Master Informatique et Télécommunication
Parcours Systèmes Informatiques et Génie Logiciel

DTest : un Framework de Tests pour Applications Distribuées

Auteur : Lionel Duroyon

Responsable du stage : M. Eric Noulard
Equipe d'accueil : DTIM/SER

Onera Centre de Toulouse

Juin 2008

Résumé

DTest est un Framework de test d'applications distribuées écrit en Python. L'objectif premier de DTest est de : « Mettre en œuvre de façon simple des tests fonctionnels et/ou d'intégrations *automatisables* pour des applications *distribuées* » c'est-à-dire de vérifier que l'exécution distribuée de différents processus correspond à un comportement attendu.

Pour cela, DTest se base sur l'observation des éléments distribués de l'application soumise au test. Une observation est un élément donnant une information sur le comportement d'un processus (sorties standard, variables d'états, paquets TCP).

DTest a comme objectif plus large de créer un framework générique permettant l'exécution de tests distribués et faisant le lien avec des techniques de vérifications formelles.

UNIVERSITE PAUL SABATIER TOULOUSE III

ONERA
THE FRENCH AEROSPACE LAB

Remerciements :

Je remercie mon responsable de stage Eric Noulard pour son encadrement de stage motivant, sa disponibilité et ses conseils.

Je remercie également les stagiaires de la « poule moyenne » du DTIM : Florian, Alexandrine et Pierre pour leur bonne humeur et leur gentillesse.

Je tiens également à remercier les thésards et les autres stagiaires du DTIM : Loïc, Mikel, Cédric, Stéphanie, Julien, Sophie et Joël ainsi que tout le personnel du centre de Toulouse de l'Onera.

Table des matières

1 Introduction

1.1 Présentation de l'ONERA

L'Office National d'Etudes et Recherches Aérospatiales est un établissement public français (EPIC) de recherche dédié au secteur aéronautique et spatial. Il a été créé en 1946.

L'ensemble des travaux effectués par l'ONERA sont liés à la recherche finalisée, ils se situent dans de nombreux domaines comme l'avionique, la défense ou encore les transports spatiaux, et touchent de nombreuses disciplines, de la chimie à l'informatique.

L'ONERA est financé à la fois par l'état, concernant les recherches à long terme, et sous contrats avec des entreprises pour des travaux plus courts. L'Office mène des travaux sur de grands enjeux sociétaux pour la France et l'Europe : développer la compétitivité industrielle, protéger l'environnement, renforcer la sécurité.

Sa division en quatre branches : mécanique des fluides et énergétique, physique, matériaux et structures, traitement de l'information et systèmes lui permet d'être présent dans la plupart des disciplines scientifiques liées à l'aérospatial.

L'ONERA conserve aussi une double approche calcul - expérimentation pour ses travaux, ce qui lui permet d'assurer un processus solide d'acquisition de connaissances.

De plus, depuis 60 ans, l'ONERA a tissé des liens avec les industriels et reste ainsi très compétent en ce qui concerne la connaissance de l'application des recherches. L'entreprise possède un parc de moyen d'essais unique en Europe.

Le centre de Toulouse de l'ONERA a été créé lors du transfert de l'Ecole Nationale Supérieure de l'Aéronautique et de l'Espace à Toulouse.

1.2 Présentation du DTIM

L'émergence et le déploiement très rapide des nouvelles technologies de l'information et de la communication en ont fait un secteur industriel et commercial majeur. Le traitement de l'information a de surcroit, par son rôle dans la maîtrise des performances, des coûts et de la sécurité, un impact déterminant sur la conception, le déploiement et l'exploitation de systèmes aérospatiaux.

Dans ce contexte le Département Traitement de l'Information et Modélisation (DTIM) a pour mission d'élaborer des recherches disciplinaires et pluridisciplinaires propres à ce domaine scientifique en inscrivant ses résultats dans les perspectives et finalités principales suivantes :

- aéronautique : aide à la conception, méthodes et moyens de réalisation, fonctionnement et exploitation optimale, environnement d'exploitation
- espace : évolution des concepts de missions, autonomie, répartition bord/sol, exploitation des satellites et notamment imagerie, et photo-interprétation
- défense : sécurité des systèmes, systèmes d'information et de commandement, renseignement, fusion de capteurs, fusion de données, guidage d'engins

Le DTIM développe des études et des recherches débouchant sur l'élaboration :

- de modèles et méthodes aptes à affronter la complexité de systèmes dont les constituants peuvent être des systèmes informatiques, des systèmes techniques, des agents artificiels ou des personnes
- de techniques et d'outils informatiques qui supportent de façon robuste ces modèles et ces méthodes.

1.3 Plan du rapport

En partant d'une implémentation fonctionnelle de DTest, ce stage avait deux objectifs :
- le premier, pratique, consistant à enrichir DTest avec des fonctionnalités utiles et contribuer à l'automatisation de l'exécution de tests distribués avec un exemple pour le logiciel CERTI[1] (développé au centre de Toulouse de l'ONERA)
- le second, exploratoire, consistant à examiner et mettre en œuvre des techniques de vérification formelles en lien avec ces tests distribués

La partie pratique a nécessité l'utilisation du langage Python et la compréhension du fonctionnement d'une simulation distribuée HLA[2].

La partie exploratoire a été constituée d'une recherche bibliographique dans le domaine du test logiciel, de la lecture de différents articles scientifiques sélectionnés et d'un approfondissement de connaissance dans les techniques de model-checking.

On exposera le travail réalisé de la manière suivante :

Après avoir introduit DTest et son architecture, nous présenterons les principes de base de DTest qui se sont révélés au long du stage :
- Les pas de test
- Les observations
- Les actions
- La synchronisation
- Les traces d'exécution

Nous montrerons ensuite comment fonctionne DTest à l'aide d'un exemple simple de définition d'une séquence de tests distribués.

Puis, nous situerons DTest par rapport :
- Aux différentes techniques de tests
- A la génération de test
- Aux autres outils de tests distribués
- A l'ingénierie des modèles

Enfin, nous aborderons la partie liée à la vérification de propriétés sur :
- Les séquences de tests
- Les traces d'exécution

2 Présentation de DTest

DTest est un Framework de test d'applications distribuées écrit en Python. L'objectif premier de DTest est de : « Mettre en œuvre de façon simple des tests fonctionnels et/ou d'intégrations *automatisables* pour des applications *distribuées* » c'est-à-dire de vérifier que l'exécution distribuée de différents processus correspond à un comportement attendu.

Pour cela, DTest se base sur l'observation des éléments distribués de l'application soumise au test. DTest est un sous-projet de TSP[3], et est utilisé pour tester le projet CERTI. Il est publié avec une licence Open Source LGPL.

DTest a comme objectif plus large de créer un framework générique permettant l'exécution de tests distribués et faisant le lien avec des techniques de vérifications formelles.

[1] http ://www.cert.fr/CERTI/index.fr.html
[2] http ://www.onera.fr/dtim/simulation-distribuee/index.php
[3] http ://savannah.nongnu.org/projects/tsp

2.1 Architecture de DTest

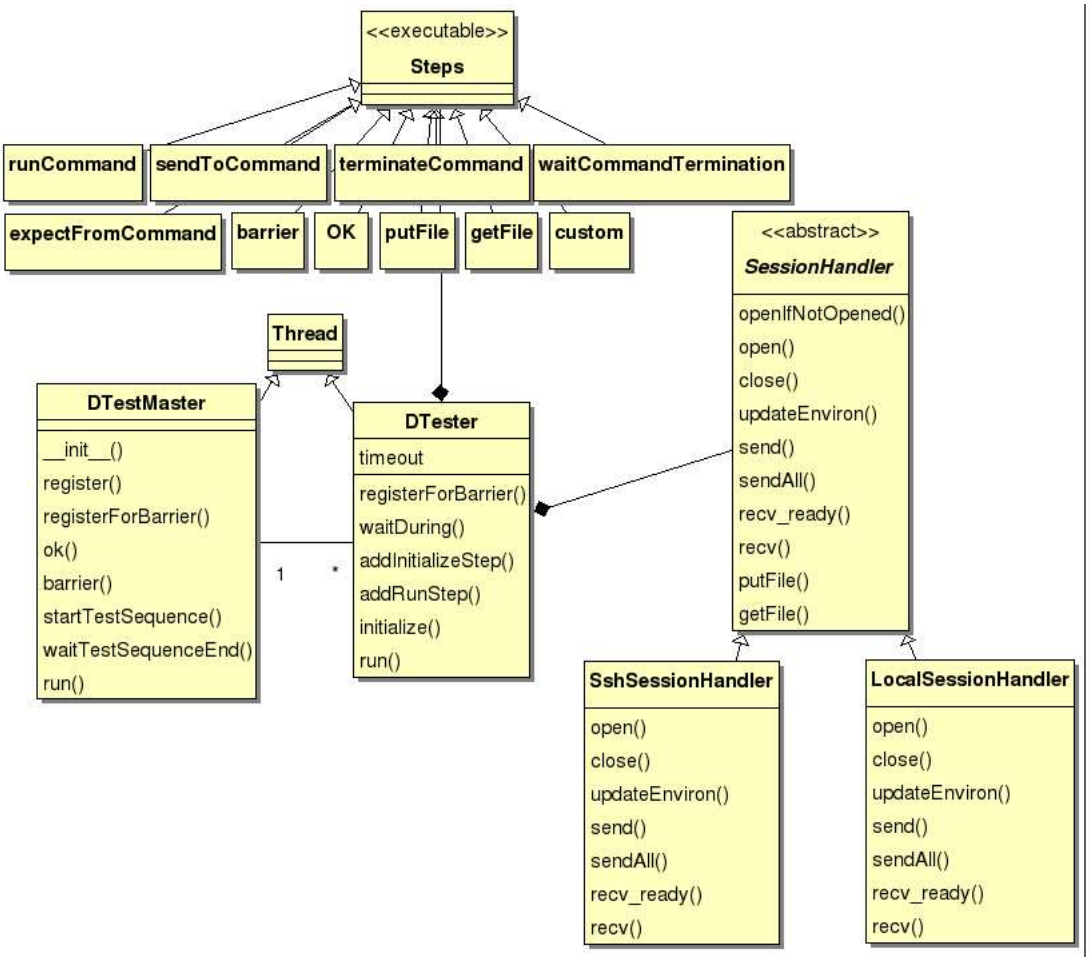


FIG. 1 – Diagramme de classes de DTest

Le diagramme de classes de DTest (cf figure 1) permet de présenter rapidement son architecture. Pour définir une séquence de tests distribués, on ajoute des pas de test aux DTesters. Un DTester est associé à un gestionnaire de session assurant la gestion de la connexion à une machine (dans l'implémentation actuelle un gestionnaire de session SSH). Le DTestMaster coordonne tous les DTesters. Il assure leur synchronisation grâce au mécanisme de barrière (voir chapitre 2.5).

