UNIVERSITÉ DE NANTES

X8II031

UFR Sciences et Techniques

M1 ALMA
2016-2017

Faut-il tester le modèle ou le code des composants ?

Auteurs :
Montalvo Araya,
Charles-Eric Bégaudeau
et Charlène Servantie

2017

Remerciements

Nous aimerions remercier Pascal André, Gilles Ardourel et Jean-Marie Mottu avec lesquels nous avons travaillé au cours de ce module d'Introduction à la recherche pour leur patience et leurs explications essentielles à l'avancée de notre travail.

Table des matières

Introduction

Ce rapport reprend le travail qui a été effectué dans le cadre du cours d'Introduction à la Recherche du master 1 ALMA 2017 par notre groupe composé de Montalvo Araya, Charles-Eric Begaudeau et Charlène Servantie.

Le sujet sur lequel nous avons travaillé est : "Faut-il tester le modèle ou le code des composants ?", un sujet proposé par Pascal André, Jean-Marie Mottu et Gilles Ardourel de l'équipe AeLoS [1], qui ont été nos encadrants au cours de cette découverte du monde de la recherche.

Nous avons tout d'abord choisi de présenter le modèle Kmelia, le framework COSTO et les concepts abordés dans le chapitre 1, à travers notamment les différents articles que nous avons été amenés à lire pour appréhender le sujet de recherche. Ensuite nous allons parler dans le chapitre 2 sur le test de modèle à travers le générateur de harnais de COSTOTest, puis nous présenterons l'utilisation de JUnit pour faire du test sur le code Java généré et notre processus de travail pour générer ces tests dans le chapitre 3. Nous avons ensuite comparé le test de modèle avec le test de code dans le chapitre 4, où nous discutons également de notre travail et de nos blocages au cours de ce travail. Enfin, nous avons essayé d'apporter des réflexions sur de nouveaux tests et l'analyse de mutation au chapitre 5

1. Architectures et Logiciels Sûrs, https://ls2n.fr/equipe/AeLoS/

Chapitre 1

Kmelia et le framework COSTO

Contexte

Pour ce travail de recherche, nous avons été amenés à utiliser le framework COSTO ainsi que le langage de modélisation Kmelia, que nous présentons dans cette partie.

Nous avons choisi de reprendre certains des concepts sur lesquels reposent les travaux que nous avons lu pour pouvoir avoir une trace de l'ensemble des concepts avec lesquels nous avons été amenés à travailler au cours de ce travail et permettre une meilleure compréhension de l'ensemble.

L'équipe AeLos

L'équipe AeLos [1] est une équipe du LS2N [2] dirigée par Christian Attiogbé. AeLos travaille sur plusieurs axes de recherche en lien avec le développement de logiciels sûrs et les modèles.

1. Architectures et Logiciels Sûrs https://ls2n.fr/equipe/AeLoS/
2. Laboratoire des Sciences du Numérique de Nantes par lesquels ils sont définis. https://ls2n.fr/

Notions de test

Comme l'explique P. André et al. dans [1] dans le développement dirigé par les modèles, il est important de vérifier que le modèle abstrait est correct pour avoir un résultat le plus sûr possible. Cette vérification demande une combinaison de techniques de tests car on a un système comprenant des composants basés sur des services requis et des services fournis. Ce système a trois types de propriétés chacune vérifiée par une méthode spécifique :

- interaction, vérifiée par model checking,
- cohérence, vérifiée par theorem proving,
- conformation du comportement au contrat fonctionnel, vérifiée par model testing spécifique.

Développement dirigé par les modèles

Le développement dirigé par les modèles (souvent abrégé en MDD, pour Model-driven Development) est un type de développement basé sur une représentation du problème posé avec des méta-modèles et des modèles qui permettent également de décrire la solution que l'on souhaite apporter à ce problème. On utilise souvent un langage de modélisation pour représenter le problème ainsi que les solutions qu'on y apporte, et un exemple connu est l'UML [3]. Le langage Kmelia en est un autre exemple, sur lequel nous en parlerons plus en détails dans la section 1.

Vérification et Validation

Dans le cadre du développement de produit, et notamment de produit informatique, on teste la qualité du produit en utilisant à la fois de la vérification et de la validation.

La vérification correspond au test de correction d'une phase ou de l'ensemble. On vérifie que le produit fait ce qui est demandé sans bugs ou erreurs de réalisation.

La validation quant à elle repose sur la conformité du produit avec ce qui a été demandé par le client, c'est à dire que la réalisation convient aux spécifications et répond aux besoins exprimés.

3. Unified Modeling Language, est un langage de modélisation graphique, permettant la création de diagramme représentant le logiciel ou programme, http://www.uml.org

Model Checking

La vérification de modèles, ou model checking, consiste à tester si les propriétés d'interaction d'un modèle sont satisfaites.

Model Testing

Le test de modèles, ou model testing, consiste à réaliser des tests en boite noire en ce basant sur un modèle abstrait. On vérifie que le code correspond bien au modèle. Le harnais de test que nous aborderons dans la section 2 fonctionne sur ce principe.

La démonstration par expérimentation a été faite avec le langage de description de modèle Kmelia et l'outil COSTO, et l'article précise qu'il faut plus de travail et d'amélioration des outils pour avancer, notamment sur la partie de la spécification et de la vérification de la qualité de service. Il faut également de nouveaux éléments de langage pour les contraintes de temps et de ressources, ce qui nécessite des tests des techniques de vérification et validation associées.

Platform Independent Model PIM

C'est le modèle logiciel qui est indépendant de l'implémentation du logiciel (langage, ...). Ce terme est fréquemment utiliser dans le contexte d'une approche de l'architecture du logiciel basé sur les modèles. L'idée est de baser l'architecture du logiciel sur son modèle pour la rendre indépendante d'une quelconque plateforme d'implémentation.

Il existe des outils spécialisés pour ce genre d'approche tels que VIATRA, ATLAS ou Kmelia (CostoTest).

Component Testing

Ce type de test est utilisé dans les logiciels à composants. Pour s'assurer de la correction du logiciel il est nécessaire de s'assurer de la fiabilité de chacun de ses composants de manière individuelle.

Ce genre de test se concentre sur un composant à la fois ce qui nécessite dans la plus part des cas de créer des mocks pour simuler les autres composants que le composant sous test requiert pour fonctionner.

Langage Kmelia

Le langage Kmelia[2] est un langage de modélisation développé par l'équipe AeLoS, qui permet de décrire des modèles de composants basés sur des services eux-mêmes décrits par Kmelia. Les composants ainsi créés par Kmelia sont indépendant de l'environnement d'exécution et donc non exécutables, ce qui permet de s'affranchir de la contrainte de la plate-forme sur laquelle le programme sera exécuté.

