProcessorRegistry.put(UMLPostProcessor.class.getName(), post);
9          builder.setPostProcessorRegistry(postProcessorRegistry);
```

Listing 4.5 – Cas de comparaison EMF Compare

Après avoir indiqué l'extension nécessaire de nos modèles, je récupère les différences relevées à l'aide des méthodes de la classe "Comparison". Ensuite, je compare ses différences à l'aide de l'interface "IEqualityHelper" qui contient la méthode "matchingValues(candidate, expectedElement)" qui compare deux objets comme indiqué dans le listing 4.6 suivant :

```
1
2  public boolean validateComparison(List<Object> listOfExpectedElement, List<
       Object> listOfDiff) {
3          boolean equal = true;
4          Map<Object, Object> map = new HashMap<>();
5          final IEqualityHelper equality = comparison.getEqualityHelper();
6          // Check that the list of expected element and list
7          // of differences have the same size
8          boolean haveSameSize = (listOfExpectedElement.size() == listOfDiff.
       size());
9          Assert.assertTrue("the list of expected element have not the same
       size", haveSameSize);
10         // affect all expected transitions to key map
11         for (Object element : listOfExpectedElement) {
12             map.put(element, null);
13             assertNotNull(map);
14         }
15         // check that the map and the list of object have same size
16         assertEquals(map.size(), listOfExpectedElement.size());
17
18         // check if matchingValues is true, if yes affect the
19         //difference to the element map.
20         for (Entry<Object, Object> diffMap : map.entrySet()) {
21             for (Object diff : listOfDiff) {
22                 if (equality.matchingValues(diffMap.getKey(), diff)) {
23                     map.put(diffMap.getKey(), diff);
24                 }
25             }
26         }
```

```
27          // check that each key(expected element) has an associated value(
        diff)
28          for (Entry<Object, Object> mapElements : map.entrySet()) {
29              if (mapElements.getValue() == null) {
30                  equal = false;
31              }
32          }
33          return equal;
34      }
35  }
```

Listing 4.6 – Cas de comparaison EMF Compare

4.3 Travail futur

Au cours du développement, il faut prendre le temps de réaliser les tests correspondants à chaque nouvelle fonctionnalité ajoutée à une application. Mais ce temps se récupère au moment du test et du débogage. Il suffit de lancer les tests unitaires pour savoir rapidement où les bugs se situent, ce qui permet de les corriger sans perdre de temps.
L'intégration continue est un ensemble de pratiques utilisées en génie logiciel qui consiste à vérifier à chaque modification de code que le résultat des modifications ne produit pas de régression dans l'application développée.
Dans la suite de mon stage, je vais m'intéresser à un outil d'intégration continue simple mais puissant qui est le serveur open source Jenkins qui offre de nombreuses possibilités pour rendre le processus d'intégration continue plus efficace par le biais de plugins.

Conclusion et perspectives

Bilan

Ce stage était une expérience très enrichissante qui m'a donné la chance d'utiliser des technologies récentes dans un projet de grande envergure et de découvrir la méthode agile de gestion de projet.
Découvrir différentes API (JUnit, Hamcrest, XMLUnit, EMF Compare) utilisés dans le domaine des tests était un avantage pour moi, cela m'a permis d'apprendre plusieurs approches et alternatives pour réaliser des tests. J'ai appris que c'est important d'utiliser des API puissantes mais on doit toujours les adapter à notre contexte d'utilisation.
Le but des tests unitaires est de tendre à l'automatisation des tests lors du déploiement afin de pouvoir faire des modifications au code tout en ayant un control continu sur sa qualité et ainsi éviter l'introduction de bugs dans certains cas particuliers. Mon travail futur qui consiste à la découverte de logiciel d'intégration continue Jenkins sera une opportunité et un plus intéressant dans mon projet professionnel en tant qu'Analyste test et validation.
Ce stage m'a permis d'avoir une autonomie de travail surtout lors des périodes de télétravail. Il m'a aussi donné l'occasion de travailler souvent en mode debug sous eclipse pour suivre l'exécution de mes tests et détecter d'ou vient l'erreur.

Les tests unitaires réalisés au cours de mon stage ont prouvé le bon fonctionnement des fonctionnalités que j'ai testé. Une étape importante pour faire quelques correctifs des fonctionnalités. Après l'intégration continue, mes tests auront un intérêt important dans les développements prochains de projet Sim4Sys afin de pouvoir modifier facilement et rapidement le code.

Perspectives

La perspective de l'automatisation des tests unitaires est recommandée car elle est intégrée dans des outils d'intégration continue tel que Jenkins, Mettre en place une plate-forme d'intégration continue est une tâche technique assez longue. Mais une fois réalisé, c'est à la fois un confort de travail et une sécurité dont on ne peut plus se passer.

Voir l'importance de l'API EMF Compare explorée à la fin de mes tests, je recommande d'utiliser cette API pour l'implémentation des premiers tests que j'ai développé (test de la génération de la machine à état) car dans ce test, j'ai utilisé des implémentations basiques en java pour comparer des éléments UML. Comme vous avez vu, on peut avoir des tests efficaces et robustes avec une implémentation simple et légère avec EMF Compare.
EMF Compare peut être utilisé dans la comparaison des objets attendu avec les objets de tests dans la plupart des tests unitaires pour éviter les développements classiques Java afin de rendre les tests simples et robustes.
Parmi d'autres perspectives, je trouve que pour la fonctionnalité de la génération de la machine à état, il serait important de générer des "/exit" car il peut avoir des opérations importantes qui peuvent intervenir en sortie lors de la génération de diagrammes d'état.

Bibliographie

Documents internes à l'entreprise

https ://www.cil4sys.sharepoint.com/
https ://www.sim4sys.com/

Sites internet

https ://www.france-innovation.fr/membre/cil4sys-engineering/
https ://www.cil4sys.com/
https ://www.eclipse.org/forums/
https ://www.eclipse.org/papyrus/
https ://www.OpenClassrooms.com/
https ://www.junit.org/junit5/
https ://stackoverflow.com/
https ://www.eclipse.org/emf/compare/
https ://www.eclipse.org/emf/compare/documentation/latest/tutorial/tutorial.html
https ://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html
http ://www.hamcrest.org/JavaHamcrest/tutorial
https ://www.jenkins.io/doc/

Table des figures

Listings