Deutsch-Französisches Hochschulinstitut
Institut supérieur franco-allemand de techniques, d'économie et de sciences

Université Paul-Verlaine - Metz

Hochschule für Technik und Wirtschaft des Saarlandes
University of Applied Science

Mémoire de Licence

Design, développement et intégration d'un outil pour des tests automatisés d'interfaces web

Rolf SCHRÖDER

Responsable pédagogique	:	**Prof. Dr. Reinhard BROCKS**
Maître de stage	:	**François GOMBAULT**
Entreprise	:	**Anevia S.A.**

Septembre 30, 2011

Remerciements

Tout d'abord, je remercie ma famille et mes amis. Ils m'ont montré leur intérêt pour un sujet qu'ils connaissaient rarement.

Je tiens à remercier Prof. Dr. Reinhard BROCKS pour la prise en charge de ce mémoire et pour ses conseils.

Ce travail est le fruit d'un stage effectué à Anevia. Je remercie toute l'équipe de l'entreprise pour les bons moments passés ensemble et l'aide apporté. Un grand merci soit rendu surtout à François GOMBAULT et Florent AUDEBERT.

Déclaration sur l'honneur

Je soussigné, Rolf SCHRÖDER, certifie qu'il s'agit d'un travail original et que toutes les sources utilisées ont été indiquées dans leur totalité. Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine. À ma connaissance, ce travail n'était jusqu'à présent jamais rendu sous cette ou une forme semblable à un jury d'examen.

Fait à Gentilly, le 30 Septembre 2011

Table des matières

Chapitre 1

Introduction

Un interface web est souvent utilisé pour accéder un dispositif à distance. L'explication évidente est que l'appareil se trouve rarement juste à côté de l'utilisateur et/ou que la machine ne dispose tout simplement pas d'écran pour afficher des informations. Depuis des années, ces dispositifs sont alors équipés d'un serveur web afin de les rendre accessible par un navigateur. Aujourd'hui avec l'importance croissante d'Internet et du *cloud computing*, plus en plus de logiciels ont également tendance à utiliser le navigateur comme interface homme machine. Le développent du marché mobile a fait en sorte que des plate-formes différents émergent et qu'on souhaite que le même logiciel soit utilisable à des endroits bien différents (maison, bureau, en public). Or, la représentation visuelle à travers le web est indépendant du système d'exploitation (voir du système graphique) et se prête ainsi très bien pour créer des interfaces reconnaissables.

Les sites des réseaux sociaux sont des signes d'annonciateur d'un futur dans lequel on ne fait plus la différence entre application locale et site web. Le système *ChromeOS* de Google (un système d'exploitation disposant comme seule application un navigateur) témoigne également de l'effondrement des frontières. Le nouveau standard HTML5 (prévu pour 2014) autorise aux sites web un accès plus direct à l'ordinateur (accès ressource hardware audio/video etc.) et leur donne ainsi la même puissance que toute autre logiciel [7]. Il est donc tout à fait normal que de plus en plus d'applications se basent sur le web.

Le volume de ces « logiciels du web » ne cesse de croître. Le besoin d'assurance de qualité augmente proportionnellement avec la complexité d'un logiciel. Les tests de fonctionnalités deviennent désormais très importants. De ce point de vue, le basculement vers des applications web est plutôt favorable : l'accès libre au code source d'une page web est une qualité intrinsèque. Chacun peut analyser la structure de l'interface graphique et le développement d'outils de programmation et beaucoup plus simple.

Un test de fonctionnalité doit prendre en compte toutes les actions susceptibles d'être effectuée par l'utilisateur. La procédure de *testing* devient aussi complexe que le logiciel et demande beaucoup d'engagement. Par expérience on sait que les tests sont souvent lésés lors du développement. Un programmeur ne peut aujourd'hui guère estimer l'ampleur de ses changements dans le code. Pour ne pas prendre de risques, il devrait faire des tests étendus à chaque changement. La question de l'automatisation se pose très vite. Il existe déjà plusieurs *frameworks* (Selenium, Windmill, Watir) pour enregistrer et re-exécuter les interaction d'un utilisateur avec le site web. Pour des raisons économique et qualitatives,

quasiment chaque entreprise va faire avancer l'automatisation de tests de ses produits (informatiques) aujourd'hui.

L'assurance de qualité et l'automatisation de tests sont à la base de ce mémoire. Le travail s'est effectué lors d'un stage entreprise et il y a donc un rapport très pratique au problème. Le but concret du stage consistait à augmenter le degré d'automatisation de *testing* par la génération des cas de test. Étant donné que l'entreprise dispose déjà d'une infrastructure de test, ses besoins sont très concrets.

Après la présentation de l'entreprise, l'environnement de test existant est illustré. La connaissance des différents logiciels employés et de leurs interactions est nécessaire pour comprendre le travail effectué. Suivent des explications des approches choisies et d'autres considérations préliminaires avant les détails sur l'implémentation. Une résumé et un conclut ce mémoire.

1.1 Anevia

Le stage s'est déroulé chez Anevia S.A., entreprise active sur le marché de la vidéo sur IP. IPTV [1] permet la distribution de flux multimédia sur un réseau local (LAN). Une gamme de produits est destiné aux opérateurs de télécommunications, aux fournisseurs de services par satellites et aux fournisseurs d'accès à Internet (FAI). Une deuxième gamme est dédié au marché d'hospitalité (hôpitaux, hôtels). Les solutions proposées permettent de diffuser sur le réseau local à la fois la télévision en direct à la fois des contenus enregistrés. Ses passerelles DVB vers IP (« Flamingo » et « ViaLive ») récupèrent le signal satellite (DVB-S), terrestre (DVB-T) ou d'un câble (DVB-C) et diffusent le flux. Ils peuvent également assurer le décryptage des chaînes. Anevia offre également des serveurs Video on Demand (« Toucan » ou « ViaDemandServer ») qui diffusent les contenus sur leurs disque durs. Ces serveurs peuvent s'abonner (et enregistrer) les flux provenant des passerelles DVB vers IP. La combinaison des deux types de produits permet une variété de service comme la diffusion des chaînes payantes ou des programmes enregistrés auparavant. Les flux peuvent également être généré à partir des films, de publicités ou d'autre contenus. Les ordinateurs, les set-top boxes, les téléviseurs modernes et toute autre appareil avec un accès réseau peut profiter des services. Il existe également des fonctionnalités de conversion du contenu, de la surveillance et de l'équilibrage des charges. Les systèmes peuvent installés en redondance ce qui rend les produits susceptible d'un déploiement professionnel. Les produits sont en général installés par des intégrateurs, mais l'entreprise offre également des formations pour les futurs utilisateurs. Bien que d'origine française, Anevia vend ses produits aujourd'hui partout dans le monde. L'entreprise compte environ 50 employés.

1. Internet Protocol Television

Chapitre 2

Présentation de l'infrastructure

Ce chapitre illustre l'infrastructure existante dans l'entreprise. Lors de la création et de l'exécution de tests, plusieurs logiciels différents, dont les plus importants sont décrits ici, interviennent. TTCN-3 est introduit brièvement afin de mieux comprendre le système de test Testerman. Une comparaison par exemples est proposée. La description du processus d'exécution d'une campagne de test conclura ce chapitre.

2.1 Selenium

Selenium est un kit pour la création et l'exécution de tests unitaires. Il s'agit d'outils pour l'automatisation des tests d'interfaces web. Le *framework* regroupe plusieurs composants logiciels qui ont été développés indépendamment à la base. Ses développeurs proviennent de beaucoup d'organisations différents.

La création d'un test se fait essentiellement à l'aide de **Selenium IDE**, une extension Firefox permettant d'enregistrer des interactions de l'utilisateur avec un site web. Le logiciel est capable de repérer les éléments HTML d'un site web via leurs attributs (comme *id* ou *name*) ou par XPath [1]. Même l'interaction en dehors de l'interface web proprement dit – comme des boîte de dialogues – est enregistré. Selenium IDE enregistre toutes les actions utilisateur dans un format très simple. Chaque *selenese* a un nom et éventuellement un ou deux arguments. L'utilisateur peut exporter la suite des selenese vers des formats différents (voir ci-dessous).

Selenium RC à son tour permet piloter un navigateur. Il se présente comme serveur auquel on peut transmettre des actions à exécuter. Selenium RC communique par la suite avec le navigateur et « agit » comme un utilisateur humain en simulant ses interactions (clavier, souris, temps d'attente, etc.) avec l'interface web. Tous les navigateurs populaires (Firefox, IE, Opera, Safari, ...) sont supportés. Le logiciel peut également demander au navigateur de lui fournir certaines informations sur l'état des éléments présent sur le site web. Il existe plusieurs méthodes pour transmettre les commandes (qu'il va rediriger vers le navigateur) au serveur. Selenium met à disposition des « pilotes » : ce sont des bibliothèques écrites en plusieurs langages de programmation courants (Java, Python, Ruby et C#). En utilisant les fonctionnalités fournies, on peut écrire son propre programme pour

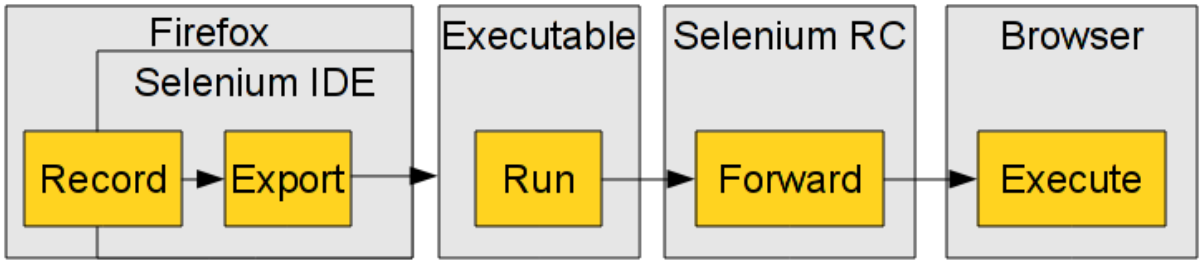
1. XPath est un langage permettant d'accéder aux nœuds d'un document XML

se connecter à Selenium RC. Ces pilotes utilisent un protocole HTTP pour communiquer au serveur Selenium RC et on pourrait donc même envisager d'envoyer les commandes « en clair » , c'est-à-dire sans passer par les pilotes (voir la documentation sur [8]).

Il existe un troisième logiciel : **Selenium Grid**. Ce programme permet de gérer plusieurs instances de Selenium RC afin de rendre l'exécution des test plus performante. Avec Selenium Grid le testeur peut lancer une instance de Selenium RC pour chaque navigateur (p.ex.) tout en utilisant le même test. Ceci permet de faire passer le même test dans une multitude d'environnements sans pour autant le relancer chaque fois. Le grid n'est pas important dans le cadre de mon projet.

Chaque outil Selenium est déjà assez puissant mais c'est en les liant qu'on met en place une vraie installation d'environnement de test. L'association Selenium IDE avec Selenium RC se fait avec des **formateurs**. Un formateur traduit la liste des actions utilisateurs (enregistrées par Selenium IDE) vers un langage (de programmation) quelconque. Le code source généré peut être compilé directement. Une fois exécuté ce nouveau programme fait appel aux pilotes Selenium afin de communiquer avec le navigateur par le biais de Selenium RC. Selenium IDE peut faire appel aux formateurs pour l'exportation des selenese. L'équipe de développement Selenium fournit déjà des formateurs pour les langages supporté par leurs pilotes. (Le fonctionnement d'un formateur est expliqué ultérieurement).

FIGURE 2.1: Chaîne d'interactions



(enregistrement avec Selenium IDE et export avec un formateur ; lancement du programme généré ; transmission des commandes par Selenium RC au navigateur, réexécution)

2.2 TestLink

TestLink est un logiciel de management de test. Il met à disposition une base de données contenant une multitude d'objets : des cas de tests, des suite de tests, des produits (à tester), des spécifications, des exigences fonctionnelles (*requirements*) etc. Un cas de test contient en outre les actions à exécuter et les résultats attendus. Le testeur peut également enregistrer le résultat d'une exécution d'un test lors d'une campagne de test. TestLink a une API (et prévoit des bibliothèques) qui permet d'accéder la base de données avec des langages de programmation différents. Ainsi, on peut automatiser une campagne de tests : un programme peut aller chercher les tests prévu pour un certain produit, les exécuter (s'ils sont automatisés) et peupler la base de données avec les résultats obtenus.

TestLink est un logiciel libre (publie sous la licence GPL) principalement basé sur PHP et MySQL.