Ce langage permet de spécifier les services requis par un composant ainsi que ceux qu'il fournit. Ceci permet de modéliser des systèmes complexes en garantissant autant que possible une bonne lisibilité de la conception et une indépendance de la plateforme d'exécution, ce qui permet également d'avoir des composants réutilisables et de faire de l'assemblage de composants.

La génération de code et l'analyse de ses spécifications ont été mises en oeuvre dans la plateforme COSTO, sous forme de plugins Eclipse.

```
1  COMPONENT VehicleTester
2  INTERFACE
3    provides : {testcase1}
4    requires : {computeSpeed}
5    autorun: {testcase1}
6  USES {PLATOONTESTLIB,PLATOONLIB,DEFAULT}
7  VARIABLES
8    obs verdict: Boolean ;
9  SERVICES
10
11 provided testcase1 ()
12 Interface
13   extrequires: {computeSpeed}
14 Variables
15 computespeedresult : Integer;
16 Sequence
17 {
18  # init sequence
19  # call
20   computespeedresult:= !! computeSpeed(getData("safeDistance"));
21   # oracle evaluation
22   verdict:= (computespeedresult = getData("oracledata"));
23   # transmit verdict
24   assertT(verdict);
25   #end of service
26   SendResult()
27   }
```

```
28 End
29
30 required computeSpeed(safeDistance : Integer) : Integer
31 End
32 END_SERVICES
```

Algorithme 1.1 – Exemple de fichier Kmelia décrivant un composant

On peut constater que le .kcp contient :

- COMPONENT qui définit le nom du composant,
- INTERFACE qui liste les compinterface, les interfaces du composant,
- USES qui liste les complibclause utilisés par le composant,
- VARIABLES qui liste les compobservvariablesclause, les variables observables du composant,
- SERVICES description des services du composant et de leur fonctionnement,
- provided qui liste les services fournis par le composant,
- Interface qui liste les servinterface, les interfaces du services,
- Variables qui liste les servvariablesclause, les variables du services
- required qui liste les services nécessaires au fonctionnement du composant,

Framework COSTO

COSTO est un outil qui se présente sous la forme de plugins pour Eclipse permettant notamment de générer du code à partir de fichiers Kmelia.

Cet ensemble de plugins a été développé pour supporter la spécification et l'analyse des systèmes de composants Kmelia. Il gère les spécifications Kmelia et la vérification des propriétés primaires (analyse syntaxique, vérification de type, analyse statique,...)[3].

COSTO utilise notamment le framework ANTLR[4] pour la reconnaissance du langage Kmelia et son interprétation. ANTLR prend en entrée une grammaire formelle définissant un langage et produit le code reconnaissant ce langage.

Le framework fournit une interface *IService* pour la gestion des services. Pour pouvoir appelé les services le framework fournis aussi des *IChannel*. Les channels appellent et récupère les données renvoyées pour un service donnée.

Un composant utilisant le framework a des *IProvidedService* qui sont les services fournis par le composant. Un composant possède aussi des *IRequiredService* qui sont les services nécessaire pour le fonctionnement du composant.

Il dispose également d'un plugin COSTOTest qui permet de générer des harnais de tests à partir du code qui a été généré par COSTO depuis des fichiers Kmelia. Le plugin permet de faire des tests sur des services précis et peut créer des mocks qu'on lui demande tout seul.

4. http://www.antlr.org/about.html

Les channels

Le langage Kmelia est un langage qui utilise des *goto*s, ce que le langage Java ne permet pas, à cause de cela le code généré doit imité cette fonctionnalité du langage de modélisation et donc crée des classes ***nom-classe****LTS* comme :

- SimpleDriver_giveSafeDistance
- SimpleDriver_giveSafeDistanceLTS

La première contenant la second et l'initialisant comme ceci :

```
1    public void initLTS(){
2         nom-classeLTS  lts =
3            new  nom-classeLTS ();
4        this.setLTS(lts);
5        lts.setService(this);
6        lts.init();
7    }
```

Algorithme 1.2 – Initialisation d'un LTS

La fonction callService()

CallService est une fonction de l'interface IService qui appelle un Service en prenant les paramètres suivant :

- le nom du channel, il est normalement sous la forme **_nom-du-service**
- le nom du service,
- les paramètres d'on le service a besoin. Attention ceux-ci ne sont pas les services requis,
- le service sur le quel on appelle la fonction ,

Vous pourrez retrouver en annexe le diagramme de séquence de la fonction callService 6 et de callService avec le mock testChannel 6.

```
1  IProvidedService serv = driv.getProvidedService("pos");
2
3  serv.callService("_pos", "pos",
4    new Object[]{mylib.PlatoonTestlibMap.getData("safeDistance")},
5    serv);
```

Algorithme 1.3 – Initialisation d'un LTS

Chapitre 2

Test du modèle et génération du harnais

Ce chapitre est basé principalement sur les recherches précédentes des étudiants de Master 1 et de Master 2 dont nous avons utilisé les rapports pour avancer dans notre travail.

Notre objectif étant d'effectuer une comparaison entre le test du modèle et le test du code, comme le travail sur le test du code avait déjà été effectué en amont par le groupe d'étudiants de Master 2[3], nous nous sommes documentés sur cette étape sans y apporter de nouvelles informations.

Ce chapitre repose principalement sur les rapports des étudiants précédents et sur la documentation disponible sur le site de COSTO [1]. Nous avons choisi de reprendre ces informations parce que nous avons passé du temps à les comprendre et qu'il nous parait important d'avoir une explication du modèle et de la génération du harnais au sein de ce rapport.

Modèle utilisé : Platoon System

Pour le test du modèle et la génération du harnais, nous avons travaillé avec l'exemple du PlatoonSystem [2] qui est un modèle représentant un système de véhicules qui se suivent avec un pilote au départ. Ce modèle est représenté schématiquement par la figure 2.1 avec les services fournis et requis. Ces véhicules doivent répondre à des contraintes de vitesse minimale, maximale et de distance de sécurité, chaque véhicule modifiant sa vitesse en fonction de celle du véhicule devant lui.