Nous allons maintenant décrire les principes mis en œuvre dans DTest.

2.2 Les pas de test dans DTest

Voici la liste des pas de test pouvant être ajoutés à un DTester :

- *ok* : méthode d'affichage Test Anything Protocol[4] indiquant si un pas de test réussit ou échoue
- *runCommand* : exécute une commande à travers le gestionnaire de session
- *expectFromCommand* : bloque le testeur jusqu'à ce qu'un motif de chaine de caractère soit disponible sur le gestionnaire de session
- *terminateCommand* : envoie au gestionnaire de session l'ordre de terminer la session

[4] http ://testanything.org/wiki/index.php/

– *barrier* : bloque le testeur jusqu'à ce que chaque participant à la barrière ait atteint la barrière spécifiée
– *sendToCommand* : envoie directement une chaîne de caractère au gestionnaire de session
– *waitCommandTermination* : attend jusqu'à ce que la session se termine
– *custom* : permet de lancer n'importe quel « callable » Python

Chaque DTester gère 3 fichiers contenant les envois, réceptions et erreurs de la session. Ces fichiers peuvent être utilisés à des fins de mise au point.

2.3 Les observations

Dans DTest, une observation est un élément donnant une information sur le comportement d'un processus. Ces éléments peuvent être :

1. Les sorties standard (actuellement utilisées grâce au pas de test expectFromCommand)
2. Les variables d'états
3. Les paquets TCP

D'autres éléments à observer pourront être utilisés, en fonction des moyens fournis par chaque application.

2.4 Les actions

Une action est une stimulation d'un équipement sous test.

On peut distinguer deux types d'actions :
– Les actions de lancement des processus
– Les actions au cours de la séquence d'exécution rendant le test *actif*

Actuellement dans DTest, on utilise le pas de test runCommand pour interagir avec les processus à tester.

2.5 La synchronisation

Il existe deux moyens de synchronisation dans DTest :
– Le mécanisme de barrière utilisé pour synchroniser les différents DTesters. Sa sémantique est la suivante : un DTester atteignant le pas de test « barrierA » est bloqué jusqu'à ce que tous les DTesters participant à cette barrière aient atteint le pas de test « barrierA ».
– Les pas de test « expectFromCommand » des DTesters permettent d'attendre l'arrivée d'une chaîne de caractères sur la session associée. Le DTester est bloqué jusqu'à l'arrivée de cette chaîne (un timeout est déclanché en cas d'attente trop longue). Le testeur peut ainsi vérifier le comportement d'un processus en observant ses sorties standard.

NB : On synchronise les DTesters en fonction des autres DTesters et de l'exécution des processus sous test. Les processus sous test s'exécutent indépendemment des DTesters.

2.6 Les traces d'exécution

Une trace d'exécution consiste à associer un traitement à l'ensemble des pas de test. Grâce à un « Trace Handler » on peut obtenir des traces d'exécution de différentes formes :
– Test Anything Protocol : affiche si le résultat de l'exécution d'un pas de test est ok/nok (trace par défaut)

– Message Sequence Charts (MSC cf annexe A) : permet de visualiser l'exécution de la séquence de tests sous la forme d'un diagramme proche d'un diagramme de séquences
– Trace Promela : on modèlise le comportement des différents testeurs de la séquence de tests à l'aide de code Promela
– Trace d'observations : contient les observations collectées lors de l'exécution de la séquence de tests

Les trois dernières traces ont été rajoutées lors du stage.

Génération de traces d'exécution

Une des premières fonctionnalités ajoutée lors du stage à DTest a été de permettre la génération de trace d'exécution sous la forme de diagrammes Message Sequence Charts.

Deux modes de génération ont été ajoutés :

– La trace réelle sérialisant l'exécution de chaque pas de test de chaque DTester auprès du DTest-Master.
– La pseudo-trace permettant de simuler l'exécution des pas de tests dans le but de vérifier des propriétés sur la séquence de tests sans l'exécuter.

3 Exemple d'utilisation de DTest

3.1 Exemple du script md5

Nous allons illustrer comment fonctionne DTest à travers l'exemple d'une copie de fichiers. Le but de cet exemple est de s'assurer qu'un fichier a bien été copié d'un serveur A à un serveur B. Pour faire cela, nous voulons comparer la somme md5 du fichier original sur A à la somme md5 du fichier copié sur B. On suppose, pour cet exemple, que Python est disponible sur les machines A et B.

Ici, seulement une barrière est utilisée (lignes 119 et 130) pour s'assurer qu'on ne copie pas le fichier sur le serveur B avant de l'obtenir du serveur A.

Les pas de test expectFromCommand (lignes 114 et 135) sont utilisés pour attendre que le script générant la somme md5 écrive « md5done » sur sa sortie. Une fois écrit, nous pouvons être sûr que les fichiers de résultats associés au script calculant la somme md5 ont été générés et nous pouvons récupérer ces fichiers résultats.

Le scénario réalisé par le script est le suivant :

1. placer le script md5 sur le serveur A - ligne 110
2. générer la somme md5 du fichier source - ligne 113
3. récupérer le fichier source du serveur A - ligne 116
4. récupérer la somme md5 du fichier source - ligne 118
5. placer le script md5 sur le serveur B - ligne 128
6. placer le fichier source sur le serveur B (qui devient alors le fichier de destination) - ligne 132
7. générer la somme md5 du fichier destination - ligne 134
8. récupérer la somme md5 du fichier destination - ligne 137
9. comparer les deux sommes md5 (res1 et res2) - ligne 143

```
107 #tester1 steps
108  tester1.addRunStep("ok",True,skip="%s starting, source file : %s"%(tester1.name,file1))
109 #1/put md5script on server A
```

```
110 tester1.addRunStep("putFile",md5scriptname,getDir(source_param['fich'])+md5scriptname)
111 tester1.addRunStep("runCommand",command="chmod +x "+getDir(source_param['fich'])+md5scriptname)
112 #2/generate md5sum of source file
113 tester1.addRunStep("runCommand",command="%s"%(md5command1))
114 tester1.addRunStep("expectFromCommand",pattern="md5done")
115 #3/get source file from server A
116 tester1.addRunStep("getFile",source_param['fich'],"source_file")
117 #4/get md5sum from server A
118 tester1.addRunStep("getFile",resultfile1,"res1")
119 tester1.addRunStep("barrier","source getted")
120 tester1.addRunStep("terminateCommand")
121 tester1.addRunStep("waitCommandTermination")
122 #we check that diff result on standard output is like "same files"
123 tester1.addRunStep("ok",True,skip="%s has finished"%tester1.name)
124
125 #tester2 steps
126 tester2.addRunStep("ok",True,skip="%s starting, destination file : %s"%(tester2.name,file2))
127 #5/put md5script on server B
128 tester2.addRunStep("putFile",md5scriptname,getDir(dest_param['fich'])+md5scriptname)
129 tester2.addRunStep("runCommand",command="chmod +x "+getDir(dest_param['fich'])+md5scriptname)
130 tester2.addRunStep("barrier","source getted")
131 #6/put source file on server B (becoming dest file)
132 tester2.addRunStep("putFile","source_file",dest_param['fich'])
133 #7/generate md5sum on server B
134 tester2.addRunStep("runCommand",command="%s"%(md5command2))
135 tester2.addRunStep("expectFromCommand",pattern="md5done")
136 #8/get md5sum from server B
137 tester2.addRunStep("getFile",resultfile2,"res2")
138 tester2.addRunStep("terminateCommand")
139 tester2.addRunStep("waitCommandTermination")
140 #9/compare res1 and res2
141 def compareFiles(dtester):
142     dtester.ok(filecmp.cmp("res1","res2"),"Compare res1 with res2")
143 tester2.addRunStep(compareFiles)
144 #tester2.nb_steps += 1
145
146 # Here begins the test
147 dtest.DTestMaster.logger.setLevel(level=logging.WARNING)
148 dtest.DTester.logger.setLevel(level=logging.WARNING)
149 dtest.SSHSessionHandler.logger.setLevel(level=logging.WARNING)
150
151 # Add some trace Handlers
152 traceManager = dtest.TraceManager()
153 # TAP goes to stdout
154 traceManager.register(dtest.TAPTraceHandler())
155 # MSC goes to file MSC-trace
156 traceManager.register(dtest.MSCTraceHandler())
157 # Promela goes to PROMELA_trace
158 traceManager.register(PromelaTraceHandler())
```

3.2 Diagramme de déploiement du script md5

La figure 2 montre le déploiement de DTest pour l'exemple du script md5.

Dans ce diagramme, le tester1 gère une session SSH avec la machine lanoux, le tester2 avec la machine aurore.cict.fr. Le DTestmaster et les DTesters sont exécutés localement comme c'est toujours le cas avec DTest.

Un autre exemple de script correspondant au test d'une simulation distribuée de billard utilisé pour tester CERTI est décrit en annexe D.