2.3 TTCN-3

La *Testing and Test Control Notation version 3* est un langage de test développé par l'European Telecommunications Standards Institute (ETSI). La notation ressemble à des langages de programmation et elle est spécialement dédiée au développement de tests de protocoles (de communication). TTCN-3 prend une approche très universelle en se libérant complètement du système sous test (*System Under Test*, SUT). Le même cas de test peut être exécuté dans des environnements bien différents. Des adaptateurs servent d'intermédiaire entre le script de test abstrait et le SUT. Quelques caractéristiques principales sont : un système de typage fort, définition des test concurrents (des tests avec plusieurs composants), support d'une communication synchrone ou asynchrone, support de *timers* et une facilité d'exécution automatique. L'industrie a approuvé l'utilisabilité du standard. Il a été employé pour le développement de tests de conformité dans des domaines différents comme VoIP, IPv6, Digital Mobile Radio, WiMax, LTE, etc. TTCN-3 est surtout utilisé dans le domaine de la communication mais peut être adopté pour les tests unitaires (logiciels) ou les systèmes distribués. Il existe plusieurs outils open source et commerciaux autour de TTCN-3.

2.4 Testerman

Suivant les définitions de TTCN-3, on considérera Testerman comme un *test system*. C'est donc à priori un environnement capable d'exécuter des tests TTCN-3. Bien que Testerman est une plate-forme inspirée par TTCN-3, il n'execute pas ses cas de test abstraits. L'environnent essaie plutôt d'apporter les concepts de TTCN-3 au langage de programmation **Python**. Testerman n'est donc pas conforme à la norme. En dehors du simple fait qu'un cas de test (ou un module) est écrit un Python, toutes les notions de la notation ne sont pas prise en compte. Il existe bien entendu beaucoup d'expressions prédéfinies (comme `alt`), les timers, les ports, les *behaviours*, les composants principaux/parallèles etc. D'autres notions de TTCN-3 sont mis à disposition tout simplement par Python (`if`/`else` etc.). Mais certaines (comme les *altsteps* ou les *snaphots*) manquent. La prise en compte des types est beaucoup moins stricte. En effet, on peut *mapper* un port à n'importe quel d'autre sans que les types de messages correspondent. En plus, il ne faut pas forcement prédéfinir les types de messages envoyé ou reçu. Le message contient n'importe quel objet ou type (Python). Néanmoins, pour une meilleure factorisation, il est évidemment possible de définir un *template* (les dictionnaires de Python s'y prêtent très bien).

Testerman est très modulaire. Avec ce logiciel on peut créer, exécuter (et analyser) n'importe quel cas de test. L'environnent ne se focalise pas sur un domaine spécifique mais met plutôt à disposition l'infrastructure de base. Les termes *probe* et *codecs* sont introduit afin de communiquer de façon très flexible avec les systèmes sous test. En s'appuyant sur la terminologie TTCN-3, une **probe** est ce qu'on appelle un adaptateur. C'est donc un outil qui permet au test (abstrait) de communiquer avec le SUT (concret). Elle se place « derrière » un *system interface port*. Le programmeur de test ne l'utilise pas directement, il ne communique qu'avec les ports. Pour cela, il faut qu'il initialise la liaison entre la probe et le port, appelé *binding* en Testerman. Le binding ce fait avant la partie contrôle

d'un module. Il existe des probes pour la communication via SSH, la connexion vers des base de données (MySQL, Oracle), l'accès à un annuaire LDAP et également une probe pour communiquer avec Selenium RC. D'un autre côte, un **codec** donne la possibilité d'encoder/décoder des messages (notion également introduite par TTCN-3). En fait, il se peut que la probe n'implémente que la communication bas niveau avec le SUT. Pour rendre l'écriture des tests plus simple, un ou même plusieurs codes peuvent transformer le message avant/après la communication avec le système via la probe. Existent des codecs pour des protocoles comme RTSP [2] ou HTTP. Même un code XML est accessible. Chacun peut rajouter ses propres probes ou codecs (des API existent) et l'accès à (et le test de) n'importe quel système devient ainsi possible.

Testerman se présente comme serveur auquel des clients peuvent se connecter. Une fois le client connecté, il peut par exemple lancer l'exécution d'un test, voir les fichiers de logs ou partager les fichiers ats (contenant les tests) avec d'autre clients. Le client **QTesterman**, fourni avec Testerman, est une application graphique utilisant le framework Qt. Elle permet non seulement d'éditer les cas de tests et de les éxecuter, mais aussi de visualiser le résultat et en faire des rapports. Ce client est la solution la plus intuitive pour se faire une idée du système en général. QTesterman se prête également assez bien pour analyser les résultats d'une exécution d'un test ou de toute une campagne.

Testerman est un logiciel libre publié sous la licence GPLv2. Ce sont surtout des entreprises d'orientation télécommunication qui l'utilisent [3]. Anevia a aidé à améliorer Testerman.

2.5 Structure d'un module de test

Un ou plusieurs test sont en générale regroupés dans un **module**. La définition – selon le standard TTCN-3 – d'un tel module sera brièvement présentée dans cette section. Les modules en Testerman suivent cette organisation prévue plus ou moins. Des modules exemplaire en TTCN-3 et en Testerman illustrent les différences et les point communs.

2.5.1 Un module TTCN-3

En TTCN-3, le mot-clé `module` introduit la définition d'un plusieurs ou test(s), de paramètres, de fonctions et d'autres objets. Un module doit être indépendant, c'est-à-dire exécutable sans dépendances (d'autres modules par exemple). La partie dite contrôle du module peut être considérée comme le programme principal : elle définit l'ordre dans laquelle les cas de test sont exécutés. Il est possible de faire dépendre l'exécution d'un test du verdict (du résultat) d'un test précédent. L'appel d'un cas de test durant la partie contrôle est indépendant du module lui-même. Le verdict est retourné par le système de test et non pas par le cas de test directement. Quant au test, il doit être indépendant aussi et devrait laisser le SUT dans un état sain.

2. Real Time Streaming Protocol, protocol de communication des flux multimédia
3. information issue d'un échange avec Sébastien Lefèvre, développeur de Testerman

Listing 2.1: Structure générale d'un module TTCN-3

```
module MyModule {
  modulepar { // defintion for module parameters
    float x := 3;
  }

  import from OtherModule all; // imports from other modules
  import from YetAnotherModule { // selective import
    type MyType;
  }
  //records
  type record MyRecordType{
  ...
  }
  // template(s)
  template MyRecordType MyTemplate := {
    ...
  }
  // test case definition(s)
  testcase MyTestCase ... {
  }
  //behavior definition(s)
  function MyFunction ... {
    ...
  }
  // Other definitions for testcases, groups, altsteps, ...

  control {
    var verdicttype verdict := none;
    verdict = execute(MyTestCase(MyTemplate));
    if (verdict == pass) {
      ...
    }
  }
}
```

Un (cas de) test est un bout de code abstrait ressemblant à une fonction : il peut avoir de paramètres (des arguments) et retourne un résultat. Le test communique avec le SUT ou d'autres composants à travers des **ports**. Un port est une sorte de canal de communication pour échanger des données. La communication est soit basée sur l'envoi/la réception asynchrone de messages (*message-based*), soit basée sur l'appel de procédures d'objets distants (*procedure-based*). Chaque port du test est *mappé* avec un port du *Test System Interface* (un message envoyé au port [du test] arrive alors au port du TSI). Cet interface représente une vue générale sur le SUT. Un adaptateur sert d'intermédiaire entre l'interface et un système sous test concrète. Le principe des ports permet d'abstraire le cas de test. En effet, plusieurs étapes « traduisent » le message du test pour qu'il puisse arriver à plusieurs systèmes concrets (bien entendu avec le même Test System Interface et un adaptateur pour chaque système). Le test ne communique qu'avec ses ports et le système de test se charge du reste.

Comme déjà évoqué, le résultat d'un test est son **verdict**. Le verdict nous dit si le test a échoué ou pas. TTCN-3 définie plusieurs verdicts possibles : *pass* (le SUT a réagi comme attendu), *inconc* (il n'est pas sur si le test est passé ou pas), *fail* (le SUT n'a pas passé le

test), *error* (une erreur est apparue lors de l'exécution) ou *none* (la valeur par défaut du verdict avant toute assignation). Le verdict peut être modifié durant l'exécution, néanmoins une sorte de hiérarchie entre les verdicts existe. Un test échoué (failed) ne peut plus changer son verdict en pass ou inconc. Le verdict peut être récupéré par la partie contrôle du module et influencer l'exécution du prochain test.

2.5.2 Un module Testerman

En Testerman, un module peut être découpé en quatre segments :

1. un bloc de méta-données
2. une ou plusieurs classe(s) représentant le ou les tests
3. la configuration de l'adapter de test
4. la partie contrôle

La première partie contient des informations diverses comme la description du test. La définition de paramètres pour ce module est le plus important dans cette partie. Comparable à TTCN-3 (mot-clés `modulepar`), il existe une syntaxe pour définir certaines variables et leurs valeurs par défaut. Lors du l'exécution le test system, en occurrence Testerman, peut leur affecter d'autres valeurs. Il est important de savoir que ces paramètres sont modifiables par le test durant l'exécution (non conforme à TTCN-3). Ces variables peuvent par exemple être des informations d'authentification par défaut. Cette approche permet au test de rester indépendant du SUT. Toutes les métadonnées sont enregistrées sous un format XML et « commentées » afin que le parser de Python les ignore. La section 4.6.7 explique le système de ces paramètres globaux en détail.

Chaque test en Testerman doit être défini dans une classe héritant de la classe `TestCase`. La méthode `body()` de cette classe contiendra le test proprement dit, c'est à dire les actions à exécuter, la rendu du verdict etc. On se sert du langage Python pour écrire des fonctions, des conditions et d'autres aspect du déroulement du test. Testerman fournit presque toute fonctionnalité propre à TTCN-3 (le *port mapping*, la communication avec les ports, les branches, ...). Il est possible de définir plusieurs tests et de subordonner leur exécution à des conditions (voir la partie contrôle). Testerman implémente évidemment tous les verdicts définis par TTCN-3

La troisième partie se consacre à la liaison des entre *test system interface ports* et une *probe*. Une probe est un *Test Adapter* ou un *SUT Adapter* selon la terminologie TTCN-3. Il s'agit maintenant de communiquer à Testerman, qu'on voudrait faire lier un test system interface port (référencé par son nom) avec une probe. Quand le port reçoit un message, Testerman sait maintenant comment le traiter : Il est transmis à la probe configurée pour ce port.

À la fin, la partie contrôle « démarre » l'exécution du test. Comparable à TTCN-3, l'ordre de l'exécution des test est définie ici. On peut également exécuter un test selon le résultat d'un autre test préalable.

Voici la structure exemplaire d'un module en Testman :

Listing 2.2: Structure générale d'un fichier ats

```
# __METADATA__BEGIN__
# <?xml version="1.0" encoding="utf-8" ?>
# <metadata version="1.0">
# <description>description for test case MY_TEST_CASE</description>
# <prerequisites>prerequisites</prerequisites>
# <parameters>
...
# </parameters>
# </metadata>
# __METADATA__END__

#imports, functions, constants, templates

class TC_MY_TEST_CASE(TestCase):

  def body(self):
    ...
    p = self.mtc['port']
    port_map(p, self.system['port_probe'])

    p.send()
    alt([
      [p.RECEIVE(),
        lambda: setverdict(PASS),
      ],
    ])

    ...


##
# Test Adapter Configurations
##
conf = TestAdapterConfiguration('local_config')
conf.bind('port_probe', 'probe:myprobe', 'myprobe')

##
# Control definition
##
with_test_adapter_configuration('local_config')
verdict = TC_MY_TEST_CASE().execute()
if (verdict == PASS):
  # execute other test cases ...
```

La fonction `with_test_adapter_configuration()` permet ici de « publier » les liaisons entre les probes et les ports. Selon les cas de tests, celles-ci peuvent changer. Avant l'exécution du test, on demande à Testerman de fixer les liaisons jusqu'à présent seulement déclarées.

2.6 Déroulement d'une exécution d'un test

Anevia emploie Testerman comme environnement de test. Une installation TestLink sert à la définition des cas de test pour tous les produits de l'entreprise. Afin de tester un certain firmware, une campagne de test (une suite de plusieurs cas de test) est lancée.