1. http://costo.univ-nantes.fr/
2. http://costo.univ-nantes.fr/application/vehicle-exemple/

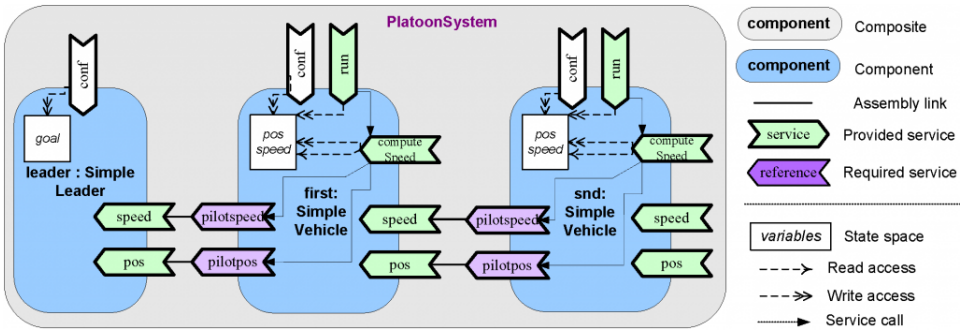


FIGURE 2.1 – Schéma du PlatoonSystem

Génération du harnais de test

COSTOTest est le plugin servant à générer et lancer les harnais de tests, on l'utilise en lançant Eclipse avec la bonne installation [3]

Pour lancer le harnais il faut cliquer sur *GenerateTH*, ensuite il faut sélectionner le service à tester. Ensuite le plugin affiche une fenêtre où il est possible de placer des mocks sur les services nécessaires au test. Une fois que tout est paramétré, il est possible de lancer le harnais de test. Une fenêtre s'ouvre et permet de choisir quelle valeur les mocks doivent renvoyer, le harnais procède à un test sur ces valeurs et génère un fichier de verdict. Par la suite il est possible de réutiliser un harnais de test sans devoir le re-paramétrer si on souhaite faire un test sur des valeurs. Il est aussi possible de tester plusieurs jeux de valeurs en même temps il suffit de préciser le nombre que l'on souhaite et de fournir les données de tests.

3. http://costo.univ-nantes.fr/tools/technical-environment-of-costo/eclipse-installation/

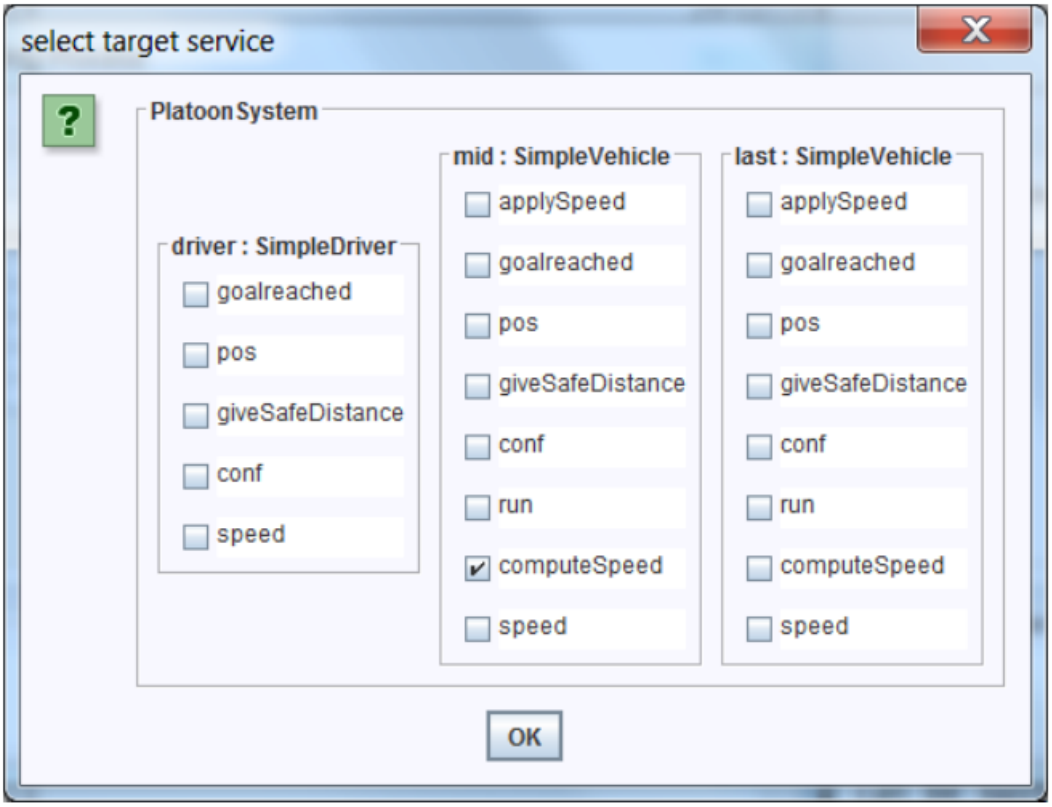


FIGURE 2.2 – Image sélection de service COSTOTest

Voici l'écran sur lequel on peut rentrer les valeurs de test et où on peut créer des mocks.