3.3 Message Sequence Chart du script md5

Le « Message Sequence Chart » de la figure 3 représente la trace d'exécution des pas de test de l'exemple du script md5. Il est généré directement par DTest à partir de son exécution.

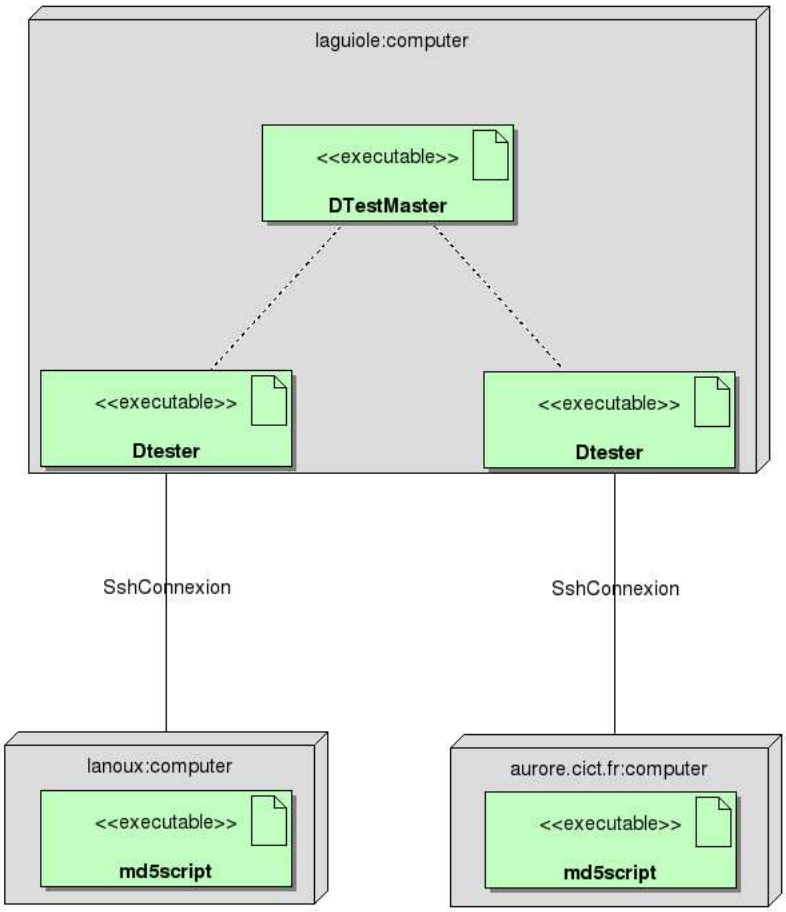


FIG. 2 – Diagramme de déploiement pour l'exemple du script md5

4 Court panorama des tests logiciels

Nous avons voulu situer DTest parmis les tests logiciels pour définir plus précisement quels types de tests il pouvait mettre en œuvre.

Dans le domaine du logiciel, l'activité de test permet de s'assurer qu'un logiciel fonctionne conformément à ses spécifications et qu'il ne produit pas de faux résultats. Plus une erreur est détectée tard, plus elle coûte cher à corriger.

On peut classer les tests de logiciels informatiques selon différents axes :
- le niveau de visibilité de la structure interne de l'objet à tester
- la nature de l'objet à tester (à mettre en relation avec le cycle de développement logiciel)
- la caractéristique de l'objet à tester

Ce chapitre à été rédigé à partir de : [8, 2, 3, 30, 1, 32].

4.1 Tests statiques / tests dynamiques

Les tests statiques (ou analyse statique [4]) désignent les techniques de tests qui ne nécessitent pas d'exécuter le logiciel. La vérification syntaxique du code et les revues de code font partie des tests statiques. L'analyse statique englobe une famille de méthodes formelles qui dérivent automatiquement

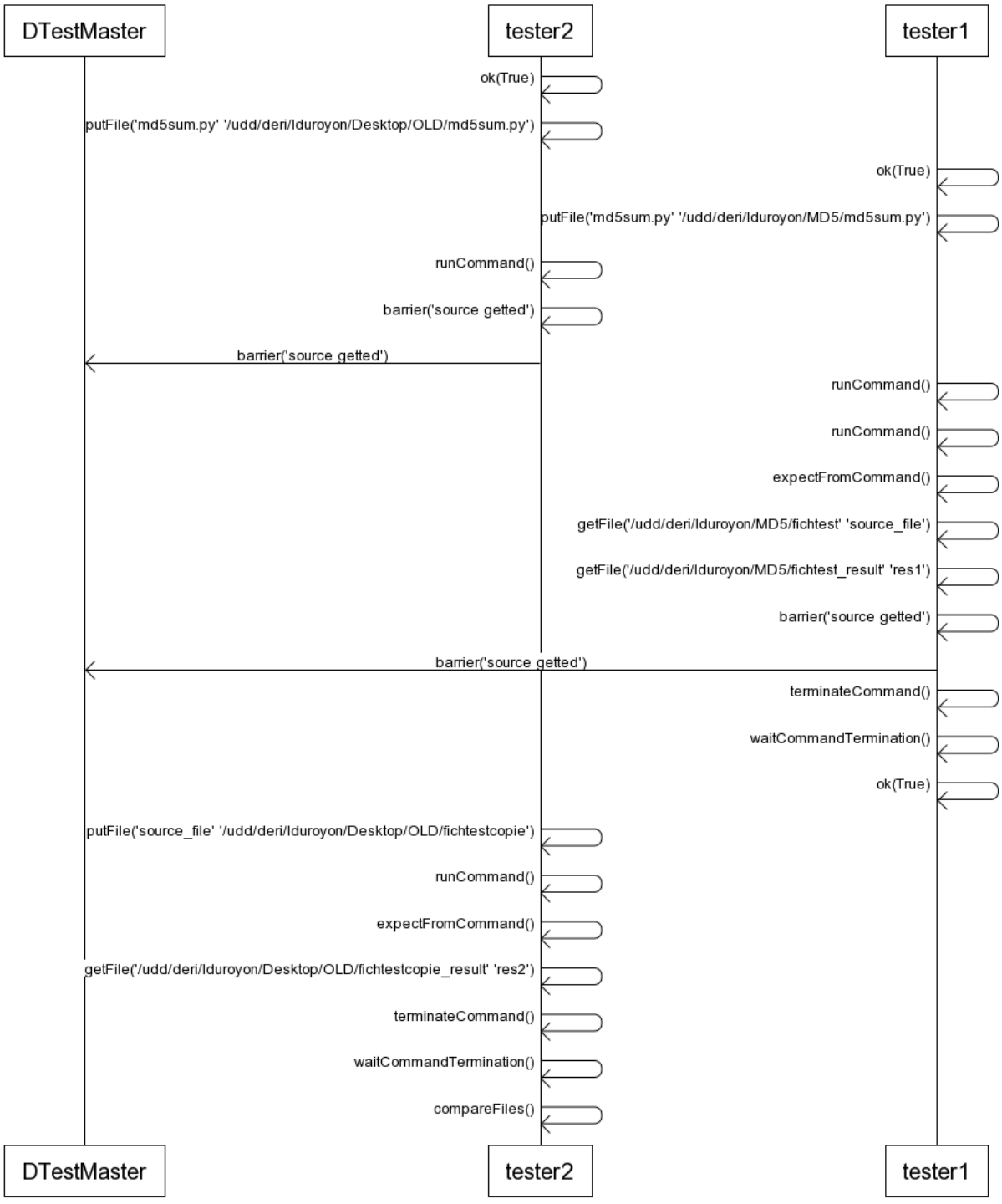


FIG. 3 – Message Sequence Chart pour l'exemple du script md5

de l'information sur le comportement de logiciels ou de matériels informatiques.

Il existe deux grandes familles d'analyses statiques formelles de programmes :

1. le model checking qui considère des systèmes à état fini, ou qui peuvent être réduits vers des systèmes à états finis par abstraction ;
2. l'analyse statique par interprétation abstraite qui approxime le comportement du système.

Les tests dynamiques, au contraire, sont élaborés en fonction de l'exécution du logiciel. Il s'agit de concevoir les jeux de tests pour les tests structurels et les tests fonctionnels. La mise en œuvre des jeux de tests est souvent empirique, on décide, en fonction de ce que l'on souhaite tester, des données d'entrée à fournir au programme et des résultats attendus.

4.2 Classification selon le niveau de connaissance de la structure de l'objet à tester

4.2.1 Tests boîtes noires

Ce sont des tests, fonctionnels ou non-fonctionnels, sans référence aux structures internes du composant ou du système. Ce type de test peut être appliqué à différents niveaux de test logiciel.

4.2.2 Tests boîtes blanches

Ces tests sont basés sur une analyse de la structure interne du composant ou système. Dans le cas du test structurel nous considèrerons l'ensemble des chemins reliant le point d'entrée au point de sortie qu'il s'agisse du graphe de contrôle comme celui du flot de données. Ce test repose sur la sélection de données d'entrée déclenchant l'exécution de certains de ces chemins.

4.2.3 Tests boîtes grises

Situés entre les tests boîtes noires et les tests boîtes blanches, il existe également des tests boîtes grises. Dans ce type de tests, le testeur a accès aux structures de données internes ainsi qu'aux algorithmes pour concevoir ses jeux de tests, mais le test s'effectue au niveau utilisateur (boîte noire). Si l'on manipule des données d'entrée et de sortie, on ne fait pas des tests boîtes grises mais des test boîtes noires car les entrées et sorties sont clairement en dehors de l'application à tester.