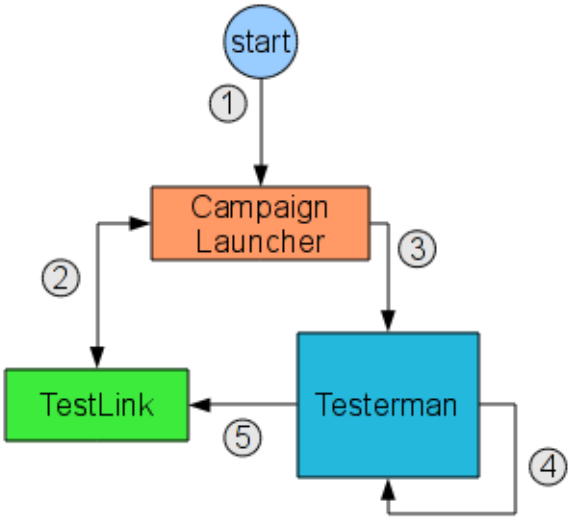
L'exécution d'une campagne de test se décompose en des parties plus ou moins complexes. Les étapes de base sont :

1. L'identification du système sous test
2. La récupération de la liste des tests à exécuter
3. La préparation de la campagne de test
4. L'exécution séquentielle de chaque cas de test
5. L'enregistrement des résultats (optionnel)

Plusieurs logiciels interagissent tout au long du processus. Le tout est contrôlé par un script appelé `launch_testlink_campaign.py`. Pour déclencher la campagne, on fait appel à ce programme en lui donnant l'adresse IP du système sous test et la version du produit (quel *firmware* doit être testé). Par la suite, la base de donnée de TestLink est interrogé afin de récupérer la liste des tests. Pour l'exécution automatisée d'un test, il ne faut pas chercher beaucoup d'informations dans la base. Il suffit de connaître son identification puisque c'est le script de test (local) correspondant qui « sait » quoi faire. De façon générale, on trouve un seul test dans un fichier ats (abstract test suite = un module TTCN-3). Beaucoup de tests sont valables pour plusieurs produits d'Anevia. Une campagne contient facilement plus de deux cent tests. Se font tester entre autres les fonctionnalités *Video On Demand*, *livestreaming*, l'API SOAP, l'interface web ou encore les connexions réseau. Après avoir donc reçu la liste des tests, `launch_testlink_campaign.py` la trie, vérifie que les fichiers ats existent et en enlève éventuellement quelque uns (liste noire locale) de l'ensemble. En fait, tous les fichiers de test sont enregistrés dans un dossier prédéfini (*repository*). Les noms des fichiers correspondent au nom du test dans la base de TestLink. Il est donc simple de savoir, si un test est déjà automatisé et s'il peut être exécuté maintenant.

La liste filtrée et d'autres paramètres deviennent alors la configuration de la campagne de test. Cette configuration est transmise à Testerman et l'exécution des tests déclenchée. Testerman exécute les tests les uns après les autres. Si le test contient une interaction avec l'interface web du SUT, Testerman fait appel à la probe Selenium. On voir d'ailleurs apparaître la fenêtre du navigateur et la simulation des actions utilisateur. À la fin de l'exécution de chaque test, un petit bout de code permet de mettre à jour la base de donnée TestLink (afin retenir que tel test de tel firmware est passé ou pas).

FIGURE 2.2: Déroulement d'une campagne

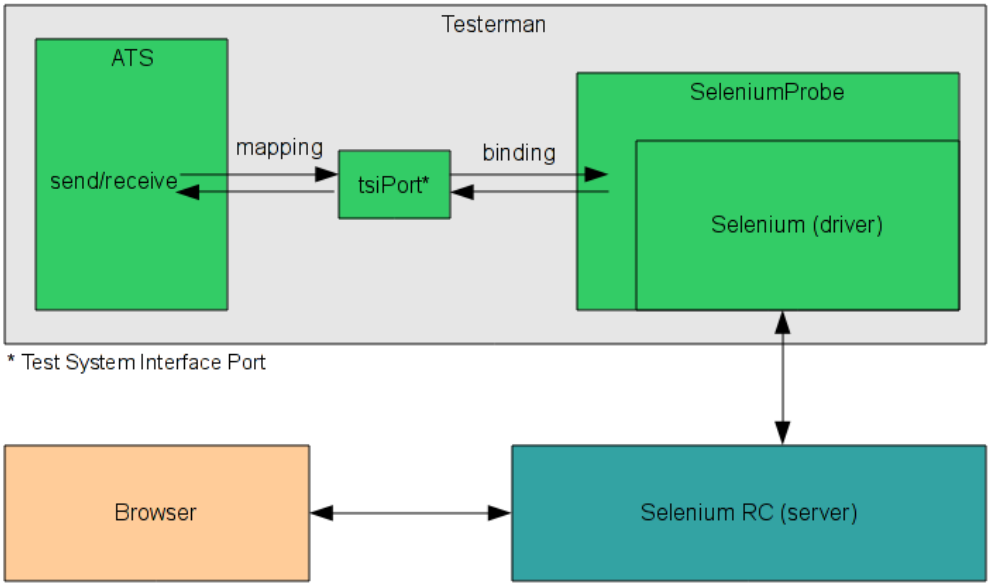


(1 : démarrage, 2 : récupération de la liste des tests, 3 : déclenchement de la campagne, 4 : exécution des tests, 5 : mise à jour de la base de données)

Quelques remarques concernant la procédure :

- La liste des tests n'est pas l'unique information transmise à Testerman. En effet, `launch_testlink_campaing.py` lit également le contenu d'un fichier de configuration et transmet son contenu. Ainsi, chaque développeur peut avoir sa configuration locale écrasant les valeurs par défaut dans chacun des cas de test. Cette possibilité est particulièrement pratique lorsque les équipes de tests ne travaillent pas toujours sur le même site : la configuration locale rend compte de l'environnement différente (p.ex. la désignation d'autres satellites). L'exemple sous 2.6.1 illustre une telle « individualisation » .
- Il est possible de n'exécuter qu'un seul test. À ce moment là, il est plus simple (et plus propre) de le traiter comme une campagne contenant qu'un seul test.
- Un mode *debug* permet d'éviter de contacter la base de données (pour raison de performance).
- La probe Selenium de Testerman utilise le pilote développé par l'équipe Selenium.
- Et `launch_testlink_campaing.py` et une petite partie de code introduit dans Testerman utilisent l'API de TestLink pour la communication avec sa base de données.
- Testerman et Selenium RC doivent être lancés avant le processus décrit, puisqu'ils sont des serveurs qui se mettent à l'écoute de nouvelles commandes. Des scripts simplifient leur démarrage.
- Après toute exécution, il convient d'analyser celle-ci avec QTesterman (visualisation de l'exécution, analyse des logs). Tous les tests ne sont pas encore à la hauteur (ou tout simplement plus à jour). Il se peut alors qu'un test échoue tout simplement parce qu'il est mal programmé et non pas parce que le SUT ne réagis comme attendu.

FIGURE 2.3: Interactions Testerman/Selenium



Les logiciels utilisés ont parfois les même fonctionnalités. Ainsi, Testerman peut enregistrer les résultats des campagnes (et même en faire des rapports). Mais TestLink est utilisé pour achever ce but. Ce fait s'explique, en outre, par un certain nombre de tests exécutés manuellement. Ils ne peuvent par définition par apparaître dans Testerman. La base de donnée de TestLink se prête également mieux pour la conception de tests (quel firmware, conditions, etc.). De même, Selenium IDE serait capable de simuler lui-même les actions utilisateurs (sans Selenium RC). Procéder ainsi est fortement déconseillé parce que l'utilisation de l'extension Firefox ne se laisse pas automatisé et l'intégration dans une environnement de test automatisée semble pas faisable. L'entreprise se sert donc chaque fois d'une partie des logiciels pour en faire un ensemble fonctionnel.

2.6.1 La configuration d'une campagne

Comme déjà évoqué plusieurs fois, une campagne se configure individuellement. La configuration finale pour Testerman ressemble au bloc des métadonnées (définition de paramètres globaux) d'un script ats. La configuration personnelle de chaque développeur y figure. Après ces données, les scripts de test sont listés. De nouveau, on peut attribuer à certain paramètres une valeur spécifique pour le cas de test précis (mot-clés : `with`). Le listing 2.3 montre une configuration d'un développeur. Listing 2.4 illustre la configuration de Testerman qui peut en résulter.

Listing 2.3: Configuration personnelle

```
# Testlink
PX_UPDATE_TESTLINK=0
PX_TESTLINK_API_URL=http://123.1.2.3/testlink/lib/api/xmlrpc.php
PX_TESTLINK_DEVKEY=5b02044e8fb0bb21d624d0084854efc2 # rschroder

# license stuff
PX_HARDWARE_ID_REGEXP=^[a-f0-9]{34}$ # do not add \' around !!!

# location specific configuration for satellites
PX_DVBS_FREQ=11538

...
```

Listing 2.4: Configuration Testerman

```
# __METADATA__BEGIN__
# <?xml version="1.0" encoding="utf-8" ?>
# <metadata version="1.0">
# <description>Testlink campaign for 'None' run on '172.27.114.161'</description>
# <prerequisites>No prerequisites</prerequisites>
# <parameters>
# <parameter name="PX_UPDATE_TESTLINK" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TESTLINK_API_URL" default="http://123.1.2.3/testlink/lib/api/xmlrpc
    .php" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TESTLINK_DEVKEY" default="5b02044e8fb0bb21d624d0084854efc2" type="
    string"><![CDATA[]]></parameter>
# <parameter name="PX_HARDWARE_ID_REGEXP" default="^[a-f0-9]{34}$" type="string"><![CDATA
    []]></parameter>
# <parameter name="PX_DVBS_FREQ" default="11538" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TEST_PLAN_ID" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_BUILD_ID" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_IP_HOST" default="172.27.114.161" type="string"><![CDATA[]]></
    parameter>
...
# </parameters>
# </metadata>
# __METADATA__END__

ats /Firmware_Releases/anevia-1.ats with PX_TEST_CASE_ID=1
ats /Firmware_Releases/anevia-2.ats with PX_TEST_CASE_ID=2
ats /Firmware_Releases/anevia-3.ats with PX_TEST_CASE_ID=3
```

Chapitre 3

Concept et Considérations

Ce chapitre explique plus précisément les buts du projet. Puisque il s'agit de génération de code, le pour et le contre de celle-ci est examiné de plus près. Par ailleurs, certaines sources de problèmes avec les tests automatisés d'interface web montrent les limites de l'approche. La description du développement montre la méthodologie choisie.

3.1 Les objectifs du projet

L'objectif général du stage et du projet est de rendre le travail de l'équipe testing plus ergonomique et efficace. Plusieurs logiciels interagissent lors de l'exécution des tests, qu'ils soient automatisés ou pas. Ce sont souvent les étapes manuelles entre l'utilisation de deux logiciels différents qui prennent du temps. Dans le contexte d'Anevia, les logiciels Testerman et Selenium interagissent. La création (= programmation) des tests pour Testerman se fait majoritairement à la main. Selenium IDE est utilisé pour repérer l'identification des éléments HTML sur l'interface web mais ces informations doivent être copiées à la main dans le fichier ats représentant le test correspondant. Mais Selenium IDE est à la base prévu pour enregistrer chaque interaction de l'utilisateur avec l'interface web et de l'exporter dans un format quelconque. Il est donc souhaitable d'avoir un dispositif qui traduit automatiquement les actions enregistrées dans un format compréhensible pour Testerman. Ce logiciel, qu'on appelle **formateur**, devra également prendre en compte les détails spécifique à l'infrastructure d'Anevia.

Parfois, un test doit être adapté à des nouveaux besoins. Le test a été mal conçu, l'interface du SUT a changé ou les exigences par rapport au résultat ont changé. Dans ce cas, le développeur a deux possibilités : changer le test à la main ou régénérer le code. La première option est moins propre puisque le programmeur risque de rajouter des petits « hacks » peu souhaitables. La régénération du test (avec bien sur les changements revendiqués) n'est possible que si on dispose les sources du test, c'est à dire des actions antérieurement enregistrées. Un des objectifs à réaliser consiste donc à pouvoir extraire les actions originales à partir du code généré. La possibilité d'une **importation** (d'un script de test Testerman dans Selnium IDE) sans peine est un deuxième objectif. Manœuvrer facilement entre les deux formats serait d'une grande aide pour les testeurs.

Le résultat généré doit être exact. En dehors de cette évidence, la formalisation peut entrainer un meilleur style du code source. Plus le modèle est conforme aux bonnes pratiques

de la programmation et des logiciels en question, plus le code sera lisible et efficace. Un bon exemple peut faire école pour tous les tests futurs.

Dans le cadre du projet, une bonne documentation est également prévue. Des explications détaillées doivent permettre aux futurs utilisateurs d'utiliser le générateur de façon autonome. La communication de conseils et des bonnes pratiques évite de faire des erreurs simples et rend le travail efficace dès le début.

Comme déjà mentionné, les besoins concrets de l'entreprise sont à prendre en compte. Ceci n'empêche néanmoins pas de développer un générateur général utile à l'extérieur de l'entreprise. Il faut donc trouver une solution propre qui sépare les deux aspects.

3.2 Pourquoi générer du code ?

Il y a plusieurs types de génération de code. On différencie la génération du *bytecode*, donc du code machine, de la génération de code source. Dans notre cas nous nous ne soucions que du second. Le code source peut servir à deux choses différentes. Soit, il est complet, c'est-à-dire il peut être compilé et le résultat obtenu est déjà outil. Soit il sert comme base pour des modification manuelles ultérieures. Quand il n'y pas be besoin de manipuler le code après sa génération, il n'y pas de contraints en dehors de son exactitude syntaxique : il pourrait être « illisible » pour l'être humain.