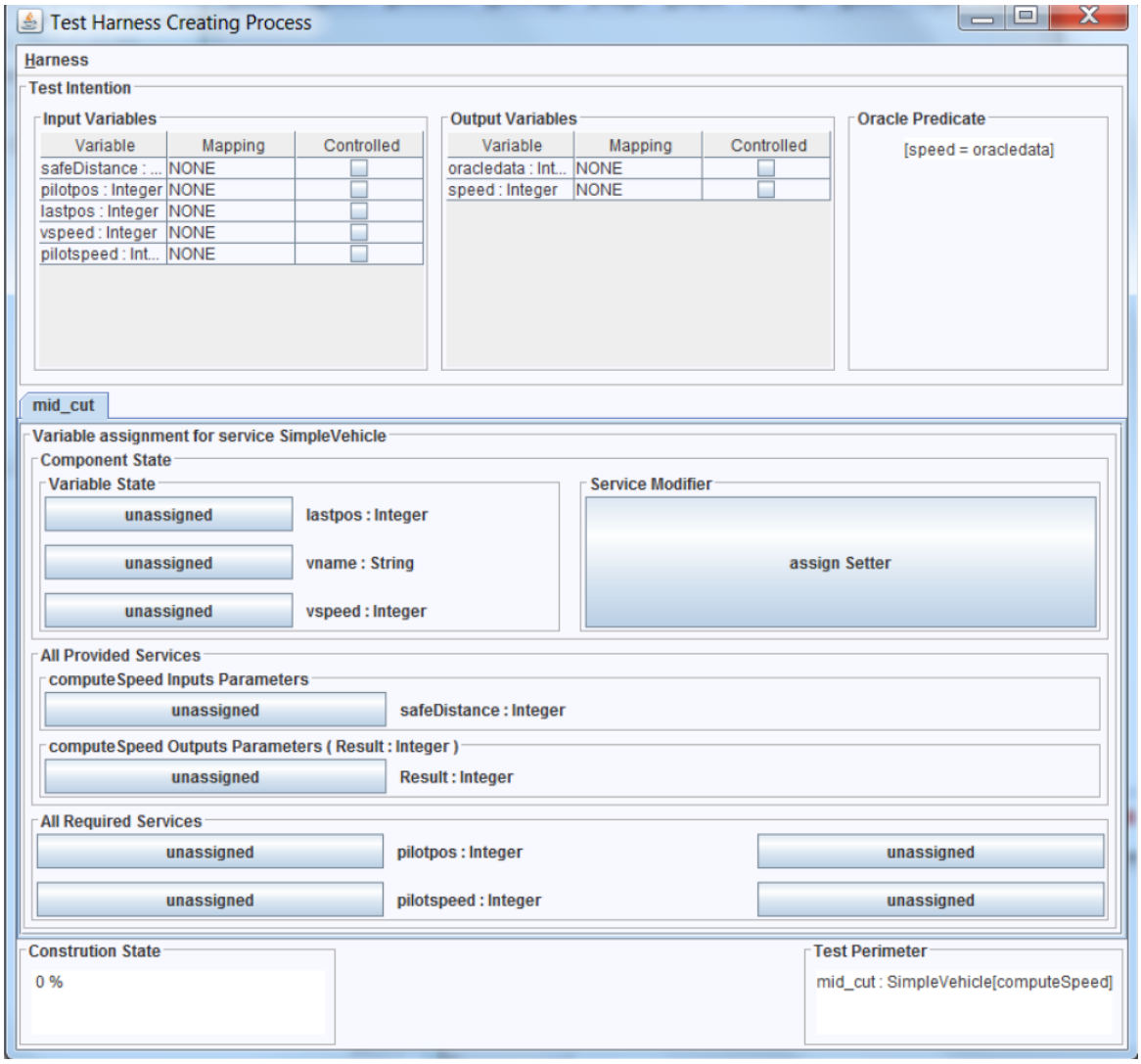


FIGURE 2.3 – Image sélection des valeur du test et crétion des mocks COSTOTest

Et voici à quoi ressemble un harnais de test complété sur le service *computeSpeed*

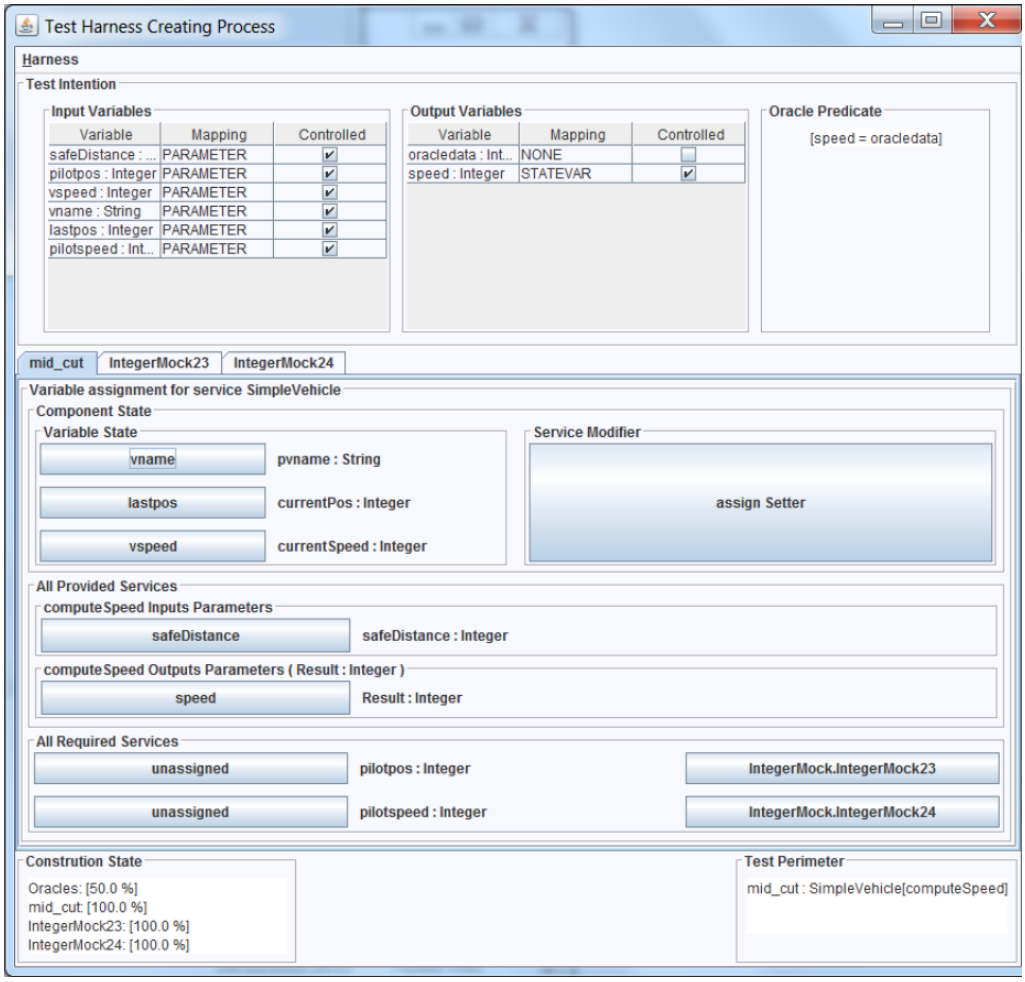


FIGURE 2.4 – Harnais complété COSTOTest

Chapitre 3

Test du code Java avec JUnit

Mise en place

Pour notre travail, nous avons utilisé la version standalone contenant les plugins COSTO et Eclipse, mais également effectué l'installation des différents plugins sur Eclipse Juno car une partie d'entre nous utilisait une distribution Linux.

Pour pouvoir garder les différentes versions de notre travail ainsi que pour pouvoir mieux travailler en groupe nous avons utilisé un dépôt Git [1].

Nous avons travaillé avec la version 4 de la librairie JUnit [2], et nous avions commencé à travailler avec Mockito [3] avant d'en abandonner l'usage car cela nous posait des soucis de version de JRE et qu'il était plus intéressant et potentiellement plus simple de chercher à faire nos mocks par nous-même.

Notre objectif étant d'établir un test du code généré, nous avons donc créé un nouveau package [4] dans lequel créer nos tests, de manière à ce qu'ils ne disparaissent pas si jamais nous régénérions le test pour une raison ou une autre.

1. https://github.com/E1drad/COSTO
2. http://junit.org/junit4/
3. http://site.mockito.org/
4. kmelia.autonomousSimplePlatoon.testsM1IR

Processus

Nous nous sommes renseignés sur les pratiques de tests unitaires ainsi que les séries de tests notamment grâce à l'ouvrage Tests Unitaires en Java [4] conseillé par nos enseignants référents.

Pour faire du test unitaire, il faut isoler le composant à tester du système et notamment de ses services requis de manière à pouvoir réellement tester le composant uniquement, et pas toute l'architecture sur laquelle il s'appuie derrière.

Ceci implique de prévoir des mocks pour simuler ces services et n'étudier que le comportement du composant.