4.3 Classification selon la nature de l'objet à tester

Ci-dessous le cycle de développement en V dont chaque phase descendante est mise en relation avec une phase de test ascendante :

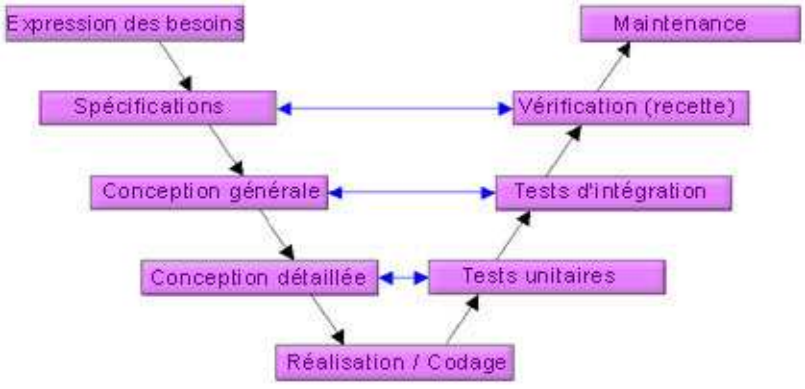


Fig. 4 – Schéma du cycle de développement en V

4.3.1 Les tests unitaires

Ils ont pour but de vérifier le bon fonctionnement d'un module indépendemment du reste du programme. Cette phase doit être particulièrement soignée car elle permet de détecter au plus tôt des erreurs, limitant ainsi leur propagation. Les tests unitaires vont pour cela tester la couverture du code, c'est-à-dire l'ensemble des instructions à tester. On rejoue l'ensemble des tests unitaires après chaque modification pour s'assurer qu'il n'y a pas de régression. Ils correspondent au test de la phase de conception détaillée du cycle en V. Le terme générique « xUnit » désigne un outil permettant de réaliser des tests unitaires dans un langage donné. [9]

4.3.2 Les tests d'intégration

Une fois chaque module logiciel testé par des tests unitaires, les tests d'intégration permettent de s'assurer que les différents modules fonctionnent bien ensemble. La démarche d'intégration la plus classique (bottom-up) consiste à intégrer module par module jusqu'à obtenir l'application complète. L'intégration continue est la fusion des tests unitaires et des tests d'intégration car le programmeur détient toute l'application sur son poste et peut donc faire de l'intégration tout au long de son développement. Les tests d'intégration sont à mettre en correspondance avec la phase de conception générale du cycle en V.

4.3.3 Les tests de validation

Le test de validation permet de vérifier si toutes les exigences client décrites dans le document de spécification d'un logiciel, écrit à partir de la spécification des besoins, sont respectées. Les tests de validation se décomposent généralement en plusieurs phases :

La validation fonctionnelle : les tests fonctionnels vérifient que les différents modules ou composants implémentent correctement les exigences client. Ces tests peuvent être de type valide, invalide, inopportuns, etc...

La validation solution : les tests solutions vérifient les exigences clients d'un point de vue "use cases", généralement ces tests sont des tests en volumes. Chaque grand use-case est validé un par un, puis tous ensemble. L'intérêt est de valider la stabilité d'une solution par rapport aux differents modules qui la composent, en soumettant cette solution à un ensemble d'actions représentatif de ce qui sera fait en production.

La validation performance, robustesse : les tests de performance vont vérifier la conformité de la solution par rapport à ses exigences de performance, alors que les tests de robutesse vont essayer de mettre en évidence des éventuels problèmes de stabilité dans le temps (fuite mémoire, résistance au pic de charge)

Ces phases de validation fonctionnelles peuvent être complétées par une phase de validation plus technique consacrée aux tests de haute disponibilité (tests cluster ou autres) [7].

4.4 Classification selon l'objectif du test :

4.4.1 Les tests de non-régression

Les tests de non-régression ont pour but de vérifier que les évolutions apportées par une nouvelle version d'un logiciel n'altèrent pas les fonctionnalités préexistantes, que ce soit de manière directe ou indirecte.

4.4.2 Les tests fonctionnels

Les tests fonctionnels sont des tests boîtes noires qui permettent de vérifier que les exigences fonctionnelles et techniques du cahier des charges sont respectées. A partir de la spécification, le testeur écrit les jeux de tests ainsi que les résultats attendus en vue de tester une certaine fonctionnalité du système. Bien que ces tests soient généralement effectués en phase de validation, ils peuvent se dérouler plus tôt, pour tester un composant par exemple.

4.4.3 Les tests passifs

Dans les tests passifs [29], il n'est pas nécessaire que l'implantation intéragisse avec le testeur. Les traces d'exécution sont observées à travers des points d'observation.

4.4.4 Les autres types de tests

Parmis les autres types de tests on peut aussi citer, sans être exhaustif :
- Les tests de conformité : vérifiant qu'un système satisfait une norme ou un standard.
- Les tests de performance : testant que les performances annoncées dans la spécification sont bien atteintes.
- Les tests de robustesse : mesurant la robustesse du logiciel.
- Les tests de vulnérabilité : vérifiant que le logiciel ne contienne pas de failles de sécurité.
- Les tests d'utilisabilité : assurant ici que l'interface utilisateur soit simple d'utilisation.

4.5 DTest parmis les tests

Après cette classification succinte, nous pouvons situer DTest au niveau :
- des tests boîtes noires : il n'a pas accés à la structure interne de l'application testée.
- des tests systèmes : il vise à tester tout ou partie d'une application distribuée.

Il permet de réaliser
- des tests passifs (ou actifs) : si le test n'utilise que des observations, le test est passif. Si des actions sont utilisées pour intéragir avec l'équipement sous test, alors le test est actif.
- des tests fonctionnels : on peut vérifier si une fonctionnalité d'une application distribuée correspond bien à la fonctionnalité définie en phase de spécification.
- des tests d'intégration : on peut facilement écrire une séquence de tests pour des applications existantes en adaptant le script aux observations disponibles (sorties standard).
- des tests de non-regréssion : on peut rejouer une séquence de tests après avoir modifié l'application afin de s'assurer que son comportement reste le même.

5 DTest et la génération de cas de test

Il existe des techniques permettant de générer automatiquement des jeux de tests à partir des spécifications. Parmis celles-ci, on peut s'intéresser à la génération de cas de tests à partir de model-checker [10, 16, 19]. Dans cette dernière, on utilise la capacité qu'ont les model-checkers pour générer des contre-exemples lorsqu'ils détectent des assertions invalides. Pour un cas de test ayant un comportement caractérisé par la propriété p, on crée un modèle de l'application sur lequel on tente de vérifier « toujours non p », s'il existe un contre-exemple généré par le model-checker, ce contre-exemple fournit la cas de test désiré.

DTest peut apporter sa contribution aux techniques de génération de tests à partir d'un modèle (au sens model-checking) d'une application. Il peut apporter :

– L'environnement d'exécution des tests, dans le cadre du processus de génération de tests. Exemple la partie test harness & drivers de la figure 5 [10] :

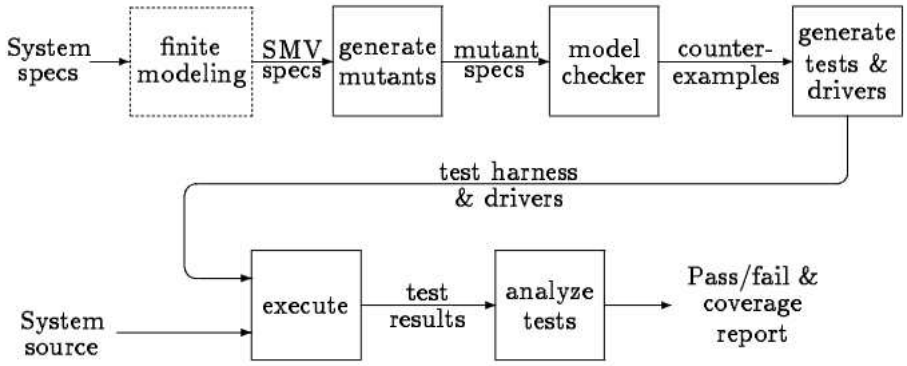


FIG. 5 – Schéma d'un processus de génération de tests

6 DTest et les autres outils de tests distribués

Avant d'approfondir le développement de DTest, nous avons voulu comparer DTest avec les autres outils de tests distribués pour savoir s'il existait des outils permettant de mettre en œuvre de façon simple des tests distribués. Nous nous sommes intéressé plus particulièrement à Test and Testing Control Notation (TTCN cf annexe B) que nous allons décrire succintement.

DTest et TTCN

L'apport principal de TTCN pour DTest aurait été d'utiliser un langage standard afin d'exprimer les séquences de tests distribués. Le langage cœur de TTCN est proche des séquences de tests pouvant être exprimées avec DTest. De plus les notions de Main Test Component et de Parallel Test Component utilisées pour tester les applications distribuées en TTCN sont proches de celles de DTestMaster et de DTester présente dans DTest. Mais le mécanisme d'implémentation de TTCN est un système global qui commence par la description de séquences abstraites de tests, puis qui est suivi par la compilation de ces séquences en code exécutable et qui se termine par l'écriture de codes et d'adaptateurs pour adapter ce code à la plate-forme d'exécution [28]. La définition de séquences de tests abstraites est généralement associée à l'utilisation d'outils commerciaux implémentant différentes plate-formes d'exécution.

DTest contraste avec les autres outils de tests distribués grâce à sa simplicité d'utilisation pour la définition et l'exécution des séquences de tests distribués. L'utilisateur peut se concentrer sur l'écriture de séquences de tests sans se soucier de l'implémentation de codecs ou autres adaptateurs. Un autre avantage de DTest par rapport aux outils ou langages permettant de faire des tests distribués est sa licence open source, adaptée aux activités de recherches sans l'empêcher d'intéresser des industriels.

7 DTest et le Model Driven Architecture

Pour finir de situer DTest, nous nous sommes intéressé à l'approche de Model Driven Architecture (MDA cf annexe C).

Le processus de test n'est en général pas intégré à l'approche MDA. Cependant, il existe des approches de génération de cas de tests en utilisant l'approche MDA [26], voir même certaines approches de générations conjointes de code et de tests [20]. DTest se positionne par rapport au Model Driven Architecture de manière similaire à la génération de tests.