La structure générale du code source (du même langage) est souvent semblable. Il parait donc tout à fait normal de laisser faire la machine le travail de base avant de se lancer dans la programmation. La génération de ce genre de code source comporte beaucoup d'avantages. Il est évident que l'ordinateur peut « écrire » beaucoup plus vite qu'un programmeur. Le générateur est capable de créer plusieurs centaines de lignes de code en moins d'une seconde. Pour faire cela, il n'a que besoin d'un gabarit (flexible) de la structure générale du code à générer et d'informations sur le contexte actuel. L'écriture des parties invariantes du code n'est qu'un gaspillage de temps pour le programmeur. Leur génération lui laisse le temps de se consacrer au problèmes plus créatifs.

Le générateur permet de favoriser certaines styles de programmation. Les conventions (de dénomination ou de structuration) sont respectés ce qui rend le code très lisible et plus simplement révisable. Il ne s'agit pas d'accuser les programmeurs de mauvais style. Mais plus que la façon d'écrire du code est formalisé, moins il y a d'erreurs d'inattention et plus le travail sur le projet est simplifié. La qualité du code écrit à la main peut varier, alors que le générateur est toujours aussi performant (ou pas). Il est comme une mémoire très puissante : une fois intégré le bon comportement, il s'en « souviendra » toujours. La qualité du code ne peut qu'augmenter. Les programmeurs sont par la suite contraints à suivre le chemin tracé ce qu'il améliore également leurs compétences (en supposant que le code généré est meilleur en pratique). En dehors de la qualité et de l'efficacité, il y a donc un aspect pédagogique non négligeable.

Le générateur traduit de l'abstrait vers le concret. Les outils de conception et les environnements de développent profitent beaucoup aux ingénieurs, qui peuvent réaliser leurs idées très vite.

L'autre côte de la médaille est que l'on fait très vite confiance au code généré sans remettre en question sa qualité et/ou son efficacité. En plus, la machine reste quand même assez

limitée. Dans le cadre de ce projet, il est arrivé parfois qu'il faille régénérer le même code après l'avoir déjà modifié à la main. À ce moment la, il est difficile de gérer les versions différentes. Le problème ne se poserait évidemment pas sans la possibilité de régénérer le code.

3.3 Limitations du formateur et des tests automatisés d'interface web

Évidence, le formateur n'exporte que des commandes Selenium. Or, beaucoup de tests incluent non seulement une interaction avec l'interface web du SUT mais aussi d'autres vérifications « derrière » . Dans le contexte d'un produit Anevia, on imagine par exemple la configuration de l'envoi d'un flux vidéo sur le réseau. Même si l'interface affiche le bon enregistrement de la manipulation, il faut toujours vérifier la réception du flux sur une machine distante. Il y a donc beaucoup de cas (au moins dans l'entreprise) où le code généré n'est que le début de la création du test. À ce moment là, il faut avoir la bonne procédure pour ne pas confondre la version (nouvellement) générée avec celle modifiée auparavant.

Selenium IDE enregistre les actions utilisateurs. Il n'est donc pas possible anticiper ces actions (p.ex. sur un interface pas encore implémentée) ou des fonctionnalités futures. Un développement piloté par les tests (*test driven development*) ne peut pas être réalisé. Ce n'est pas problématique pour le formateur puisque ce n'est dans aucun cas son but. Ceci dit, dans la même idée, les tests sont très dépendant de la structure de l'interface. Si elle évolue (et ce serait donc comme si le test [déjà existant] était écrit avant l'implémentation proprement dite), l'exécution du test échoue à cause d'une commande selenese impossible (cliquer sur un élement non-existant, ...). En ce qui concerne le site web, le code généré n'est pas très stable. Ni Selenium IDE ni le formateur peuvent changer ce fait. Il faut donc créer le test relativement tard par rapport au développement de l'interface web ou du moins s'assurer que certains éléments clés du site sont toujours accessibles. L'état du système sous test doit également être bien défini.

En général, tout ce que peut faire l'interface web est représenté par le navigateur. Il peut y exister des exceptions comme le déclenchement d'un téléchargement d'un fichier. Un test doit éventuellement vérifier le contenu du fichier (imaginons que l'interface web génère un fichier à chaque téléchargement). Avec le formateur, on ne peut qu'aller jusqu'au moment du téléchargement. En utilisant Testerman comme environnement de test, le problème est relativement simple à résoudre. Son approche universelle s'avère ici très utile : une probe (ou même quelques lignes de Python directement inséré en « clair ») examinerait très vite le fichier.

L'exécution automatique d'un test est assez formidable. On n'y trouve quasiment que des avantages. Néanmoins, il y aura probablement des cas ou la réaction du SUT est si dynamique que la création d'un script de test (généré ou pas) sera compliquée. Dans ce cas, il est sûrement légitime de questionner le fonctionnement du système sous test qui semble assez compliqué. Mais tout de même, le testeur doit réfléchir sur l'utilité d'un script de test difficiles à maintenir.

Le formateur provoque la *test code duplication* [1]. Il peut y avoir des tests ressemblants

(notamment pour la *fixture*) pour lesquelles l'utilisateur exécute les même actions lors de l'enregistrement de son interaction avec le site web. Bien évidemment, le code généré sera pareil. Lors d'un changement de l'interface (web) du SUT, il faut refaire (dans notre contexte : réenregistrer) tous les tests. Une factorisation après la génération du code est concevable mais entraînera une conséquence lourde : le script de test ne sera plus compréhensible pour le formateur (lors de l'importation en Selenium IDE). C'est d'ailleurs un autre problème : un script de test manipulé à la main et importé dans Selenium IDE perdra ses additifs. Le formateur ne les reconnaît évidemment pas. Il faut donc « jongler » avec les versions différentes si on veut modifier un cas de test par le biais de Selenium IDE.

3.4 Déroulement du développent

Comme déjà mentionné plusieurs fois, un environnement de test existait déjà. Il y a une bonne centaine de tests automatisée sous forme de scripts. Néanmoins, certains de ces test ne fonctionnaient pas ou devaient être adapté aux évolutions de l'infrastructure de l'entreprise. Tout au début du stage, j'ai été amené à « réparer » quelques tests. Ceci m'a permis de bien comprendre le rôle de chaque logiciel employé. Plus important, je connaissais désormais la structure générale des script de tests. La lecture de la documentation de Testerman aide également à écrire un test proprement. Il était clair que le code généré doit satisfaire aux recommandations émises.

La deuxième partie consistait à effectuer une étude de faisabilité. Malgré la ressemblance de TTCN-3 (de Testerman) avec d'autres langages de programmation, il n'étais pas clair si c'était vraiment possible d'exporter les actions d'utilisateurs vers le Format Testerman. Ceci dit, un formateur pour Python existait déjà et il a finalement servi comme base pour les commandes les plus simples. Les extensions Firefox sont écrit en JavaScript et on peut tout simplement analyser leur code en ouvrant les fichiers correspondants. Leur étude rendait les interactions entre Selenium IDE et les formateurs explicites. Sur Internet, il y a également un peu (!) de documentation sur la façon d'écrire un formateur. Le code source de Selenium IDE n'est quasiment pas documenté (« Use the source, Luke! »). En fait, plusieurs possibilités (cf. ci-dessous) existent pour créer un nouveau formateur. Malgré moi, j'ai réalisé deux prototypes avant de me décider définitivement. Avec du recul, mes premieres tentatives d'implémentation se sont avérées être une bonne aide pour la suite. Je connaissais très bien la structure interne de Selenium IDE ce qui a beaucoup aidé lors du développement.

Suivit une longue période de « edit-compile-run » . J'ai fait une erreur de débutant pendant le développement : la communication avec les futurs utilisateurs était quasiment inexistante. Ce n'est que vers la fin du processus que la discussion en équipe m'a permit de mieux comprendre leurs besoins concrets et d'en tenir compte. Le développement a duré plus longtemps que je ne l'avais imaginé. J'ai sous-estimé l'ampleur des fonctionnalités à implémenter.

La documentation (explications du fonctionnement, lignes de conduite, capture vidéo de l'écran, ...) se faisait en même que la phase de tests détaillés. Après la correction de plusieurs bugs, le formateur a atteint une certaine maturité. Par la suite quelque cas de test dans le *repository* ont été remplacés par une version complètement générée. L'utilisation

du formateur dans un contexte réel donnait également des idées d'amélioration qui ont été implémentées directement.

3.5 Utilité en dehors de l'entreprise

Le formateur développé a pour but de resserrer les liens entre deux logiciels libres. Il était dès le début très important pour moi de créer une solution générique. Si le programme a une approche universelle – au lieu de ne prendre en compte que les exigences d'une seule entreprise – tout le monde pourrait en profiter. Anevia m'a conforté dans cette idée. N'importe quel individu ou entreprise qui déploie Testerman (comme environnent de test) pour commander Selenium pourrait profiter du développement. Tout au long du projet, j'ai gardé une séparation stricte entre les besoins spécifiques d'Anevia et l'utilisateur lambda. En effet, deux formateurs ont été développés. Le formateur spécifique à Anevia utilise des fonctionnalités de l'autre afin d'écraser certaines valeurs de configuration ou d'effectuer des petits changements dans le comportement.

Chapitre 4

Implémentation

Dans cette partie, la réalisation sera illustré plus concrètement. D'abord il est montré comment on peut rajouter un nouveau formateur dans l'infrastructure existante de Selenium IDE. Suivent les différentes possibilités d'implémentation avec leurs avantages et inconvénients. Les adaptations pour Anevia sont également expliquées. Quelques petits exemples illustrent la génération de code. Un cas de test complet est proposé parmi eux. Des réflexions sur la qualité du formater clôturent ce chapitre.

4.1 La liaison entre Selenium IDE et ses formateurs

Il a déjà été mentionné plusieurs fois que Selenium IDE enregistre les interactions de l'utilisateur avec l'interface web. Celles-ci sont non seulement les actions directes (cliquer sur un lien, entrer des données dans un formulaire, ...) mais aussi des vérifications (ex : vérifier la présence d'un élément HTML sur le site). Pour ces deuxièmes « commandes » Selenium IDE propose des menus contextuels afin de repérer les éléments sur l'interface et de rajouter les vérifications à la listes de commandes. Un simple cas par exemple serait l'entrée des données d'authentification dans un formulaire, l'appui sur le bouton de connexion et la vérification de la présence d'un certain texte (« Bienvenue ! ») sur la page suivante. Il y aurait aux moins trois commandes différentes à enregistrer. Quand l'utilisateur veut par la suite exporter cette liste d'actions dans un certain format/langage, Selenium IDE fait appel au formateur correspondant. Celui-ci reçoit la liste et doit traduire chaque selenese vers « son » langage (Python, Java, etc.). Pour rappel, il peut utiliser les pilotes – communiquant avec Selenium RC – fournis par les développeurs. En général, les formateurs rajoutent aussi un cadre (*header* et *footer*) autour du selenese pour que le code généré soit utilisable (=compilable) tout de suite.

4.2 Le Fonctionnement interne de Selenium

Avant de s'occuper du formateur Testerman, il faut examiner de plus près les *selenese*. En fait, il y a deux niveaux de commandes et Selenium IDE n'utilise pas forcement les même que Selenium RC. Ce dernier est responsable de l'accès direct à l'interface web. Les commandes de Selenium IDE représentent une couche supérieure. Prenons un exemple : avec

Selenium IDE, on peut stocker la valeur d'un élément HTML dans une variable, vérifier cette même valeur ou attendre qu'elle soit égale à une autre valeur prédéfinie. Dans les trois cas, l'élément HTML est le même. Les seleneses correspondant sont `storeValue()`, `verifyValue()` respectivement `assertValue()` (la différence entre assert et verify est expliquée plus tard) et `waitForValue()`. Tout cela n'a pas d'importance pour Selenium RC dont le seul but est de fournir cette valeur. Par conséquence, toutes ces commandes se basent sur la même commande de niveau inférieur, notamment `getValue()`. Ce qui change c'est la façon d'interpréter cette valeur. Évidemment, toutes les commandes n'ont pas deux niveaux, quelques unes sont suffisamment « simples » .

TABLE 4.1: Quelques exemples de commandes

La commande ... (Selenium IDE)	se base sur ... (Selenium RC)
`click()`	`click()`
`type()`	`type()`
`open()`	`open()`
`assertEditable()`	`isEditable()`
`verifyEditable()`	`isEditable()`
`waitForConfirmation()`	`getConfirmation()`
`storeConfirmation()`	`getConfirmation()`
`verifyConfirmation()`	`getConfirmation()`
`waitForConfirmationPresent()`	`isConfirmationPresent()`

En règle générale, toutes les commandes accédant à une propriété d'un élément HTML (son contenu, son état, ...) ont cette division des deux niveaux. Selenium RC connait ses *accessors* en tant que `isXXX()` (retournant un résultat boolean) ou `getXXX()` (retournant la valeur d'une propriété). Selenium IDE n'utilise ses commandes que sous forme « emballée » en y ajoutant plus de fonctionnalité. Comme déjà dit, ceci peut être par exemple une comparaison.