Nous avons commencé dans notre travail par chercher à comprendre les différents appels de services effectués par le système dans l'exemple du PlatoonSystem. [5]

Au tout début nous avions voulu tester le service *computeSpeed* de la classe *SimpleVehicle* mais comme ce service était plutôt compliqué car il nécessite d'autres services pour fonctionner nous avons d'abord testé des services plus simple comme le service *pos* et *goalreached*.

Cela nous a permis de nous familiariser avec le système des channels du framework. Cependant nous nous sommes vite rendus compte qu'il serait plus simple d'utiliser des mocks pour les channels et nous avons donc créé une classe *TestChannel* qui est un mock qui simule le fonctionnement d'un channel.

Nous avons travaillé à la génération de test pour la méthode computeSpeed(). Nous avons utilisé la classe *TestChannel* pour ces tests. Pour les tests sur *computeSpeed* d'un *SimpleVehicle* nous avons assigné un *TestChannel* au service *computeSpeed* du véhicule. ComputeSpeed étant un service nécessitant les services pilotpos et pilotspeed nous avons créé des mocks sur ces services en utilisant *TestChannel*, ce channel fournit aussi une *safeDistance* car computeSpeed en a également besoin. Ensuite il suffit de fournir les valeurs que nous voulons via les mocks puis de récupérer le résultat de computeSpeed() pour faire les tests.

5. Dont la mise en place est expliquée ici :
http://costo.univ-nantes.fr/application/vehicle-exemple/

Blocages

L'ensemble de l'architecture du système nous a posé quelques soucis de compréhension lors de nos travaux.

Nous avons notamment bloqué pendant une longue période sur la compréhension du langage Kmelia, du fonctionnement de COSTO et de la génération du harnais de test par COSTOTest, même si nous n'avions pas besoin d'une compréhension en profondeur de tout le système.

Nous avons également tenté une approche réflexive sur laquelle nous avons rapidement bloqué à cause des channels (voir 1)

```
1 public Boolean isGuardSatisfied(String transition) {
2     if (transition==null)
3         return false;
4     if ("i___i1___1".equals(transition))
5         return this.service.guard_i___i1___1();
6     if ("i1___f___2".equals(transition))
7         return this.service.guard_i1___f___2();
8     return true;
9 }
```

Algorithme 3.1 – exemple de fonction généré

Channel

Nous avons eu du mal à comprendre à comment se servir des channels pour appeler les services et recevoir leurs résultats. Notamment comprendre comment fonctionne la fonction *callService()* qui est la fonction qui permet d'appeler les services via un channel.

Par la suite nous avons utilisé une classe qui est un mock de la classe Channel pour pouvoir faire les tests plus facilement.

Code

Voici le code de la classe TestChannel que nous a fourni M. Ardourel pour pouvoir faire un mock de channel pour utiliser dans nos tests.

```
1  package kmelia.autonomousSimplePlatoon.testsM1IR;
2  /**
3   * Classe mock channel fournie par Gilles Ardourel
4   */
5  public class TestChannel extends Channel{
6      private Object result;
7      private Object[] callparams;
8      private Map<String,Object[]> mockvalues;
9      public void setCallparams(Object... callparams) {
10         this.callparams = callparams;
11     }
12     /**
13      * @param callname name of the required service to mock
14      * @param values results to be returned by the
15      * mocked service
16      */
17     public void addMockValue(String callname, Object...values){
18         mockvalues.put(callname, values);
19     }
20     public Object getResult() {
21         return result;
22     }
23     public TestChannel(String name, IService client,
24         IService server) {
25         super(name, client, server);
26         mockvalues= new HashMap<>();
27     }
28     public void clearMockValue(){
29         mockvalues.clear();
30     }
31     @Override
32     public Object[] receiveMessage(String channel,
33         String message, Class<?>[] paramtypes,
34         IService orig) {
35         System.err.println("␣the␣Service␣wants␣to␣"
36             + "receive␣a␣message,␣use␣a␣mock␣"+message);
37         return null;
```

```
38      }
39      @Override
40      public void emitMessage(String channel, String message,
41          Object[] params, IService orig) {
42          System.out.println("␣the␣Service␣emits␣"+message);
43      }
44      @Override
45      public void callService(String channel, String message,
46          Object[] params, IService orig)
47          throws KmlCommunicationException {
48          System.out.println("the␣service␣calls␣another␣"
49              + "Service␣"+channel+"␣"+message);
50      }
51      @Override
52      public void returnService(String channel, String message,
53          Object[] params, IService orig) {
54          System.out.println("the␣service␣"+message+"␣returns␣"+
55              params[0]);
56          orig.ack(this,0);
57          result=params[0];
58      }
59      @Override
60      public Object[] receiveServiceCall(String channel,
61          String message, Class<?>[] paramtypes, IService orig) {
62          return this.callparams;
63      }
64      @Override
65      public Object[] receiveServiceReturn(String channel,
66          String message, Class<?>[] paramtypes, IService orig) {
67
68          return mockvalues.get(message);
69      }
70      @Override
71      public void close(IService source) {
72          // Overriding to NOP because this channel is one-ended
73      }
74      @Override
75      public void cut(IService source) {
76          // Overriding to NOP because this channel is one-ended
77      }
78  }
```