1. Il propose un environnement simple pour définir et exécuter des séquences de tests distribués pouvant être utilisé pour la phase de tests de l'approche MDA.
2. Un environnement simple d'exécution de tests séparé de l'approche MDA.

DTest n'est pas contradictoire avec les approches de génération de tests et de MDA, il peut s'intégrer parmis ces dernières en proposant un environnement de définition et d'exécution simple et automatisable pour les tests, distribués ou non.

8 Vérification de propriétés

DTest a comme objectif plus large de créer un framework générique permettant l'exécution de tests distribués et faisant le lien avec des techniques de vérifications formelles, nous allons commencé par une introduction au model-checking suivie de l'explication des techniques de vérification que nous proposons.

8.1 Model-checking

Le model-checking fait partie des techniques de vérification automatique de programmes dynamiques [21]. Dans le model-checking, on cherche à vérifier un ensemble de propriétés exprimées le plus souvent en logique temporelle sur le modèle d'un programme.

Ci-dessous un schéma illustrant le principe du model-checking [27] :

Un modèle est une abstraction du programme préservant ses caractéristiques principales mais simplifiant sa complexité. Il est composé de deux parties [10] :

1. Une machine à état définie à l'aide de variables, des valeurs initiales de ces variables et de la description des conditions sous lesquelles ces variables changent de valeur.
2. Un ensemble d'invariants et de contraintes décrits en logique temporelle sur les chemins possibles d'exécution.

Une fois le modèle défini, un model-checker explore l'ensemble des états du modèle en vérifiant que les invariants et les contraintes temporelles soient respectés. Dans le cas contraire, le model-checker essaie de générer un contre-exemple sous la forme d'une séquence d'états (ce qui n'est pas toujours possible). Le model-checking est particulièrement intéressant pour vérifier des propriétés sur des systèmes concurrents comme la détection de code non exécutable ou d'interblocage, ou encore, plus généralement, toute propriété de sûreté ou de vivacité.

Il existe différents model-checker parmis lesquels NuSMV[5], SAL[6], Spin[7](une liste plus exhaustive est disponible sur [5]). Nous avons choisi d'utiliser Spin grâce à la description de la génération de code des MSC vers Promela décrite dans [23].

[5] http://nusmv.fbk.eu/NuSMV
[6] http://sal.csl.sri.com
[7] http ://spinroot.com

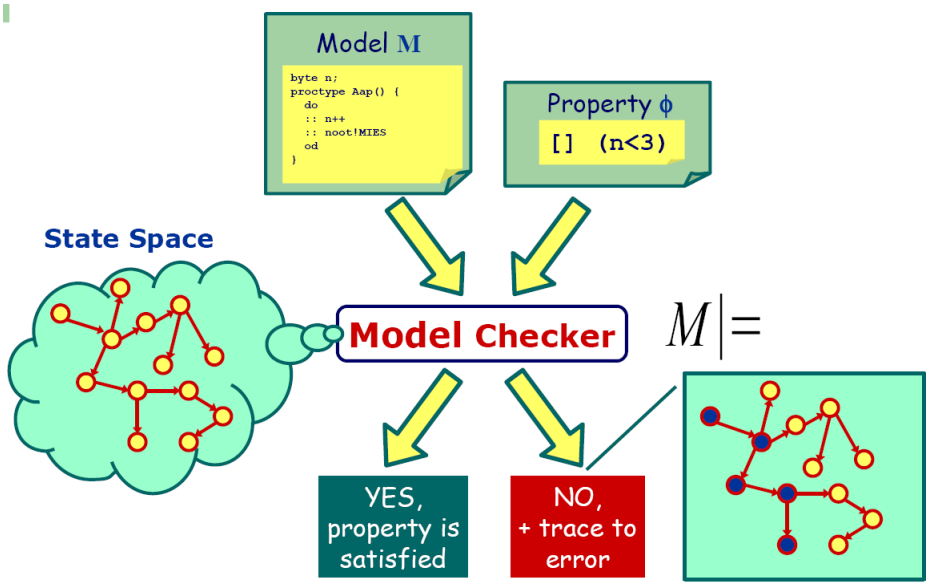


FIG. 6 – Model-Checking

8.1.1 Spin/Promela

Spin est un model-checker open-source largement utilisé. Son utilisation est associée au langage Promela. Promela permet la modélisation de systèmes concurrents et de protocoles de communication. La communication entre ces différents processus peut se faire en partageant les variables globales en utilisant des canaux de communication. Ces canaux peuvent être utilisés de manière synchrone ou asynchrone. Il n'y a pas de différences entre instructions et conditions. Une condition peut être passée si elle est vraie, une instruction si elle est exécutable. Le processus est bloqué dans le cas contraire jusqu'à ce qu'il puisse poursuivre son exécution.

8.1.2 LTL

Afin de vérifier des propriétés sur les séquences de tests, on utilise de la logique LTL. Voici un petit rappel de logique LTL.

description		symbole
opérateurs de logique temporelle	toujours	`[ ]`
	éventuellement	`<>`
	jusqu'à	`U`
	opérateur dual de jusqu'à	`V`
opérateur booléens	et	`&&`
	négation	`!`
	ou	`||`
	implication	`->`
	équivalence	`<->`
exemple de propriétés génériques	invariance	`[ ] p`
	réponse	`p -> <> q`
	précédence	`p -> (q U r)`
	but	`p-><>(q || r)`

8.1.3 Never claim

Les « Never Claim » nous permettent de vérifier des propriétés sur le système avant et après chaque instruction exécutée. Le « Never Claim » le plus simple consiste à vérifier l'invariance d'une propriété p.

```
never {
    do
    :: !p -> break
    :: else
    od
}
```

Dès que la propriété est fausse pour un état du système, le « Never Claim » se termine, ce qui indique au model-checker qu'un comportement interdit s'est produit. On peut également obtenir une violation de propriété si l'on reste indéfiniment dans un état acceptant du « Never Claim ».

Pour ce qui nous intéresse, nous allons définir un « Never Claim » qui se termine lorsqu'une séquence d'instructions a été observée (chaque instruction correspondant à une observation). On notera que les « Never Claim » sont interprétés au sein de spin uniquement en mode de vérification.

Un autre but de DTest est de vérifier des propriétés sur l'exécution de séquences de tests distribués. Dans les séquences de tests, on peut distinguer deux niveaux de vérification.

- Le premier niveau est la vérification de propriétés sur la séquence des pas de test associés aux DTesters.
- Le second niveau est la vérification de propriétés sur la séquence d'exécution elle-même.

L'approche utilisée ici contraste avec l'approche préconisée par l'ingénierie des modèles consistant à utiliser le model-checking pendant l'étape de spécification et ensuite à implémenter cette spécification.

On désire vérifier qu'une trace d'exécution satisfasse un modèle.

Pour cela l'utilisateur suit la démarche suivante :

- Il écrit un scénario de test (ici les pas de test de DTest).
- Il décrit une propriété à vérifier sur ce scénario de test.
- Il vérifie que la propriété est bien définie sur la séquence de tests (voir chapitre suivant).
- Il exécute le scénario de test obtenant ainsi une trace d'exécution.
- Il vérifie que la trace d'exécution satisfasse un modèle de l'application.

8.2 Verifier des propriétés sur les séquences de tests

Une première approche a été d'utiliser les travaux de [23] permettant d'obtenir du code Promela à partir de diagrammes MSC.

Les règles retenues ont été :

- Chaque instance MSC est mise en correspondance avec un processus Promela.
- Chaque flèche de message est représentée par un canal Promela. Ces canaux sont tous de capacité 1.
- Les messages envoyer et recevoir sont modélisés par les instructions Promela respectivement `tu!CR` et `tu?CR`.
- Les branchements sont modélisés en utilisant les instructions de branchements Goto de Promela.
- On utilise des instructions de type `full(...)->skip` pour modéliser l'envoie non bloquant d'un MSC.
- En MSC les réceptions sont par contre bloquantes, cela est implémenté en utilisant un mécanisme de garde de type `vw?[CC]->vw?CC`.

– Pour leur initialisation, tous les processus sont lancés simultanément de manière atomique

On a ensuite rajouté des règles spécifiques pour Dtest :

– Les canaux ont un nom du type `emetteurRecepteurInformation`.

– Les messages « self » correspondant aux pas de test sont modélisés par des étiquettes.

– Dans un premier temps, on suppose tous les messages de même type.

Les barrières sont implémentées à l'aide de compteurs représentant le nombre de participants ayant atteint la barrière. On teste `(nb_participants_arrives==nb_participants_attendus)` pour vérifier que tous les participants sont arrivés à la barrière (bloquant sinon).

Cette transformation de MSC à Promela permet de modéliser l'exécution de chaque testeur. On peut alors ensuite vérifier des propriétés sur l'ensemble des états possibles de la modélisation de la séquence de tests obtenue, grâce au Model-Checking. On s'assure ainsi que la séquence de tests exprime correctement les propriétés que l'on veut vérifier une fois la séquence de tests lancée.