Un formateur – générant un programme qui va communiquer avec Selenium RC – doit se charger d'envoyer la bonne commande au serveur (`click()`, `type()`, ..., `isXXX()` ou `getXXX()`) tout en prenant en compte les différents types de traitement du résultat retourné. Cette différenciation est en partie déjà faite par Selenium IDE (cf. ci-dessous). Les accessors ont souvent deux arguments. Le premier, appelé *locator*, identifie l'élément sur le site web. Sa recherche se fait en utilisant les attributs `id` ou `name`, un chemin XPath ou autre. Le deuxième paramètre est souvent un *pattern*. Dans le cas d'une vérification, l'utilisateur souhaite comparer une valeur sur le site web avec ce pattern.

Pour plus de flexibilité, Selenium IDE propose aussi l'inverse de beaucoup de commandes. On veux par exemple que la valeur ne soit *pas* égale au pattern, que l'élément ne soit *pas* visible, etc. Encore une fois, c'est au formateur (c'est-à-dire au code généré) d'assurer cette fonctionnalité.

4.3 Comment rajouter un nouveau formateur ?

Les développeurs de Selenium IDE ont prévu la possibilité de rajouter d'autre formateurs (*formatters*) afin d'exporter les tests dans des formats différents. Un formateur rajouté se présentera tout naturellement dans l'interface graphique de Selenium IDE. Puisque ce dernier est écrit en **JavaScript**, il en est de même pour un nouveau formateur. Le navigateur Firefox met à disposition une infrastructure qui permet de facilement étendre ses fonctionnalités avec des extensions. Les modules complémentaires sont gérés par un système simple à utiliser. L'utilisateur peut chercher (sur le net) des extensions, les installer, les configurer, les désinstaller etc. Ce système gère également les dépendances entre les extensions (vaguement comparable au *package manager* de certaines distributions Linux). Pour ajouter un nouveau formateur, il suffit donc tout simplement d'en faire une nouvelle extension Firefox. Dans le fichier d'installation de cette dernière, le développeur précisera la dépendance du *plugin* Selenium IDE. Le nouveau formateur s'installe par la suite comme toute autre module complémentaire de Firefox. Ceci a pour d'avantage qu'on puisse notamment compter sur cette infrastructure pour la distribution des mises à jour.

Listing 4.1: Extrait du fichier d'installation

```
<em:requires>
  <Description>
    <!--ID of the Selenium IDE plugin-->
    <em:id>{a6fd85ed-e919-4a43-a5af-8da18bda539f}</em:id>
    <em:minVersion>1.0.5-SNAPSHOT</em:minVersion>
    <em:maxVersion>1.*</em:maxVersion>
  </Description>

</em:requires>
```

Pour transformer le formateur en un plugin Firefox, plusieurs petits fichiers de configuration sont nécessaires. Ces fichiers contiennent des informations en représentation XML ou des bouts de code en JavaScript afin de charger les véritables fonctionnalités de l'extension. La méthodologie est standardisée et n'est pas intéressante dans le cadre de ce projet. Pour résumer, il suffit donc de savoir, qu'il faut créer une extension Firefox afin de rajouter un nouveau formateur pour Selenium IDE.

4.4 Approche choisie pour l'implémentation

L'équipe de développement de Selenium déconseille fortement d'utiliser l'extension Firefox Selenium IDE pour la re-exécution des tests bien que cette possibilité existe. L'interface graphique n'est pas conçu pour être intégrée dans un environnent de tests automatisés. C'est le logiciel Selenium RC qui s'en charge.

Puisque n'importe quelle suite de commandes Selenium (*selenese*) peut être exportée par chacun des formateurs, il est donc logique qu'ils doivent fournir des fonctionnalités en commun. Selenium IDE prévoit une **interface de programmation** (*API*). Chaque formateur implémente certaines fonctions qui seront appelées par Selenium IDE lors de l'exportation du test. En fait, il existe même deux API (pour être correct, il faudrait préciser que la deuxième se base sur la première).

4.4.1 L'API pour les formateurs

L'API de base est très simple. Il n'y que très peu de fonctions à programmer.

Quand Selenium fait appel à un formateur pour l'export du test, la fonction `format()` de celui-ci est exécutée. Elle reçoit comme paramètre notamment la liste des selenese. En général, le code source créé contiendra des informations en-tête (import de bibliothèque, déclaration de variables, etc.), les commande selenese (traduites) et puis le *footer* (par exemple des accolades fermantes).

Pour faire du formateur un outil universel, le cas inverse, c'est à dire la génération d'un test Selenium à partir d'un fichier source dans un certain langage peut être envisagée également. En effet, l'API prévoit la fonction `parse()` qui doit permettre d'extraire les commandes selenese d'un fichier source.

Une troisième fonction, `formatCommands()`, génère le code pour chaque selenese. Cette fonction n'est non seulement utilisé pour la génération du code source complet mais aussi pour copier les seleneses dans le bon format au presse-papiers (*clipboard*). Un menu contextuel dans Selenium IDE permet ainsi d'extraire juste certaines commandes (préalablement sélectionnées) de la liste. Cette fonction est pratique quand on n'a pas besoin de générer tout un cas de test (p.ex. mise à jour/adaptation d'un test existant).

Une fois installé, le nouveau formateur s'affichera à la fin de la liste des formateurs à disposition sur l'interface graphique de Selenium IDE. Certains formateurs ont des options de configuration. L'API conforte le paramétrage des formateurs. Le développeur fournir l'objet `configForm` qui contiendra un formulaire de configuration sous forme XUL [1]. Le formulaire peut afficher les valeurs d'un autre objet, `options`. La fenêtre de configuration de Selenium IDE permet par la suite d'adapter chaque formateur à ses besoin individuels.

Le code source sur la page suivante montre un formateur vide tel qu'il est proposé par les développeurs de Selenium IDE [2].

1. XUL (XML-based User interface Language), crée par le projet Mozilla, décrit des interfaces graphiques
2. voir le fichier `/chrome/content/formats/blank.js` dans le plugin Selenium IDE

Listing 4.2: modèle d'un formateur

```
/**
 * Parse source and update TestCase. Throw an exception if any error
    occurs.
 *
 * @param testCase TestCase to update
 * @param source The source to parse
 */
function parse(testCase, source) {
}

/**
 * Format TestCase and return the source.
 *
 * @param testCase TestCase to format
 * @param name The name of the test case, if any. It may be used to
    embed title into the source.
 */
function format(testCase, name) {
}

/**
 * Format an array of commands to the snippet of source.
 * Used to copy the source into the clipboard.
 *
 * @param The array of commands to sort.
 */
function formatCommands(commands) {
}

/*
 * Optional: The customizable option that can be used in format/parse
    functions.
 */
//options = {nameOfTheOption: 'The Default Value'}

/*
 * Optional: XUL XML String for the UI of the options dialog
 */
//configForm = '<textbox id="options_nameOfTheOption"/>'
```

4.4.2 Une deuxième « API »

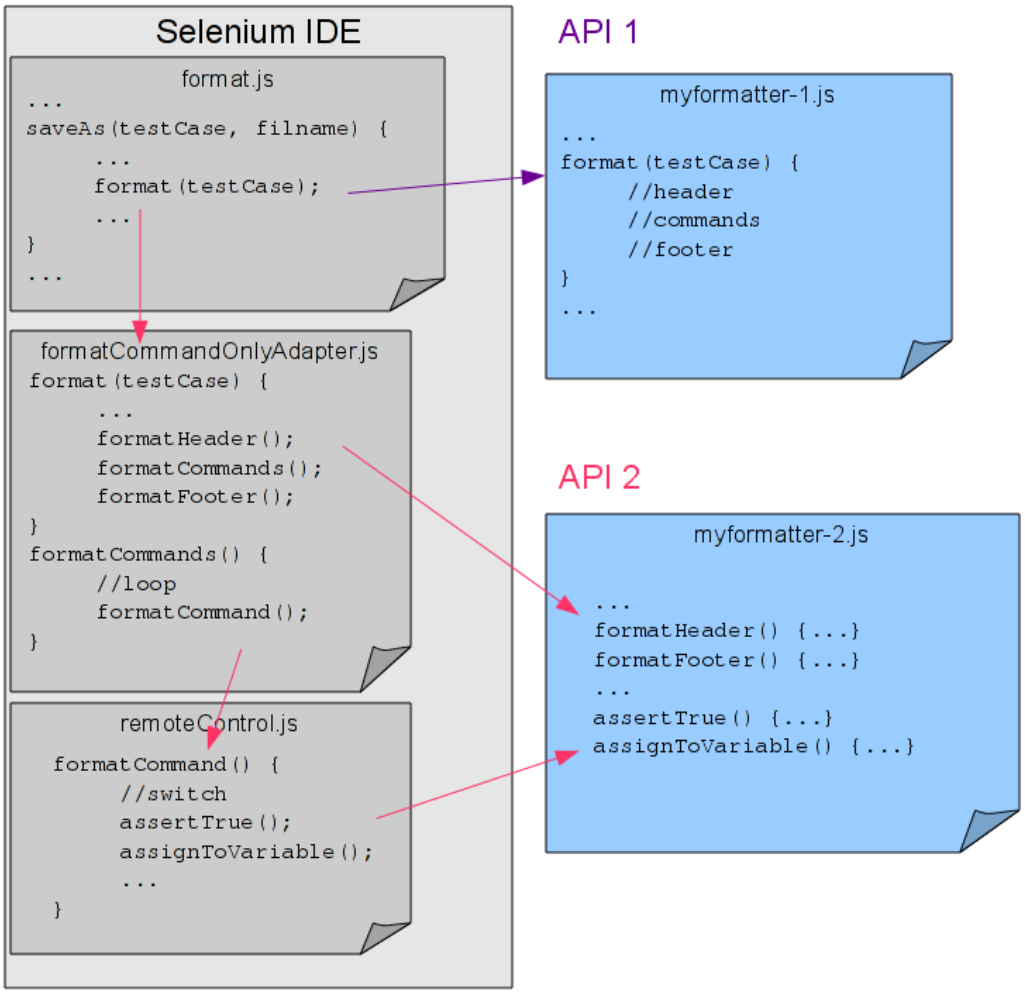
Une autre approche pour le formateur est possible. Certaines constatations le permettent. Les générations du header et du footer sont souvent assez statiques. Les commandes selenese se traduisent dans la plupart des cas par une ligne de code où on fait appel à la bibliothèque (le pilote) dans le langage qui envoie les commandes à Selenium RC. Étant donné que chaque commande selenese a une signature assez formalisée (nombre d'arguments, nom de la fonction), il est possible de faire appel à une fonction plus générique qui formate chaque commande. La plupart des formateurs existants fonctionne sur ce principe.

Selenium IDE appuie cette approche. Le logiciel fournit une couche d'abstraction autour

de l'API de base. Cette couche implémente la plupart des les fonctions de l'API. Elle se charge de la distinction des différentes commandes selenese. En se laissant aider par cette étape intermédiaire, le développeur ne doit se soucier que de la prise en compte des différents *types* de commandes. Une grande partie de travail est faite pour lui. La génération a perdu beaucoup de sa complexité (la fonction de différenciation des types de commandes est extrêmement générique et assez compliquée [environ 100 lignes de code, appel à plusieurs fonctions auxiliaires]).

Les fichiers `formatCommandOnlyAdapter.js` et `remoteControl.js` (fournis avec Selenium IDE) sont les clés du succès pour cette approche. Le schéma suivant illustre leur fonctionnement et la différence avec l'utilisation pure de l'API de base.

FIGURE 4.1: Les deux approches différentes



Comme on peut constater, le développeur doit fournir plusieurs fonctions lors de l'utilisation de la deuxième « API » . Ce sont bien souvent des petits *one-liners* qui retournent un bout de code bien spécifique à la syntaxe du langage ciblé. Il s'agit de la définition d'une comparaison (ex. `==` ou `is`), d'une affectation de variable (souvent `nomDeLaVariable = ...`), la définition d'une expression (Faut-il rajouter un point virgule après l'expression ?), etc. Dans la plupart des cas, l'implémentation est simple.

Listing 4.3: Quelques fonction exemplaires à redéfinir

```
// write a comment
function formatComment(comment) { ... }
// pause execution
function pause(milliseconds) { ... }
// print a message
function echo(message) { ... }
// assign return value to variable
function assignToVariable(type, variableName, expression) { ... }
// assert pattern matching (not!)
RegexpNotMatch.prototype.assert = function() { ... }
// verify equality
Equals.prototype.verify = function() { ... }
// verfiy a returned result
function verifyTrue(expression) { ... }
// ...
```

Après une première analyse de cette démarche, j'ai constaté qu'elle n'était pas applicable pour un formateur Testerman. Les qualités intrinsèques de TTCN-3 – et donc de Testerman – font en sorte qu'on ne peut pas se plier à la « forme » de cette l'API. En TTCN-3, la communication avec le SUT est scindée en deux parties : un message est envoyé, puis la réponse est explicitement reçue en appelant une fonction de réception. Or, Selenium IDE suppose que la même fonction émet la commande au serveur Selenium RC et retourne son résultat. En TTCN-3/Testerman, il faut impérativement séparer l'envoi du message de la réception de sa réponse. Chaque fois que l'on attend un résultat, une expression *alt* s'en charge. Une ou plusieurs branches « représentent » les expressions `if ... elseif ...` d'autres langages.