Algorithme 3.2 – TestChannel.java

A partir de ce TestChannel, cela nous a permis de créer une suite de tests pour tester la méthode computeSpeed(). Pour faciliter la lecture des test à partir du deuxième test, nous avons supprimé les parties qui ne changent pas, le code complet étant disponible sur notre dépôt git. Voici le code :

```
1  package kmelia.autonomousSimplePlatoon.testsM1IR;
2
3  public class TestOnComputeSpeed {
4      /**
5       * Test quand le vehicle est suffisament loin du driver.
6       * @throws InterruptedException
7       * @throws KmlCommunicationException
8       * @throws ServiceException
9       */
10     @Test
11     public void testComputeSpeedOk()
12         throws InterruptedException,
13         KmlCommunicationException, ServiceException{
14
15         SimpleVehicle veh = new SimpleVehicle("SimpleVehicle",
16         new TestOuterContext("test"),"last");
17         int posValue=10;
18         veh.setConfig("conf","last",posValue,0);
19         veh.init();
20
21         IProvidedService provServToTest =
22             veh.getProvidedService("computeSpeed");
23
24         //Create and assign fake test channel
25         TestChannel testChan = new TestChannel(
26             "TESTCHANN",null,provServToTest);
27         provServToTest.assignChannel(testChan);
28         veh.getRequiredService("pilotpos")
29             .setReqChannel(testChan);
30         veh.getRequiredService("pilotspeed")
31             .setReqChannel(testChan);
32
33         //assigning call parameters for Service under test
34         testChan.setCallparams(120);// safeDistance
35         testChan.addMockValue("pilotpos", 420);
36         testChan.addMockValue("pilotspeed", 40);
37
38         provServToTest.start();
39
```

```
40          Thread.sleep(1000);
41
42          provServToTest.stop(testChan);
43
44          System.out.println(testChan.getResult());
45          assertEquals(true,
46              (Integer)testChan.getResult()>0);
47      }
48
49      /**
50       * Test quand le vehicle est trop proche du driver
51       * @throws InterruptedException
52       * @throws KmlCommunicationException
53       * @throws ServiceException
54       */
55      @Test
56      public void testComputeSpeedTropProche()
57          throws InterruptedException,
58          KmlCommunicationException, ServiceException{
59              [...]
60          int posValue=10;
61              [...]
62          testChan.setCallparams(120); // safeDistance
63          testChan.addMockValue("pilotpos", 40);
64          testChan.addMockValue("pilotspeed", 40);
65              [...]
66          System.out.println(testChan.getResult());
67          assertEquals(true,
68              (Integer)testChan.getResult()==0);
69      }
70      /**
71       * Test quand le vehicle est devant le driver
72       * @throws InterruptedException
73       * @throws KmlCommunicationException
74       * @throws ServiceException
75       */
76      @Test
77      public void testComputeSpeedDevantDriver()
78          throws InterruptedException,
79          KmlCommunicationException, ServiceException{
80              [...]
81          int posValue=100;
82              [...]
83          System.out.println(testChan.getResult());
```

```
84          assertEquals(true,
85              (Integer)testChan.getResult()==0);
86      }
87      /**
88       * Test quand le driver va tres vite.
89       * Cela ne doit normalement pas affecter le vehicule
90       * @throws InterruptedException
91       * @throws KmlCommunicationException
92       * @throws ServiceException
93       */
94      @Test
95      public void testComputeSpeedDriverRapide()
96          throws InterruptedException,
97          KmlCommunicationException, ServiceException{
98              [...]
99          int posValue=10;
100         veh.setConfig("conf","last",posValue,0);
101         veh.init();
102             [...]
103         System.out.println(testChan.getResult());
104         assertEquals(true,(Integer)testChan.getResult()>0);
105     }
106     /**
107      * Test quand la safe distance est negative
108      * @throws InterruptedException
109      * @throws KmlCommunicationException
110      * @throws ServiceException
111      */
112     @Test
113     public void testComputeSpeedNegativeSafeDistance()
114         throws InterruptedException,
115         KmlCommunicationException, ServiceException{
116             [...]
117         int posValue=10;
118         veh.setConfig("conf","last",posValue,20);
119         veh.init();
120             [...]
121         testChan.setCallparams(120); //safeDistance
122         testChan.addMockValue("pilotpos", 400);
123         testChan.addMockValue("pilotspeed", 4000);
124             [...]
125         System.out.println(testChan.getResult());
126         assertEquals(true,(Integer)testChan.getResult()>0);
127         }
```

```
128     /**
129      * Test quand le vehicule a deja une vitesse
130      * @throws InterruptedException
131      * @throws KmlCommunicationException
132      * @throws ServiceException
133      */
134     @Test
135     public void testComputeSpeedDejaLancer()
136         throws InterruptedException,
137         KmlCommunicationException, ServiceException{
138             [...]
139         int posValue=10;
140         veh.setConfig("conf","last",posValue,20);
141         veh.init();
142             [...]
143         //assigning call parameters for Service under test
144         testChan.setCallparams(-1); //safeDistance
145         testChan.addMockValue("pilotpos", 400);
146         testChan.addMockValue("pilotspeed", 4000);
147             [...]
148         System.out.println(testChan.getResult());
149         assertEquals(true,(Integer)testChan.getResult()>0);
150     }
151 }
```

Algorithme 3.3 – TestOnComputeSpeed.java

Chapitre 4

Comparaison entre le harnais de test et JUnit

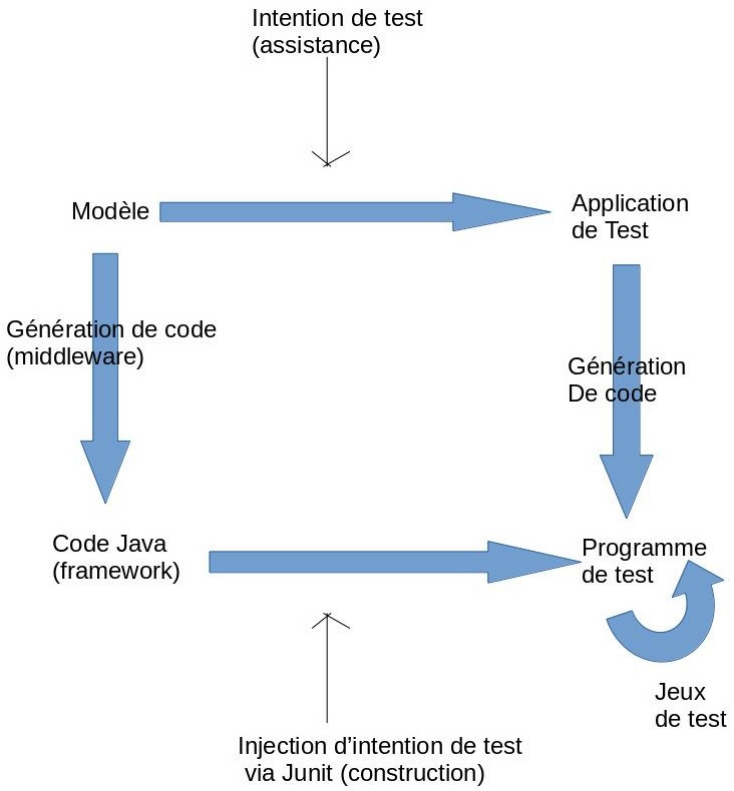


FIGURE 4.1 – Structure des tests sur le modèle par assistance ou construction

Sur la figure ci-dessus, on peut voir les deux manières de tester un modèle, soit à la manière du harnais de tests créés par COSTOTest en utilisant

les intentions de tests Kmelia, soit comme nous avons cherché à le faire en générant du code conforme au modèle que nous testons ensuite manuellement en injectant nos intentions de tests via JUnit.

L'hypothèse de départ est qu'il est plus rentable en terme de temps de générer les tests sur le modèle et non pas sur le code généré à partir du modèle.

Intérêts du test sur le modèle par rapport à Junit

Grâce à l'utilisation du générateur de harnais de tests dans COSTOTest en suivant la procédure indiquée dans la documentation, la génération du test de computeSpeed() est relativement rapide à mettre en place, les fonctionnalités du plugin permettant d'intégrer les données multiples en une seule fois grâce à un fichier .csv contenant les données d'entrée et les oracles pour les tests.

De plus, la présence et l'utilisation de frameworks et autres middlewares pour que le programme gère notamment la concurrence et les buffers de communications rend la génération de tests par JUnit plus complexe.