Ci-dessous le script généré par DTest à partir de l'exemple du script md5 :

```
1  /*barrier counter declaration*/
2  byte source_getted;
3
4  /*type of message*/
5  mtype = {a};
6
7  proctype DTestMaster()
8  {
9
10         skip;
11 }
12
13 proctype tester1()
14 {
15
16         ok_0:printf("ok_0");
17
18         putFile_1:printf("putFile_1");
19
20         runCommand_2:printf("runCommand_2");
21
22         runCommand_3:printf("runCommand_3");
23
24         expectFromCommand_4:printf("expectFromCommand_4");
25
26         getFile_5:printf("getFile_5");
27
28         getFile_6:printf("getFile_6");
29
30         barrier_7:printf("barrier_7");
31
32         source_getted = source_getted + 1 ;
33
34         (source_getted==2);
35
36         terminateCommand_9:printf("terminateCommand_9");
37
38         waitCommandTermination_10:printf("waitCommandTermination_10");
39
40         ok_11:printf("ok_11");
41
42 }
43
44 proctype tester2()
45 {
46
47         ok_0:printf("ok_0");
48
49         putFile_1:printf("putFile_1");
50
51         runCommand_2:printf("runCommand_2");
```

```
52
53          barrier_3:printf("barrier_3");
54
55          source_getted = source_getted + 1 ;
56
57          (source_getted==2);
58
59          putFile_5:printf("putFile_5");
60
61          runCommand_6:printf("runCommand_6");
62
63          expectFromCommand_7:printf("expectFromCommand_7");
64
65          getFile_8:printf("getFile_8");
66
67          terminateCommand_9:printf("terminateCommand_9");
68
69          waitCommandTermination_10:printf("waitCommandTermination_10");
70
71          compareFiles_11:printf("compareFiles_11");
72
73  }
74
75  init
76  {
77          source_getted = 0;
78  atomic {
79          run DTestMaster();
80          run tester1();
81          run tester2();
82          }
83  }
```

Voici quelques exemples de propriétés que l'on peut ainsi vérifier sur la séquence d'exécution définie :

- l'exécution de la commande `getFile` précède toujours l'exécution de la commande `putFile`.
  `[ ] (getFile -> <> putFile)`
  `#define getFile tester1@getFile_6`
  `#define putFile tester2@putFile_5`
- l'exécution de la commande A implique que la barrière B a été franchie
  `[ ] (compareFiles -> barrierSourceGettedReached)`
  `#define compareFiles tester2@compareFiles_11`
  `#define barrierSourceGettedReached (source_getted==2)`

8.3 Vérifier des propriétés sur les traces d'exécution

Nous avons ensuite chercher à vérifier que la séquence d'observations issue des traces d'exécution est conforme au modèle de l'application. Nous avons pour cela repris des techniques venant des tests passifs [29]. L'approche est la suivante (cf Figure 7) :

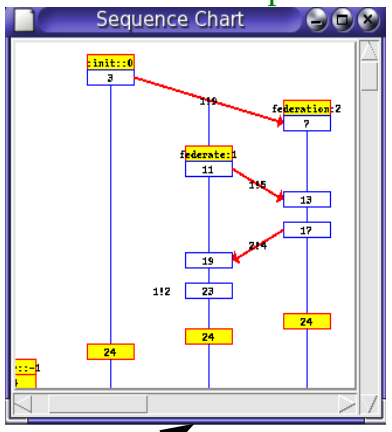
1. On récupère les observations à partir d'une trace d'exécutions
2. A l'aide des observations et d'une table de correspondance observation/modèle, on génère un automate never interdisant cette séquence d'observations (plus simple qu'une formule LTL)
3. On lance une vérification sur le modèle enrichi de cette clause never
4. Si la séquence d'observations est valide le model-checker fournit un contre-exemple
5. Sinon le model-checker ne fournit aucun contre-exemple

La limite de cette technique est la taille du modèle. Pour remedier à cela, nous aimerions faire guidée la vérification de propriétés sur des gros modèles par les observations afin de réduire l'espace d'état parcouru et obtenir ainsi un résultat en un temps acceptable.

Modèle de l'application

```
proctype federate()
{
    IDLE_FEDERATE:
        request!join;
    WAITING_JOIN_ACK:
        answer?join_ack;
    JOINED:
        printf("federate action");
        request!resign;
    WAITING_RESIGN_ACK:
        answer?resign_ack -> total=total+1;
        if
          ::(total!=MAX_FED)->goto IDLE_FEDERATE
          ::else
        fi
}
```

Contre-exemple



Séquence de tests

barrier(« start »)
expectFromCommand(« join »)
expectfromCommand(« resign »)
barrier(« end »)

Vérification

Séquence d'observations conforme au modèle

Séquence d'observations non conforme au modèle

Aucun contre-exemple

```
never {
S0:
if
::((federate@JOINED))->goto S1
::((!federate@JOINED))->goto S0
fi;
S1:
if
::((!federate@WAITING_RESIGN_ACK))->goto acceptS
::((!federate@WAITING_RESIGN_ACK))->goto S1
fi;
acceptS:
skip
}
```

join
resign

Trace d'observations

+

join/federate@JOINED
resign/federate@RESIGN

Issue du modèle

Table de correspondance observation/point du modèle

9 Conclusion

Lors de ce stage, pour la partie pratique, nous avons ajouté au framework existant de DTest :

- Un trace handler Message Sequence Charts permettant d'obtenir une visualisation graphique de l'exécution des séquences de tests
- Un trace handler permettant de générer du code Promela à partir des séquences de tests
- Un mécanisme de pseudo-exécution permettant d'associer un traitement aux séquences de tests sans exécuter les actions qu'elles contiennent

Pour ce qui est de l'automatisation des tests, DTest est actuellement utilisé pour les tests d'intégration de CERTI grâce au script de l'annexe D. Il est également utilisé par un industriel (B2I, pour un projet d'Alcatel) qui a eu connaissance de DTest lors d'une présentation à l'Onera pour tester un accélerateur HTTP pour une plate-forme multimédia par satellite.

Pour la partie exploratoire, des propositions concrètes de mise en œuvre de méthodes formelles pour la vérification de séquences de tests distribués et de validation de traces d'exécution ont été définies. Elles pourront être approfondies pour permettre de vérifier des propriétés plus complexes.

Ce stage m'a permis de découvrir le monde de la recherche avec la partie exploratoire du stage composée de lecture d'articles. J'ai notamment apprécié de pouvoir proposer des solutions puis de les mettre en œuvre pour la vérification de propriétés sur les séquences de tests. J'ai également pu approfondir mes connaissances en Python et en model-checking.

DTest peut être amélioré pour permettre de mettre en œuvre des observations et des actions plus génériques. Nous pensons qu'une perspective intéressante de DTest est de permettre de vérifier des propriétés sur de gros modèles d'applications distribuées en guidant le model-checker grâce aux observations colléctées après l'exécution des séquences de tests. On peut également penser à utiliser DTest dans le cadre du test de middleware, si ce dernier dispose de moyens d'observations adéquats.

Annexe

A Message Sequence Charts (MSC)

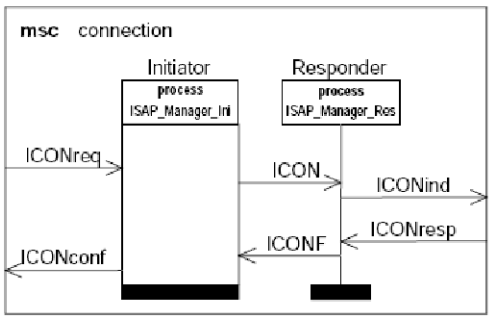
Le langage « Message Sequence Charts » (MSC), normalisé par l'International Telecommunication Union [22] fournit un langage de trace pour la spécification et la description de comportements de communication entre des composants systèmes et leur environnement par le biais d'échange de messages.

On distingue trois formes de représentations MSC :

– La première graphique, proche des diagrammes de séquences
– La deuxième orientée instance décrit les messages instance par instance
– La troisième orientée événement se focalise sur l'ordre global des événements de la séquence

Cette troisième forme est particulièrement adaptée à la description de traces d'exécution en environnement distribué.

Dans la figure 8, on peut un exemple de MSC graphique avec respectivement ses formes orientée instance et événement [25] :




```
msc connection;
inst Initiator, Responder;
  instance Initiator: process ISAP_Manager_Ini;
    in ICONreq from env;
    out ICON to Responder;
    in ICONF from Responder;
    out ICONconf to env;
  endinstance;
  instance Responder: process ISAP_Manager_Resp;
    in ICON from Initiator;
    out ICONind to env;
    in ICONresp from env;
    out ICONF to Initiator;
  endinstance;
endmsc;
```

```
msc connection;
inst Initiator, Responder;
Initiator:    instancehead process ISAP_Manager_Ini;
Responder:    instancehead process ISAP_Manager_Resp;
Initiator:    in ICONreq from env;
Initiator:    out ICON to Responder;
Responder:    in ICON from Initiator;
Responder:    out ICONind to env;
Responder:    in ICONresp from env;
Responder:    out ICONF to Initiator;
Initiator:    in ICONF from Responder;
Initiator:    out ICONconf to env;
Initiator:    endinstance;
Responder:    endinstance;
endmsc;
```

Fig. 8 – Représentation graphique et textuelle des « Message Sequence Charts »

En combinaison avec d'autres langages, il peut être utilisé pour spécifier, concevoir, simuler, tester, ou documenter des systèmes. Un domaine d'application important de MSC est la spécification de systèmes temps-réel, en particulier dans le domaine des télécommunications. MSC permet de représenter des scénarii standard d'exécution ainsi que des scénarii d'exception.

MSC est un langage pour décrire les interactions entre plusieurs instances communicantes indépendantes, ces principales propriétés sont les suivantes :

- C'est un langage de scénario décrivant l'ordre global des événements et les communications.
- Il permet d'exprimer des restrictions sur les données transmises et sur la temporisation des événements.
- MSC dispose d'un langage graphique donnant une vue d'ensemble de la communication entre différentes instances ainsi que d'un langage textuel utilisé principalement pour les échanges entre outils ou la vérification formelle.
- La description du langage est donnée en langage naturel ainsi que sous la forme d'une description formelle [24, 12].
- MSC peut être utilisé tout au long du processus d'ingénierie, de l'analyse du domaine en passant par la conception, en allant jusqu'à la phase de test. Il est utilisé de manière légèrement différente selon les phases de développement.
- MSC est largement utilisable et n'est pas réservé à un domaine spécifique d'utilisation.
- Il permet de faire de la conception structurée. Des scénarios simples décrits à partir de « Basic Message Sequence Charts » peuvent être combinés grâce à des « High Level Message Sequence Charts » pour former des spécifications plus complètes.
- MSC est souvent utilisé en combinaison avec d'autres méthodes et d'autres langages. Sa définition formelle permet une validation automatique par rapport à un modèle pouvant être décrit dans différents langages. Il peut être utilisé par exemple en combinaison avec SDL et TTCN.
- L'interprétation classique d'un scénario MSC est que l'implémentation actuelle doit au moins exhiber le comportement du scénario. Des interprétations alternatives sont également possibles. Un MSC peut par exemple être utilisé pour spécifier un comportement interdit.