Il semblait alors que le formateur Testerman ne pouvait pas profiter de l'étape intermédiaire. Cela dit, les premiers prototypes s'inspiraient du fonctionnement interne (distinction des types de commandes et appel de fonction auxiliaires) proposé par la deuxième API. Mais plus le développement avançait, plus le code devenait très complexe. Il n'était pas sûr que toutes les éventualités soient prise en compte à cause de la complexité. Il faut gérer les deux niveaux de commandes selenese (voir section 4.2), veiller à ne pas introduire trop de la redondance (surtout pour la génération des expressions alt et les branches), s'occuper de l'inverse de commandes où c'est possible (`verifyPresent()` <-> `verifyNotPresent()`), gérer les différents types de réponse (boolean, simple chaîne de caractère, expression régulière) etc. Vu que l'implémentation propre des concepts TTCN-3 était déjà un défi en soi, ces détails étaient très problématiques.

Finalement, la deuxième API a « emporté le morceau » . En utilisant celle-ci, on est sûr de ne pas oublier quoi que ce soit et on peut se concentrer sur l'essentiel du langage ciblé (en occurrence TTCN-3/Testerman). De l'autre côte de la médaille, son approche prévu n'était pas toujours en accord avec les besoin de Testerman. Il a fallu détourner l'API de temps un temps. Un tel exemple est l'affectation du verdict *pass* : la fonction `setverdict()` est appelé dans les branches de l'expression alt. Pour des raisons sémantiques, il était souhaitable que le test ne mette le verdict *pass* qu'au dernier alt (bien qu'on puisse le mettre partout parce que un `setverdict(FAIL)` ultérieur l'écraserait). Or, la deuxième API appelle pour chaque selenese la fonction correspondante du formateur. Celle-ci ne « sait » pas si elle est la dernière à être appelée et s'il faut donc mettre

le verdict à *pass*. Le formateur Testerman utilise un compteur global qui indique le nombre d'expression alt restant (en les comptant avant l'exportation). Chaque fonction peut l'accéder et adapter son comportement (=générer un `setverdict(PASS)`) si nécessaire. Globalement, le formateur Testerman s'ajuste quand même bien aux prescriptions.

Toutes les fonctions prévues ne sont pas non plus implémentées. Parfois, elles le sont pour éviter un message d'erreur peu parlant mais ne font rien en particulier. Dans la cas où une des ces fonctions est appelée l'utilisateur est averti. Le logiciel lui demande de bien vérifier le code généré. Mais dans la plupart des cas, les fonctions ne font pas sens dans la notation Testerman/TTCN-3. Il existe, en outre, la fonction `notOperator()`, censée d'inverser le résultat (boolean) d'un comparaison. Cette opérateur n'est jamais utilisé parce qu'on va plutôt adapter les branches d'un alt à nos besoins.

Un autre exemple est la fonction `joinExpression()`. Il faut savoir que certaines commande selenese peuvent retourner un tableau de valeurs, par exemple `getAllLinks()`, fonction qui retourne les attributs `id` de tous les liens sur le site. Selenium IDE suppose qu'on puisse traiter le résultat arrivant avant la comparaison. `joinExpression()` est alors chargée de concaténer le tableau retourné. Ainsi, on peut ensuite le comparer avec un motif. En revanche, dans Testerman/TTCN-3, on propose plusieurs branches pour recevoir des résultats différents. Dans tous les cas, il n'est pas possible d'exécuter une opération sur le message reçu avant de le comparer avec la condition (=le template) de la branche. Ceci dit, un codec approprié (voir 2.4) pourrait le faire. Mais Testerman n'a pas un tel codec (certes très simple). L'exemple sous 4.6.6 montre un *workaround* (la fonction `joinExpression()` du formateur Testerman est en fait vide).

4.4.3 Une évaluation des deux APIs

Officiellement, quatre langages sont supportés par les développeurs Selenium (Java, C#, Ruby, Python). Plusieurs formateurs existent pour ceux-ci et pour d'autres langages. Ceci n'est pas étonnant puisque la mise à disposition d'un API (pour les formateurs) invite au développement dans l'esprit des logiciel libres. Les deux approches proposées pour écrire un nouveau formateur sont plutôt bien choisies. La première ne peut pas être plus générale. Elle semble être conçue pour des formats complètement hors standard. On pourrait imaginer un simple listage des commandes sans aucune opération supplémentaire par exemple. Je n'ai trouvé aucune implémentation en utilisant cette API (à part une formatant un fichier CSV en ne rajoutant que des virgules entre les commandes). Dès qu'on veut générer du code pour un « vrai » langage, la prise en compte de toutes les actions différentes devient assez compliquée. Je pense que c'est pour cette raison – et pour éviter une certaine redondance – que les développeurs ont créé la deuxième API. Celle-ci est très bien conçue. Le programmeur ne se doit quasiment pas soucier des détails des commandes parce qu'une étape intermédiaire fait ce travail. Les fonctions à implémenter sont simples et souvent très courtes. Malgré la mauvaise documentation, on arrive assez bien à créer un nouveau générateur de code en analysant les sources de Selenium et en s'inspirant des formateurs fournis. Évidemment, l'API doit supposer que tous les langages de programmation ont certaines caractéristiques en commun. C'est une réflexion qui me parait tout à fait correcte, bien que l'approche posât problème pour TTCN-3 (notamment pour la réception du résultat d'une fonction). Je pense tout de même que n'importe quel langage impératif pourra être prise en compte très bien par cette deuxième API. Elle est

à juste titre la référence quand il s'agit de développer un nouveau formateur.

4.5 Adaptation aux besoins de l'entreprise

Pour l'équipe de testing d'Anevia, il était important que le code source généré soit adapté à l'environnement de l'entreprise. Un script de test Testerman quelconque fonctionne évidemment partout mais pour représenter une infrastructure concrète, quelques modifications peuvent être nécessaires.

Pour garder une implémentation « générique » sans pour autant négliger l'entreprise, **deux formateurs** existent. Le premier génère du code Testerman proprement dit. Le deuxième construit un fichier ats adapté à Anevia. En fait, le formateur Anevia est un autre plugin Firefox. Il faut l'installer en plus du formateur Testerman sans lequel il n'est pas utilisable (voir section 4.3). Le formateur Anevia écrase de nouveau certaines fonctions du formateur « normal » . Pour encore plus de flexibilité, des fonctions *hook* sont appelé (si elles existent) lors de la génération du header/footer. Le formateur Testernam ne dispose pas d'implémentation pour ces fonctions. Contrairement au formateur Anevia qui peut ainsi injecter du code (ex : import d'une bibliothèque supplémentaire ou définition et affectation d'une variable globale). On l'utilise notamment pour communiquer l'adresse IP du SUT (remise lors de lancement d'une campagne, voir section 2.6) à Selenium RC. L'utilisation des *hooks* était la façon la plus propre d'éviter la redondance. Leur existence ne nuit quasiment pas la lisibilité ou l'efficacité du code. Néanmoins, ils témoignent de l'adaptation du formateur à un environnement concret.

D'autres détails sont tout simplement adaptables par la fenêtre de configuration des formateurs. Anevia par exemple utilise une petite fonction pour envoyer le résultat d'un test à TestLink après l'exécution. Le formateur importe cette fonctionnalité (=génère le code pour son importation). L'onglet de configuration permet d'y acceder.

4.6 Exemples de code généré

Dans cette section, quelques exemples de génération sont montrés. Ils montrent la transformation des commandes selenese en code Testerman. Certaines lignes de code généré sont volontairement omises afin de pouvoir attirer l'attention sur l'essentiel. L'exemple complet sur la page 36 illustre en outre la génération du header et du footer.

4.6.1 Exemple de base

Certaines commandes sont relativement simples à réaliser puisque elles ne demandent pas un traitement de leur résultat. Ce sont des actions comme ouvrir une page ou cliquer sur un bouton. Le formateur Testerman (comme les autres formateurs aussi d'ailleurs) fait tout simplement un appel aux fonctionnalités prédéfinies. Cet exemple montre également la génération pour d'autre langages. Les actions enregistrées sont l'ouverture du site `http://www.google.de/` et un clic sur le lien « Image » (ici référencé par son style CSS) :

Listing 4.4: Exemple de base

```
/**
 * Selenium IDE commands:
 * open('http://www.google.de')
 * click('css=#gb_2 > span.gbts')
 */

# Java
selenium.open("http://www.google.de");
selenium.click("css=#gb_2 > span.gbts");
# Python
sel.open("http://www.google.de")
sel.click("css=#gb_2 > span.gbts")
# Testerman
sel.send(["open", "http://www.google.de"])
sel.send(["click", "css=#gb_2 > span.gbts"])
```

4.6.2 Remplir un formulaire

L'exemple montre l'entrée de deux textes (dans des éléments ayant le nom « port » et « ttl ») et le clic sur une *checkbox* (nommée « rtp »). Un bouton, référencé par son style CSS est cliqué pour valider les entrées.

Listing 4.5: Exemple de base

```
/**
 * Selenium IDE commands:
 * type('name=port', '12345')
 * type('name=ttl', '6')
 * click('name=rtp')
 * clickAndWait('css=form[name=npvr] > input[type=submit]')
 */

sel.send(["type", "name=port", "12345"])
sel.send(["type", "name=ttl", "6"])
sel.send(["click", "name=rtp"])
sel.send(["click", "css=form[name=npvr] > input[type=submit]"])
sel.send(["waitForPageToLoad", "30000"])
```

Selenium IDE ajoute à certaines commandes l'expression `AndWait`. C'est utile pour donner à la page le temps de se charger. La commande est automatiquement générée sans que le formateur ne doivent interagir. Selenium RC reçoit également un *timeout* (ici 30000 millisecondes). Une fois dépassé ce délai (et la page toujours pas chargée), le serveur lance une exception.

4.6.3 Faire une pause pendant l'exécution

Parfois, il convient d'arrêter l'exécution du test. Il faut éventuellement laisser du temps à un composant hardware de démarrer une action (dans le cadre de l'entreprise ce serait p.ex. une carte DVB qui tune sur une nouvelle fréquence). On pourrait tout simplement utiliser la fonction `sleep()` fournie par une bibliothèque Python, mais cela n'est pas

conseillé. En effet, l'utilisation des *timers* permet à Testerman d'arrêter l'exécution du MTC/PTC (*Main/Parallel Test Component*) sans être bloqué [3].

Listing 4.6: Arrêter l'exécution

```
/**
 * Selenium IDE command:
 * pause('5000')
 */

# Python
time.sleep(5)
# Testerman
t_timer1 = Timer(5)
t_timer1.start()
t_timer1.timeout() #blocking
```

Le dénomination des variables (pour les timers) est croissante et automatiquement générée par le formateur. Un timer créé est tout de suite lancé. La méthode `timeout()` est bloquante et arrête l'exécution. Le timer créé est l'équivalent d'un timer TTCN-3.

4.6.4 Affectation d'une variable

L'exemple suivant montre une affectation d'une variable. On demande à Selenium RC d'affecter le texte (la valeur de cette propriété) d'un élément HTML à une variable avec le nom « myvariable » . L'élément est référencé par son nom.

Listing 4.7: Affectation d'une variable

```
/**
 * Selenium IDE command:
 * storeText('name=btnK', 'myvariable')
 */

# Python
myvariable = sel.get_text("name=btnK")
# Testerman
sel.send(["getText", "name=btnK"])
sel.receive(value = 'myvariableValue')
myvariable = value('myvariableValue')
log('myvariable = %s' % myvariable)
```

On constate plusieurs détails. Tout d'abord, le mot-clé **`value`** de TTCN-3 est implémenté en Testerman aussi. Sauf que, contrairement au standard, l'affectation du résultat d'un `receive()` ne se fait pas tout de suite, c'est à dire sur la même ligne (TTCN-3 : `MyPort.receive(MyType:?) -> value MyVar`, voir les exemples dans [3]). Testerman enregistre le message reçu d'abord dans une structure interne ; avec comme référence la valeur du paramètre `value` (l. 10) À n'importe quel endroit dans le même composant de test (MTC/MPC), on accède à ce cette structure interne en appelant la fonction `value()`. Elle permet de récuperer les valeurs enregistrées auparavant. Ici, il n'y a pas de *template* imposé (en tant que paramètre du `receive()`) puisque il s'agit d'enregistrer quoi qu'il arrive du port.