Notre avis

Après tout le temps passé à comprendre comment fonctionne COSTO, le harnais de test généré, utiliser JUnit nous paraît moins efficace notamment car il faut créer soi-même les mocks que COSTOTest génère automatiquement.

Cependant malgré la pénibilité de devoir utiliser JUnit ou tout autre framework de test (Mockito, Cobertura, ...), il reste utile de test le "*produit fini*".

De manière plus pratique l'avantage de Junit par rapport à Kmelia est que plus personnes sont former à son utilisation et qu'il ne demande pas d'apprendre un nouveau langage.

Chapitre 5

Mutations et autres tests

Nous avions pour objectif après la génération des premiers tests de réaliser une analyse de mutations sur le code généré pour vérifier que la suite de tests utilisée détectait les erreurs introduites dans du code, mais nous n'en avons pas encore eu l'occasion, ayant passé un temps conséquent sur la réalisation des tests pour computeSpeed.

Chapitre 6

Divers

Dans cette partie nous avons voulu présenter une partie du travail que nous avons fourni et qui ne rentrait pas dans la partie principale (les tests JUnit 3 et la comparaison entre le harnais et JUnit 4), notamment les retours que nous avons fait par rapport à des bugs.

Bugs

Au cours de notre travail avec COSTOTest nous avons été confrontés à plusieurs bug que nous avons remontés mais pour lesquels nous n'avons pas pu fournir de correctif.

Génération incomplète

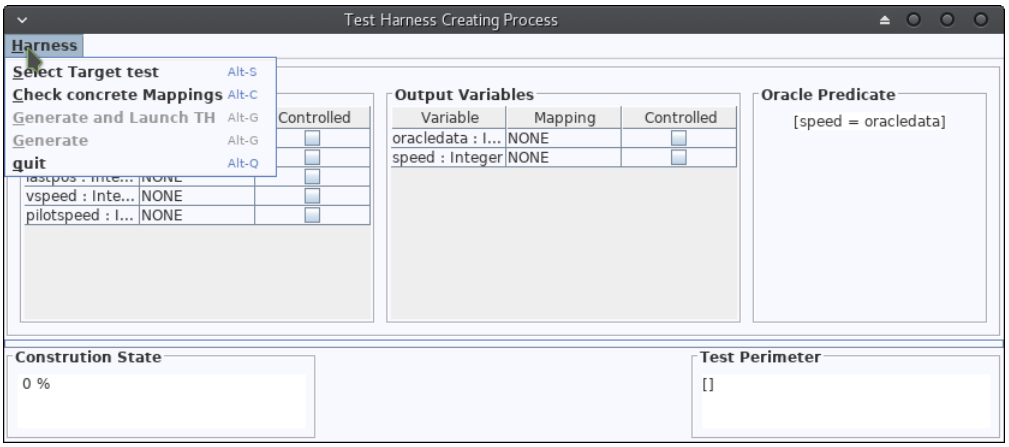
Lorsqu'on génère un harnais de test en utilisant *Generate and Launch TH*, le package mylib nécessaire à une partie du code généré n'est lui-même pas généré. [1]

Pour éviter ce bug, il suffit de lancer l'opération de génération sans la mener à sa fin puis de faire *Laucnh TH* sur la classe TH_PlatoonTestIntention.kcp qui va générer le package mylib.

1. Ce package n'est pas non plus généré lors de l'utilisation de Kml2java

Menu Harness

Nous avons constaté un bug mineur lors de l'utilisation de la création de harnais, si jamais on ouvre le menu Harness et qu'on clique sur quit, ça ferme complètement Eclipse au lieu de juste quitter la création de harnais.



Nom de véhicule bloquant

Lorsqu'on utilise *Generate TH M2Alma*, si on sélectionne *Select Target Test* dans le menu *TestHarness Creating Process* et qu'on sélectionne SimpleVehicle avec comme seul paramètre computespeed et qu'on cherche à assigner une valeur au String vname puis qu'on annule sans valider le nom du Vehicle, il devient impossible d'assigner un String vname. La fenêtre ne réapparaît plus et le boutont devient inutilisable, nécessitant un redémarrage d'Eclipse pour pouvoir à nouveau travailler avec.

Conclusion

Cette découverte de la recherche à travers un cas concret encadré nous a permis de mieux appréhender à quoi correspond le travail d'un chercheur, et de voir au travers des différentes étapes de réflexion comment ça se passe.

Nous avons mis un certain temps à réellement saisir le travail demandé car nous étions partis dans l'optique qu'on nous présente un problème et que nous devions trouver une solution qui fonctionne forcément, alors que le but de la recherche est d'émettre des hypothèses, de faire des tests pour étudier cette hypothèse et sa validité, d'en analyser les résultats et de recommencer avec une nouvelle hypothèse et donc de nouveaux tests.

Ce travail nous a sensibilisé à l'importance de la documentation de la recherche, pas juste au travers des articles et rapports à écrire en conclusion des recherches, mais surtout tout au long du processus pour garder une trace des différentes pistes suivies et pouvoir plus facilement communiquer sur le travail effectué.

Bibliographie

[1] Pascal André, Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1 : MODELSWARD*, pages 645–656, 2017.

[2] Pascal André, Gilles Ardourel, and Christian Attiogbé. Kmelia : un modèle abstrait et formel pour la description et la composition de composants et de services, revue des sciences et technologies de l'information - série tsi : Technique et science informatiques, lavoisier, 2011, technique et science informatiques. pages 627–658, 2010.

[3] Hamza Boukil, Afaf Chouidra, Hicham Dabihi, Naoufal El Gafa, Yassine Hamzaoui, and Aya Saidi. Rapport projet costo, 2015.

[4] Johanes Link and Peter Fröhlich. *Tests Unitaires en Java.* Dunod edition, 2003.