B Test and Testing Control Notation (TTCN)

« Test and Testing Control Notation » (TTCN-3 [18, 14, 31]) est un langage standardisé par l'ETSI [15] de spécification et d'implémentation de tests de conformité.

Il est utilisable pour les tests « boîtes noires » des systèmes distribués [17] , notamment :

- Les systèmes de télécommunications (ISDN,ATM,GSM,UMTS)[8]
- Internet (IPv6)
- Les systèmes CORBA[9]

Ce langage permet de décrire des séquences de tests de manière relativement abstraite par rapport à la plate-forme d'implémentation. Des séquences de tests TTCN-3 peuvent être générées à partir de diagrammes MSC [13]. TTCN-3 est disponible sous forme textuelle et sous forme graphique [11] .

L'architecture de TTCN-3 est la suivante :

Les différentes étapes pour implémenter TTCN-3 sont :

1. Traduire les séquences de tests TTCN-3 en code exécutable
2. Adapter l'environnement d'exécution, le gestionnaire de test et les codecs
3. Implémenter les mécanismes de communication

Le cycle de développement qui lui est associé est le suivant :

1. Implémenter des adaptateurs dans un langage choisi
2. Implémenter des encodeurs et des décodeurs dans un langage choisi

[8] http://www.ttcn-3.org/SuccessStories.htm
[9] http://www.corba.org/

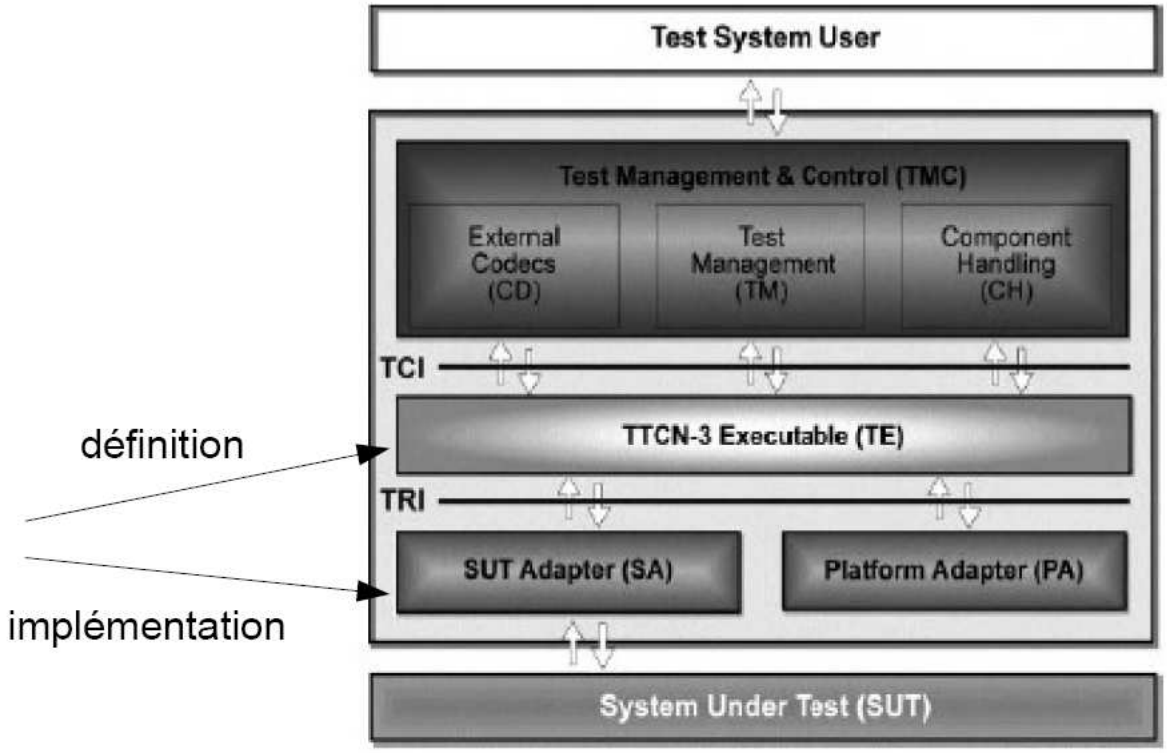


FIG. 9 – Architecture du langage Test and Testing Control Notation

3. Implémenter les cas de test en TTCN3
4. Compiler les cas de test
5. Faire le lien avec les adaptateurs, les encodeurs, etc.
6. Exécuter la suite de tests
7. Collecter les résultats et corriger le système testé en cas d'erreur

C Model Driven Architecture (MDA)

Ce standard a pour but d'apporter une nouvelle façon de concevoir des applications en séparant la logique métier de l'entreprise, de toute plate-forme technique.

En effet, la logique métier est stable et subit peu de modifications au cours du temps, contrairement à l'architecture technique. Il est donc évident de séparer les deux pour faire face à la complexification des systèmes d'information et des forts coûts de migration technologique. Cette séparation autorise alors la capitalisation du savoir logiciel et du savoir-faire de l'entreprise.

Le standard MDA doit aussi offrir la possibilité de stopper l'empilement des technologies qui nécessite de conserver des compétences particulières pour faire cohabiter des systèmes divers et variés.

Ceci est permis grâce au passage d'une approche interprétative à une approche transformationnelle. Dans l'approche interprétative, l'individu a un rôle actif dans la construction des systèmes informatiques alors que dans l'approche transformationnelle, il a un rôle simplifié et amoindri grâce à la construction automatisée.

La démarche MDA propose à terme de définir un modèle métier indépendant de toute plate-forme technique et de générer automatiquement du code vers la plate-forme choisie. Pour cela, l'accent est mis non plus sur les approches objet mais sur les approches modèle. Le but est de favoriser l'élaboration de modèles de plus haut niveau [6].

Ci-dessous un schéma simple du principe du MDA :

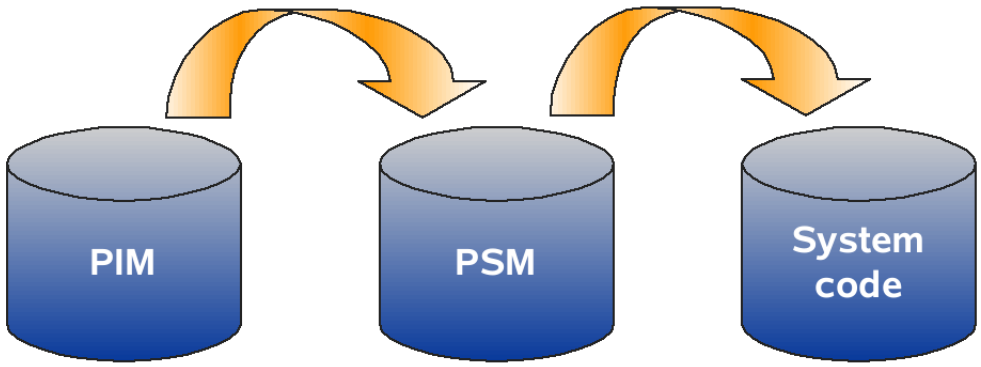


FIG. 10 – Model Driven Architecture

La démarche MDA supporte toutes les étapes de développement et standardise les passages de l'une à l'autre. Elle peut se découper en quatre points, les points 2 et 4 peuvent être répétés un nombre indéterminé de fois :

1. la réalisation d'un modèle indépendant de toute plate-forme appelé PIM pour « Platform Independent Model ».
2. l'enrichissement de ce modèle par étapes successives.
3. le choix d'une plate-forme de mise en œuvre et la génération du modèle spécifique correspondant appelé PSM pour « Platform Specific Model ».
4. le raffinement de celui-ci jusqu'à obtention d'une implantation exécutable.

Dans une démarche MDA, tout est considéré comme modèle, aussi bien les schémas que le code source ou le code binaire.