3. voir `http://testerman.fr/testerman/wiki/TestermanLanguageReference#Timers`

Le formateur ajoute automatiquement un message de log. L'expression permet de mieux analyser l'exécution du test. Inspirée par TTCN-3, la syntaxe n'est simplement qu'adaptée à Python. Dépendant du contexte (l'expression se trouvant dans la partie contrôle, dans le cas de test, ...), le message est ajouté au niveau de log correspondant.

4.6.5 Vérifier le contenu d'un élément

Pour évaluer le verdict d'un test, il faut souvent demander l'état du SUT. Dans ce cas, le logiciel de test doit faire la différence entre une expression *verfiy* et une *assert*. Il s'agit de définir la façon de réagir après l'échec d'une de ces expressions. Dans ce contexte, un « échec » veut dire qu'une partie du système sous test n'est pas dans l'état espéré (p.ex : une variable ne contenant pas la bonne valeur). Soit le test continue malgré cet échec, soit il s'arrête. Cela dépend de l'importance de l'élément à vérifier. Un état inattendu d'un composant primordial du système arrêtera le test, tandis que l'échec d'un « détail » ne le fera probablement pas (le test peut être considéré comme échoué toutefois). Dans Selenium comme ailleurs, un `assert` est souvent sensé être important pour le test. Les expressions `verify`, en revanche, sont négligeables. L'exemple suivant montre la différence de code généré. Le test vérifie le texte d'un élément référence par son id.

Listing 4.8: Assert contre Verfiy

```
/**
 * Selenium IDE commands:
 * assertText('id=headline', '42')
 * verifyText('id=headline', '42')
 * verifyNotText('id=headline', '42')
 */

# Testerman
sel.send(["getText", "id=headline"])
alt([
  [ sel.RECEIVE(template = "42"),
    lambda: log('getText(id=headline) == "42": Good!'),
  ],
  [ sel.RECEIVE(template = any_or_none()),
    lambda: self.setverdict(FAIL),
    lambda: log('getText(id=headline) != "42": Bad!'),
    lambda: stop(),
  ],
])
sel.send(["getText", "id=headline"])
alt([
  [ sel.RECEIVE(template = "42"),
    lambda: log('getText(id=headline) == "42": Good!'),
  ],
  [ sel.RECEIVE(template = any_or_none()),
    lambda: self.setverdict(FAIL),
    lambda: log('getText(id=headline) != "42": Bad!'),
  ],
])
```

```
33 sel.send(["getText", "id=headline"])
   alt([
     [ sel.RECEIVE(template = "42"),
       lambda: self.setverdict(FAIL),
       lambda: log('getText(id=headline) == "42": Bad!'),
38   ],
     [ sel.RECEIVE(template = any_or_none()),
       lambda: self.setverdict(PASS),
       lambda: log('getText(id=headline) != "42": Good!'),
     ],
43 ])
```

La syntaxe des expressions *alt* semblent peu commode au début. Cependant, on reconnaît la structure TTCN-3 : un *alt* contient plusieurs branches (des *guards* [pré-condition pour chaque branche] sont possible aussi), chaque branche à son tour contient le traitement à déclencher lors du reçu du template correspondant. En Testerman, les actions à exécuter dans une branche sont énumérées en tant que fonctions *lambda*. La raison pour cette curiosité se trouve dans la syntaxe du langage Python. Puisque Testerman est écrit en Python, il faut s'adapter au compilateur de ce langage. L'utilisation des fonctions lambda fait en sorte que les expressions ne sont évaluées qu'au moment de l'appel (=quand le test rentre cette branche). Le mot-clé *alt* est d'ailleurs implémenté comme une fonction ayant comme paramètre la liste des branches avec les actions correspondantes.

Les trois vérifications sont basées sur la même fonction `getText()`. Elle récupère la valeur de l'élément HTML avec l'id « headline » . On constate la différence entre la première est la deuxième commande : si la fonction ne retourne pas la valeur attendue (le paramètre template « 42 » , l. 11 et 22), l'exécution du test s'arrêtera dans le premier cas en appelant la fonction `stop()` (TTCN-3 : `stop;`). La deuxième commande met bien à jour le verdict, mais le test va poursuivre. D'ailleurs, Testerman implémente la même « hiérarchie » des verdicts que TTCN-3 : une fois que le test a un verdict `fail`, celui-ci ne peut plus le changer en `pass` (voir `inconc` ou `none`).

Le troisième alt illustre un test inversé. Le verdict a changé par rapport à la deuxième commande. C'est la façon la plus propre de gérer les seleneses « négatives » . On remarquera également la présence d'une deuxième assignation d'un verdict, notamment dans le cas d'un test réussi. Comme déjà évoqué, le formateur ne met le verdict du test à `pass` qu'à la dernière vérification.

Testerman fournit une multitude de fonctions pour définir un template. Ici, on voit à la fois une valeur brute (« 42 »), à la fois l'utilisation de la fonction `any_or_none()`. Elle permet d'accepter n'importe quel message reçu (même vide). Une autre fonction important est `pattern()` pour la vérification avec des expressions régulières. Les simples valeurs boolean sont également possibles comme « template » . D'autres mécanismes de comparaison sont entre autres `length()`, `empty()`, `greater_than()`. On peut utiliser des structures de données de Python (liste [tableau], dictionnaire [map], chaîne de caractère, ...) comme template. Il n'est donc souvent pas nécessaire de définir le template quelque part (bien que ce soit plus propre). Les codecs (voir 2.4) peuvent manipuler les messages avant leur comparaison avec le template. Testerman ne s'appuie pas sur un système strict de types de données et on peut relativement vite implémenter des vérifications.

L'utilisation de `any_or_none()` n'est ici que cosmétique. L'appel à un `RECEIVE()` sans paramètre aura le même effet (c'est-à-dire acceptera aussi tout message). Mais pour un néophyte, l'écriture est plus parlante.

4.6.6 Une vérification sur plusieurs éléments du même type

Le code suivant montre la vérification de tous les liens d'un site. Puisqu'il s'agit d'une opération sur un ensemble d'éléments, Selenium RC va retourner un tableau de valeurs. Le formateur prend le *pattern* (le mot-clé cherché) et « construit » un tableau autour afin d'être en accord avec le message reçu. On vérifie ici, si le mot « login » apparaît dans la liste des liens :

Listing 4.9: Récupérer un tableau

```
/**
 * Selenium IDE commands:
 * verifyAllLinks('login')
 */

# Python
try: self.failUnless(r"login", ','.join(sel.get_all_links()))
except AssertionError, e: self.verificationErrors.append(str(e))

# Testerman
sel.send(["getAllLinks"])
alt([
  [ sel.RECEIVE(template = [any_or_none(), "login", any_or_none()]),
    lambda: self.setverdict(PASS),
    lambda: log('getAllLinks() == "login" -> Good!'),
  ],
  [ sel.RECEIVE(template = any_or_none()),
    lambda: self.setverdict(FAIL),
    lambda: log('getAllLinks() != "login" -> Bad!'),
  ],
])
```

On constate les expressions `any_or_none()` autour du pattern. Pour résumer, maintenant le template est un tableau contenant zéro ou plusieurs élément(s) avant ou après l'élément contenant le mot-clé. La structure est étrange est l'utilisateur est averti lors de la génération d'un tel code (« Attention ! Vérifie bien ce que je viens de faire ! »). Quand on compare le code avec celui généré pour Python (ou d'autres langages), on trouve une différence importante : le mot clé cherché peut s'étaler sur plusieurs liens (dans le premier cas, puisque le tableau est concaténé avec `join()` et les éléments séparés par des virgules) ou doit être présent dans un élément unique (Testerman). La signification de ce test n'est donc pas du tout pareille. Le test en Python ci-dessus ne passera donc que s'il y avait un seul lien sur le site avec l'id « login » .

Mais le code Python pourrait par exemple vérifier que plusieurs liens différents sont présents (en donnant une expression régulière avec les symboles nécessaires pour omettre les virgules). Le code Testerman doit être adapté pour ce même but. Il faudrait quand même se demander ce que les développeurs de Selenium ont prévu pour cette fonctionnalité.

4.6.7 Paramétrage d'un test

Avant d'examiner un exemple complet, il faut encore connaître la gestion des paramètres par le formateur. La définition d'un module TTCN-3 peut inclure la définition de paramètres afin de le configurer. En Testerman, ce paramétrage se fait dans le bloc de metadonnées (et donc en format XML) en début du test. Un paramètre est défini par son nom, sa valeur par défaut et le type de celle-ci. Example :

```
<parameter name="PX_MY_PARAM" default="myvalue" type="mytype">
<![CDATA[]]>
</parameter>
```

Les noms des paramètres suivent en général la convention `PX_[_A-Z0-9]+`. Les paramètres sont globaux et accessibles depuis n'importe quelle partie dans le module (partie contrôle, à l'intérieur du cas de test, ...). La possibilité de paramétrage est très importante quand il s'agit de rendre un test flexible, c'est à dire utilisable dans des environnements différents. Dans le cadre du projet actuel par exemple, le formateur génère automatiquement pour tous les cas de tests quelque paramétré pour la probe Selenium, en outre le type du navigateur à utiliser ou l'adresse du SUT.

Ce système de paramétrage peut être utilisé pour des campagnes de test. La configuration d'une campagne peut contenir ces paramètres. Pendant l'exécution de chacun des tests, Testerman remplace la valeur (par défaut) du paramètre avec celle donnée par la campagne (si c'est la cas). On comprend tout de suite l'utilité : les paramètres figurant dans plusieurs modules différents (p.ex. données d'authentification) peuvent être changés en une ligne par la configuration de la campagne. Le test lui-même ne subit aucun changement. C'est la campagne qui connais tous les « détails » . Contrairement à ce qu'on pourrait penser, lors du lancement d'un test, ses paramètres globaux sont écrasés par les valeurs enregistrées dans la configuration. Il n'y pas question de *scope* global et local. Conformément à TTCN-3, où chaque module est indépendant, le test peut être exécuté sans configuration extérieure puisque chaque paramètre a toujours une valeur par défaut.

Chez Anevia, ces paramètres sont très utiles. Il se trouve que certains tests sont exécutés à des sites différents. Or, les machines sont parfois amener à tuner un signal sur un satellite spécifique. Pour simplifier, il suffit ici de savoir qu'il y a des données différentes (comme p. ex. la fréquence) à entrer dans l'interface web. Les tests localisés utilisent donc des paramètres globaux qui seront « entrées » sur l'interface web. Une configuration de campagne de test pour chaque site (et donc pour des différents satellites) rend le test polyvalent. La configuration est transmise à Testerman lors de lancement de la campagne.

Le formateur développé prend en compte ces qualités de Testerman. On peut le configurer en sorte qu'il génère toujours certains paramètres pour le bloc de métadonnées. Le formateur adapté aux besoin d'Anevia génère ainsi dans chaque script des variables pour la mise à jour du la base de données TestLink (numéro interne du cas de test, l'id du firmware, etc.).

Les quelque lignes suivantes montre l'utilisation de base d'un de ces paramètres. Pour accéder à la valeur concrète d'une de ces variables, on fait appel à la fonction `get_variable()`. L'exemple incite la probe Selenium à entrer dans un champ d'un formulaire une fréquence. Celle-ci était définie au début du module.

Listing 4.10: Paramètres globaux

```
/**
 * Testerman ats script
 */
# __METADATA__BEGIN__
# <parameter name="PX_DVBS_FREQ" default="11536" type="integer
   "><![CDATA[]]></parameter>
...
# __METADATA__END__

...
class TestCase123(TestCase):
  ...
  sel.send(["type", "id=freq", str(get_variable('PX_DVBS_FREQ'))
     ])
```

Le formateur peut faire plus que de simplement générer des paramètres pré-configurés. Pendant l'enregistrement (avec Selenium IDE), le testeur sait probablement déjà quelles actions devraient être paramétrées. Après l'entrée des valeurs durant l'exécution manuelle et avant d'exportation du test, une étape intermédiaire peut s'y rajouter. Le testeur peut remplacer chaque valeur par un nom de paramètre (`PX_XXX`) ou par ce nom et une valeur concrète. Lors de l'exportation, le formateur prendra en compte les différentes possibilités et génère le code correspondant : il génère si nécessaire la ligne de définition du paramètre en haut du script et fait appel à celui-ci (`get_variable()`) au moment de l'interaction avec le site web.