Annexes

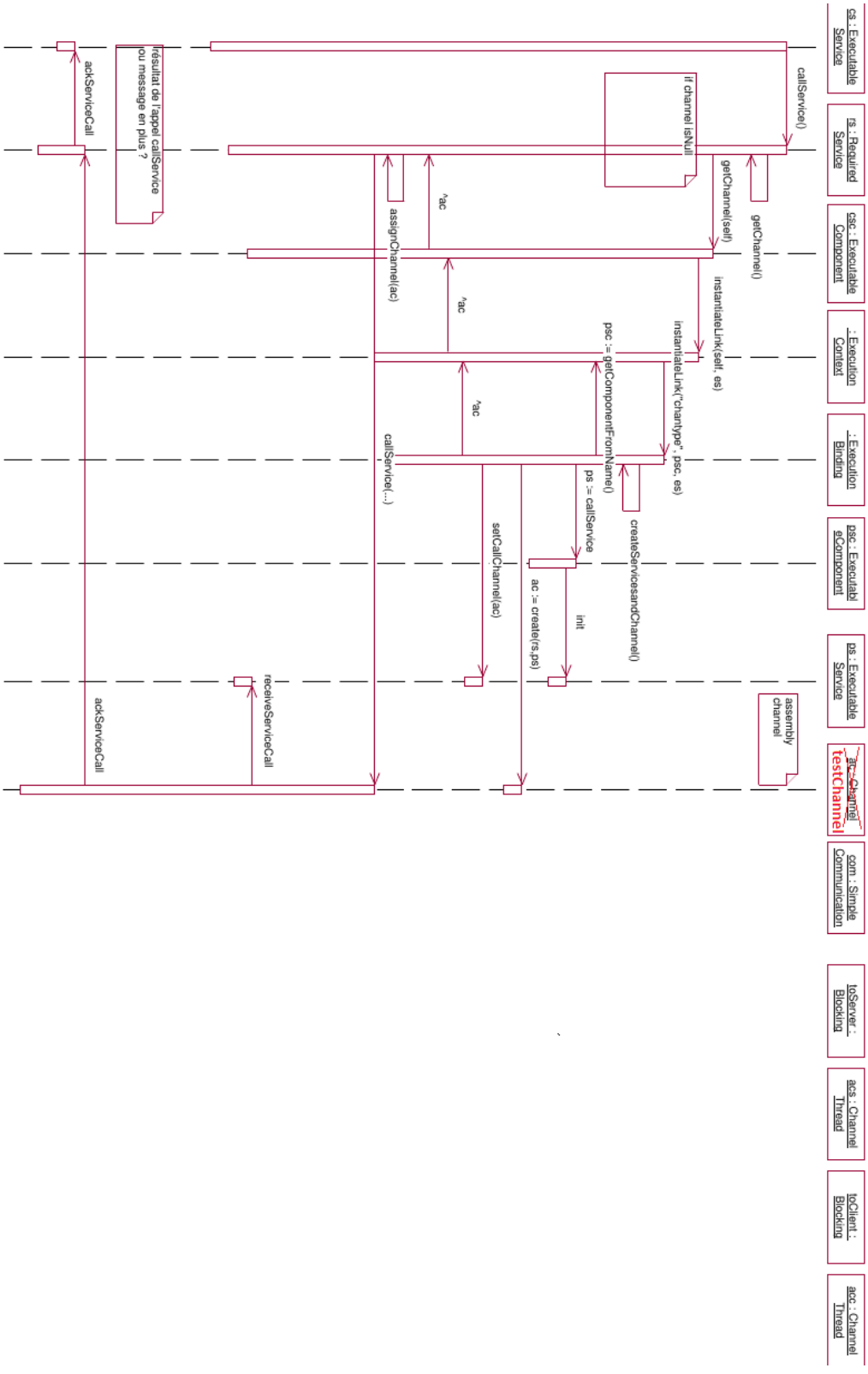
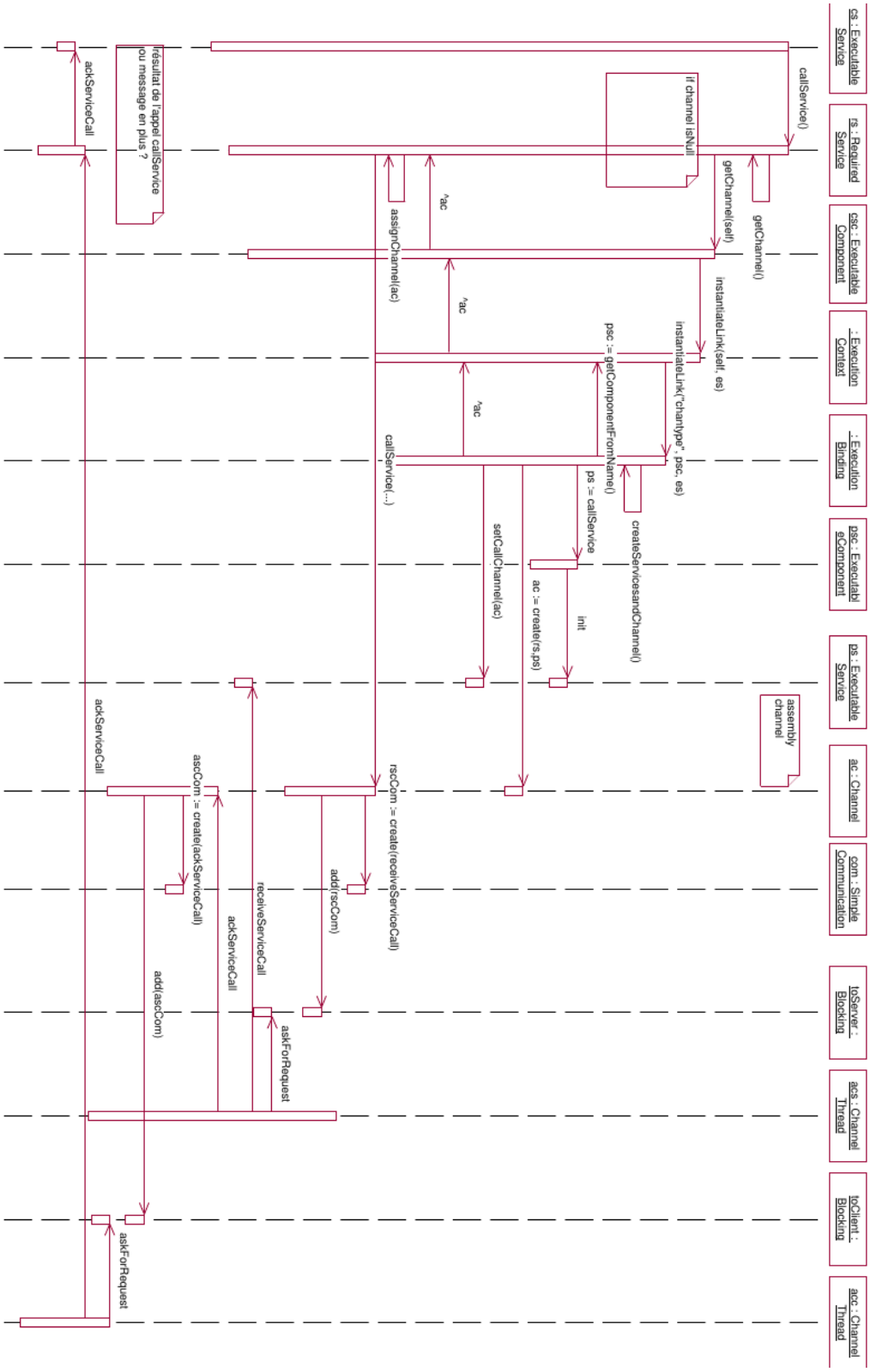


FIGURE 6.1 – Diagramme de séquence de callService avec l'utilisation du TestChannel en mock



35
FIGURE 6.2 – Diagramme de séquence de callService