D Script de test de CERTI

```
#!/usr/bin/env python

import logging
import os
import time
import threading
import getopt, sys
import dtest
from dtest.Promela_trace_handler import PromelaTraceHandler
from dtest.Observation_trace_handler import ObservationTraceHandler

pseudoExecActive=0

def usage():
    print "Usage:\n %s [--trace] [--pseudoexec] [--certi_home=<path>]
        --rtig=<user>@[<host>]:<rtig_path> --billard=<user>@[<host>]:<billard_path>" % sys.argv[0]

def getUserHostPath(argument):
    if argument.find("@") != -1:
        (user,argument) = argument.split("@",1)
    else:
        user = os.environ["USER"]
    if argument.find(":") != -1:
        (host,path) = argument.split(":",1)
    else:
        host = "localhost"
        path = argument
    retval = dict()
    retval['user'] = user
    retval['host'] = host
    retval['path'] = path
    return retval

def createAnotherOneBillardSequence(billardTester):
    billardTester.timeout = 20
    billardTester.stdout  = file(billardTester.name + ".out",'w+')
    billardTester.stdin   = file(billardTester.name + ".in",'w+')
    billardTester.stderr  = file(billardTester.name + ".err",'w+')
    billardTester.addRunStep("barrier","RTIG started")
    billardTester.addRunStep("runCommand",command="export DISPLAY=grasse:0.0")
    billardTester.addRunStep("runCommand",command=". "+certi_home+
        "/share/scripts/myCERTIenv.sh "+rtig_param['host'])
    billardTester.addRunStep("barrier","First billard started")
    billardTester.addRunStep("runCommand",command=billard_param['path']+
        " -n \""+billardTester.name+"\" -t 10 -FTest.fed -fTest")
    billardTester.addRunStep("expectFromCommand",pattern="Synchronization")
    billardTester.addRunStep("barrier","All Billard(s) started")
    billardTester.addRunStep("expectFromCommand",pattern="Exiting.")
    billardTester.addRunStep("terminateCommand")
    billardTester.addRunStep("barrier","All Billard(s) ended")

try:
    opts, args =
        getopt.getopt(sys.argv[1:], "r:b:c:t:P", ["rtig=", "billard=","certi_home=","trace","pseudoexec"])
except getopt.GetoptError, err:
    print >> sys.stderr, "opt = %s, msg = %s" % (err.opt,err.msg)
    usage()
    sys.exit(2)

if len(opts) < 2:
    usage()
    sys.exit(2)

certi_home_defined=False

for o, a in opts:
    if o in ("--pseudoexec"):
        pseudoExecActive=1
    if o in ("--trace"):
```

```
70          traceActive=1
71      if o in ("-r", "--rtig"):
72          rtig_param    = getUserHostPath(a)
73      if o in ("-b", "--billard"):
74          billard_param = getUserHostPath(a)
75      if o in ("-c", "--certi_home"):
76          certi_home = a
77          certi_home_defined=True
78
79  if not certi_home_defined:
80      if os.environ.has_key("CERTI_HOME"):
81          certi_home=os.environ["CERTI_HOME"]
82      else:
83          print "You must define CERTI_HOME environment variable"
84          sys.exit(2)
85
86  firstBillard = dtest.DTester("firstBillard",
87      session=dtest.SSHSessionHandler(billard_param['user'],host=billard_param['host']))
88
89  rtig = dtest.DTester("rtig",
90      session=dtest.SSHSessionHandler(rtig_param['user'],host=rtig_param['host']))
91  # you may change the default time out value
92  rtig.timeout = 40
93  # you add want to save the output of your dtester to a file.
94  rtig.stdout    = file(rtig.name + ".out",'w+')
95  rtig.stdin     = file(rtig.name + ".in",'w+')
96  rtig.stderr    = file(rtig.name + ".err",'w+')
97
98  # describe RTIG run steps
99  rtig.addRunStep("ok",True,"CERTI RTIG and Billard Starts")
100 rtig.addRunStep("runCommand",command="export DISPLAY=grasse:0.0")
101 rtig.addRunStep("runCommand",command=". "+certi_home+
102     "/share/scripts/myCERTI_env.sh "+rtig_param['host'])
103 rtig.addRunStep("runCommand",command=rtig_param['path'])
104 rtig.addRunStep("expectFromCommand",pattern="CERTI RTIG up and running",timeout=5)
105 rtig.addRunStep("barrier","RTIG started")
106 rtig.addRunStep("barrier","All Billard(s) ended")
107 rtig.addRunStep("terminateCommand")
108 rtig.addRunStep("waitCommandTermination")
109 rtig.addRunStep("ok",True,"CERTI RTIG and Billard Ends")
110
111 # describe billard run steps
112 firstBillard.timeout = 20
113 firstBillard.stdout  = file(firstBillard.name + ".out",'w+')
114 firstBillard.stdin   = file(firstBillard.name + ".in",'w+')
115 firstBillard.stderr  = file(firstBillard.name + ".err",'w+')
116 firstBillard.addRunStep("barrier","RTIG started")
117 firstBillard.addRunStep("runCommand",command="export DISPLAY=grasse:0.0")
118 firstBillard.addRunStep("runCommand",command=". "+certi_home+
119     "/share/scripts/myCERTI_env.sh "+rtig_param['host'])
120 firstBillard.addRunStep("runCommand",command=
121     billard_param['path']+" -n \""+firstBillard.name+"\" -t 10 -FTest.fed -fTest")
122 firstBillard.addRunStep("expectFromCommand",pattern="Press ENTER to start execution")
123 firstBillard.addRunStep("barrier","First billard started")
124 firstBillard.addRunStep("barrier","All Billard(s) started")
125 firstBillard.addRunStep("sendToCommand",string="\n")
126 firstBillard.addRunStep("expectFromCommand",pattern="Exiting.")
127 firstBillard.addRunStep("terminateCommand")
128 firstBillard.addRunStep("barrier","All Billard(s) ended")
129
130 #instanciate more billards
131 billard2 = dtest.DTester("billard2",
132     session=dtest.SSHSessionHandler(billard_param['user'],host=billard_param['host']))
133 billard3 = dtest.DTester("billard3",
134     session=dtest.SSHSessionHandler(billard_param['user'],host=billard_param['host']))
135 billard4 = dtest.DTester("billard4",
136     session=dtest.SSHSessionHandler(billard_param['user'],host=billard_param['host']))
137
138 createAnotherOneBillardSequence(billard2)
139 createAnotherOneBillardSequence(billard3)
140 createAnotherOneBillardSequence(billard4)
141
```

```
142 # Here begins the test
143 dtest.DTestMaster.logger.setLevel(level=logging.WARNING)
144 dtest.DTester.logger.setLevel(level=logging.WARNING)
145 dtest.SSHSessionHandler.logger.setLevel(level=logging.WARNING)
146
147 # Add some trace Handlers
148 traceManager = dtest.TraceManager()
149 # TAP goes to stdout
150 traceManager.register(dtest.TAPTraceHandler())
151 # MSC goes to file MSC-trace
152 traceManager.register(dtest.MSCTraceHandler())
153 # Promela goes to PROMELA_trace
154 traceManager.register(PromelaTraceHandler())
155 # Observations goes to Observation_trace
156 traceManager.register(ObservationTraceHandler())
157
158 def goTest():
159     myDTestMaster = dtest.DTestMaster()
160     myDTestMaster.registerTraceManager(traceManager)
161     if (pseudoExecActive):
162         myDTestMaster.pseudoexec=1
163     myDTestMaster.timeout = 40
164     myDTestMaster.register(rtig)
165     myDTestMaster.register(firstBillard)
166     myDTestMaster.register(billard2)
167     myDTestMaster.register(billard3)
168     myDTestMaster.startTestSequence()
169     myDTestMaster.waitTestSequenceEnd()
170
171 goTest()
```

Références

[1] Catalogue d'outils de tests open source. http://www.opensourcetesting.org/.

[2] Etat de l'art et apport relatif des différentes techniques de test du logiciel. http://www.lri.fr/~marre/PS/test_aero.ps.gz.

[3] Génie logiciel - erreur et processus de test. http://www.infeig.unige.ch/support/se/lect/prg/tst/web.html.

[4] Wikipédia - analyse statique. http://fr.wikipedia.org/wiki/Analyse_statique_de_programmes.

[5] Wikipédia - model checking. http://en.wikipedia.org/wiki/Model_checking.

[6] Wikipédia - model driven architecture. http://fr.wikipedia.org/wiki/Model_driven_architecture.

[7] Wikipédia - tests de validation. http://fr.wikipedia.org/wiki/Test_de_validation.

[8] Wikipédia - tests logiciels. http://fr.wikipedia.org/wiki/Test_(informatique).

[9] Wikipédia - tests unitaires. http://fr.wikipedia.org/wiki/Test_unitaire.

[10] Paul Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *ICFEM*, page 46, 1998.

[11] P. Baker, E. Rudolph, and I. Schieferdecker. Graphical Test Specification-The Graphical Format of TTCN-3. *SDL*, 2001.

[12] JMH Cobben, A. Engels, S. Mauw, and M. Reniers. A. Formal Semantics of Message Sequence Charts (ITU-T Recommendation Z. 120 Annex B), 1998.

[13] M. Ebner. TTCN-3 Test Case Generation from Message Sequence Charts. *Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04 : WITUL)*, 2004.

[14] ETSI. About TTCN3.

[15] ES ETSI. 201 873-1 : The Testing and Test Control Notation version 3 ; Part 1 : TTCN-3 Core Language. 2001.

[16] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999.

[17] J. Grabowski. TTCN-3-A new Test Specification Language for Black-Box Testing of Distributed Systems. *Proceedings of the" 17th International Conference and Exposition on Testing Computer Software (TCS'2000), Theme : Testing Technology vs. Testers' Requirements", Washington DC, June*, 2000.

[18] J. Grabowski and A. Ulrich. An Introduction to TTCN-3. Tutorial. *Proceedings of The TTCN-3 User Conference at the European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), May*, 2004.

[19] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 261–270, 2004.

[20] Hajo Eichler Hideto Ogawa. Mda based approach for generation of ttcn-3 test specifications.

[21] G.J. Holzmann et al. *The spin model checker*. Addison-Wesley, 2003.

[22] I. ITU. Recommendation Z. 120. *Message Sequence Charts (MSC'96)*, 1996.

[23] S. Leue and P.B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. *Proceedings of the 2nd Workshop on the SPIN Verification System, Rutgers University, August*, 1996.

[24] S. Mauw. The formalization of Message Sequence Charts. *Computer Networks and ISDN Systems*.

[25] E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 1996.

[26] M. Rutherford and A. Wolf. A case for test-code generation in model-driven systems, 2003.

[27] Theo Ruys. Concurrent and distributed programming spin tutorial part 1. Technical report, University of Twente Department of Computer Science Formal Methods & Tools, 2000.

[28] I. Schieferdecker and T. Vassiliou-Gioles. Realizing distributed TTCN-3 test systems with TCI. *TestCom*.

[29] Ana Rosa Cavalli (Institut TELECOM / TELECOM & Management SudParis). Techniques de monitorage pour la vérification de propriétés. Technical report, 2008. http://www.cert.fr/francais/deri/michel/FAC/2008/Papiers/02pres.pdf.

[30] Yves Le Traon. Le test des logiciels. www.irisa.fr/triskell/perso_pro/yletraon/cours/Maitrise-ADT/Cours1CompactTest.pdf.

[31] LE NOURS Tristan. *Présentation d'un langage de test : TTCN-3*.

[32] T. Walter, I. Schieferdecker, and J. Grabowski. Test Architectures for Distributed Systems : State of the Art and Beyond. *Proceedings of the IFIP TC6 11th International Workshop on Testing Communicating Systems*, 1998.