Le listing suivant illustre les différentes possibilités. Le formateur reconnaît les noms de paramètres avec une expression régulière. S'il trouve également une valeur par défaut (commande 3), le paramètre sera défini explicitement (pour explication : dans les commandes 3, 4, et 5, le testeur a remplacé les valeurs enregistré par Selenium IDE par une chaine sous la forme `${PX_NOM_DE_LA_VARIABLE}` avant faire appel au formateur)

Listing 4.11: Génération de paramètres

```
/**
 * Selenium IDE commands:
 * [1] click('id=button')
 * [2] type('id=freq', '11234')
 * [3] type('id=freq', '${PX_A:23111}')
 * [4] type('id=freq', '${PX_A}')
 * [5] type('id=srate', '${PX_B}')
 */

# __METADATA__BEGIN__
...
/* [3] */
# <parameter name="PX_A" default="23111" type="integer"><![CDATA
   []]></parameter>
# __METADATA__END__
```

```
...

/* [1] */
sel.send(["click", "id=button"])
/* [2] */
sel.send(["type", "id=freq", "11234"])
/* [3] */
sel.send(["type", "id=freq", get_variable('PX_A')])
/* [4] */
sel.send(["type", "id=freq", get_variable('PX_A')])
/* [5] */
sel.send(["type", "id=srate", get_variable('PX_B')])
```

On constate que le principe de l'indépendance du test ([6]) peut être violé en procèdent ainsi : Le paramètre `PX_B` n'est pas défini. Ce test ne peut être exécuté que dans une campagne (fournissant ce paramètre). Quand le développeur remplace une valeur concrète d'une commande selenese avec seulement un nom d'un paramètre, il suppose donc que la valeur ce celui-ci se trouvera soit dans les options du formateur (paramètre pré-configuré), soit dans la configuration de la campagne.

À tout moment, la fonction `set_variable()` permet changer dynamiquement la valeur d'un paramètre globale. Le formateur s'en sert notamment pour enregistrer une valeur (voir 4.6.4) quand le nom de la variable à affecter est sous la forme des paramètres globaux.

4.6.8 Exemple complet

L'exemple suivant montre le code source généré complet. Le fichier résultant dévient assez vite long et le nombre d'actions utilisateur est donc restreint. L'exemple est spécifique aux produits d'Anevia. Sur l'interface web de la machine, il est possible de changer son nom. Celui-ci est affiché en haut de la page. Le test imaginé consiste à changer le nom (*hostname*) et à vérifier qu'il est correctement affiché. Pour rendre le test utilisable dans une campagne, il faut remettre l'ancien état du système, c'est à dire changer le hostname encore une fois pour remettre le nom initial.

Les actions seront donc :

1. S'authentifier sur l'interface web
2. Accéder à la page avec la configuration réseau
3. Sauvegarder le nom de la machine actuelle
4. Changer le nom de la machine
5. Vérifier l'affichage du nouveau hostname
6. Remettre l'ancien nom
7. Déconnexion

Traduit en selenese, la liste devient déjà un peu plus longue :

Listing 4.12: Seleneses

```
/**
 * Selenium IDE commands (command | target | value):
 * open('/admin/login.php')
 * type('name=password', '${PX_PASSWORD2}')
 * clickAndWait('name=login')
 * clickAndWait('link=Network')
 * storeValue('id=hostname', 'PX_DEFAULT_HOSTNAME')
 * type('id=hostname', '${PX_NEW_HOSTNAME:newhostname}')
 * clickAndWait('id=apply_dns')
 * assertText('id=p3', 'regexp:^${PX_NEW_HOSTNAME}.*')
 * comment('clean up')
 * type('id=hostname', '${PX_DEFAULT_HOSTNAME}')
 * clickAndWait('id=apply_dns')
 * clickAndWait('css=img[alt=logout]')
 */
```

Le formateur Anevia connaît le mot de passe par défaut (voir section 4.6.7), d'où la référence à `PX_PASSWORD2` en l. 4. Il le rajoutera automatiquement dans les métadonnées. La vérification utilise une expression régulière parce que l'élément affichant le hostname (l'élément HTML dont l'id est 'p3') contient également l'adresse IP de la machine (l. 10). L'utilisation du signe Dollar pour référencer les paramètres globaux n'interfère pas avec sa signification dans les expressions régulières. Par souci de flexibilité, il ne vaut mieux pas tester la valeur entière de l'élément et juste s'assurer que le nouveau nom est présent au début de la chaîne de caractère.

Le code source (privé de quelque lignes sans importance) résultant :

Listing 4.13: Code source généré

```
# __METADATA__BEGIN__
# <?xml version="1.0" encoding="utf-8" ?>
# <metadata version="1.0">
# <description>change_hostname: This is the description</description>
# <prerequisites>prerequisites</prerequisites>
# <parameters>
# <parameter name="PX_SELENIUM_RC_HOST" default="localhost" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_SELENIUM_BROWSER" default="firefox" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_SELENIUM_RC_PORT" default="4444" type="integer"><![CDATA[]]></parameter>
# <parameter name="PX_SELENIUM_SERVER_URL" default="http://1.t100-2.lab1.anevia.com/" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_SELENIUM_CLOSE_BROWSER" default="1" type="integer"><![CDATA[]]></parameter>
# <parameter name="PX_TEST_CASE_ID" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_PASSWORD1" default="anevia" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_PASSWORD2" default="paris" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TESTLINK_API_URL" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_BUILD_ID" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_IP_HOST" default="192.168.11.252" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TEST_PLAN_ID" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_TESTLINK_DEVKEY" default="0" type="string"><![CDATA[]]></parameter>
# <parameter name="PX_NEW_HOSTNAME" default="newhostname" type="string"><![CDATA[]]></parameter>
# </parameters>
# </metadata>
```

```
# __METADATA__END__

# Please do not alter the following command list. It is used to extract a Selenium IDE
    test case from this ats file.
# command list (command | target | value):
#selenese: open | /admin/login.php |
#selenese: type | name=password | ${PX_PASSWORD2}
#selenese: clickAndWait | name=login |
#selenese: clickAndWait | link=Network |
#selenese: storeValue | id=hostname | PX_DEFAULT_HOSTNAME
#selenese: type | id=hostname | ${PX_NEW_HOSTNAME:newhostname}
#selenese: clickAndWait | id=apply_dns |
#selenese: verifyText | id=p3 | regexp:^${PX_NEW_HOSTNAME}.*
#selenese: type | id=hostname | ${PX_DEFAULT_HOSTNAME}
#selenese: clickAndWait | id=apply_dns |
#selenese: clickAndWait | css=img[alt="logout"] |


from TestlinkCase import TestlinkCase

#map selenium target with the IP of the SUT
set_variable('PX_SELENIUM_SERVER_URL', 'http://' + get_variable('PX_IP_HOST') + '/')

#change_hostname
class TC_CHANGE_HOSTNAME(TestlinkCase):
  # here might be python docstrings
  def body(self):
    #set up (port mapping)
    sel = self.mtc['port']
    port_map(sel, self.system['selenium_rc'])

    #selenium commands
    sel.send(["open", "/admin/login.php"])
    sel.send(["type", "name=password", str(get_variable('PX_PASSWORD2'))])
    sel.send(["click", "name=login"])
    sel.send(["waitForPageToLoad", "30000"])
    sel.send(["click", "link=Network"])
    sel.send(["waitForPageToLoad", "30000"])
    # store (selenium): PX_DEFAULT_HOSTNAME = getValue(id=hostname)
    sel.send(["getValue", "id=hostname"])
    sel.receive(value = 'PX_DEFAULT_HOSTNAME')
    set_variable('PX_DEFAULT_HOSTNAME', value('PX_DEFAULT_HOSTNAME'))
    log('PX_DEFAULT_HOSTNAME = %s' % str(get_variable('PX_DEFAULT_HOSTNAME')))
    sel.send(["type", "id=hostname", str(get_variable('PX_NEW_HOSTNAME'))])
    sel.send(["click", "id=apply_dns"])
    sel.send(["waitForPageToLoad", "30000"])
    sel.send(["getText", "id=p3"])
    alt([
      [ sel.RECEIVE(template = pattern(r"^" + str(get_variable('PX_NEW_HOSTNAME')) + ".*"
          )),
        lambda: self.setverdict(PASS),
        lambda: log('getText(id=p3) == regexp:"^' + str(get_variable('PX_NEW_HOSTNAME'))
            + '.*" [using PX_NEW_HOSTNAME] -> Good!'),
      ],
      [ sel.RECEIVE(template = any_or_none()),
        lambda: self.setverdict(FAIL),
        lambda: log('getText(id=p3) != regexp:"^' + str(get_variable('PX_NEW_HOSTNAME'))
            + '.*" [using PX_NEW_HOSTNAME] -> Bad!'),
      ],
    ])

    sel.send(["type", "id=hostname", str(get_variable('PX_DEFAULT_HOSTNAME'))])
    sel.send(["click", "id=apply_dns"])
    sel.send(["waitForPageToLoad", "30000"])
    sel.send(["click", "css=img[alt=\"logout\"]"])
    sel.send(["waitForPageToLoad", "30000"])


##
# Test Adapter Configurations
```

```
##
conf = TestAdapterConfiguration('local')
conf.bind('selenium_rc', 'probe:selenium', 'selenium', server_url = get_variable('
    PX_SELENIUM_SERVER_URL'), rc_host = get_variable('PX_SELENIUM_RC_HOST'), rc_port =
    get_variable('PX_SELENIUM_RC_PORT'), browser = get_variable('PX_SELENIUM_BROWSER'),
    auto_shutdown = get_variable('PX_SELENIUM_CLOSE_BROWSER'))


##
# Control definition
##
with_test_adapter_configuration('local')
verdict = TC_CHANGE_HOSTNAME().execute()
```

Le code demande peut-être quelques explications :

- Les paramètres `PX_TEST_CASE_ID`, `PX_TESTLINK_API_URL`, `PX_TEST_PLAN_ID` et `PX_TESTLINK_DEVKEY` (l. 12 – 19) sont utilisés pour communiquer avec TestLink. Le `DEVKEY` sert à l'authentification auprès du serveur TestLink se trouvant à `API_URL`. La `PLAN_ID` répresente le firmware du produit.
- Les lignes 27 à 37 contiennent la liste des seleneses. Cette liste sera exploitée afin de rouvrir un fichier ats avec Selenium IDE. C'est le processus inversé par rapport au formateur (la fameuse fonction `parse()`, voir section 4.4.1).
- La variable `PX_IP_HOST` contient l'adresse IP du SUT. Elle est communiqué par `launch_testlink_campaign.py`. Ligne 43 assure sa transmission à Selenium RC.
- La classe définie n'hérite pas de la classe `TestCase` de Testerman (l. 46). Mais `TestlinkCase` hérite de celle-ci. La couche intermédiaire se charge de la mise à jour de la base de données de TestLink après l'exécution du test. C'est une particularité due à l'infrastructure spécifique d'Anevia. Toutes les méthodes de `TestCase` (fournie par Testerman) sont évidemment à disposition.
- Lors du *binding* d'une probe (l. 91) avec un port, on peut transmettre des paramètres spécifique à celle-ci. Dans le cas de la probe Selenium, il faut qu'elle connaisse l'adresse de Selenium RC (`PX_SELENIUM_RC_HOST` et `PX_SELENIUM_RC_PORT`). Évidemment, l'adresse du SUT (`PX_SELENIUM_SERVER_URL`) doit être transmise aussi.

4.7 La Qualité et la performance du formateur

La quantité de code source du formateur est finalement plus importante que j'avais estimé. Comparé aux autres formateurs (Java, Python, Ruby, ...) il y a le double, voir le triple de lignes. Ceci n'est, selon moi, pas lié à une mauvaise factorisation mais plutôt dû aux principes de communication Testerman/TTCN-3. Le test d'une valeur est fait en une ligne pour un langage de programmation quelconque, alors qu'en Testerman il faut monter une expression *alt* avec deux branches. Il faut noter que le formateur Testerman « sait » faire plus de choses que les autres (notamment pour le paramétrage d'un cas de test).

Éviter de la redondance était très important lors du développement. Ainsi, le code est devenu assez « imbriqué » (plusieurs appel de fonctions). Malgré des commentaires explicatifs, le fonctionnement interne du formateur peut paraître difficile à concevoir au premier abord. Afin de faciliter la compréhension le plus possible, l'efficacité est parfois mise en retrait en faveur de la lisibilité. Par exemple, la liste de toutes les commandes est parcourue plusieurs fois avant d'être exportée au format Testerman. Chaque itération a un but différent et une grande boucle aurait été certes efficace mais moins compréhen-

sible. De même, certaines variables ne sont que déclarées pour que l'on comprenne leur sens en tant que paramètre concret d'un appel à une fonction (ex : `var doNotPrintX = true; printHeader(testCase, doNotPrintX);`). Vue que la performance n'est pas déterminante et que l'exportation est extrêmement rapide, j'ai préféré procéder ainsi.

Chapitre 5

Perspectives et résumé

Ce travail avait pour but de rendre une infrastructure de tests automatisés plus efficace en intégrant mieux certains de ces composants. Le formateur développé génère du code Testerman à partir des interactions utilisateur enregistrées avec Selenium IDE. Le résultat final implémente des suggestions d'une bonne programmation et fournit ainsi une bonne base pour le développement de cas de test. Le formateur n'était possible que à grâce aux modules existants dans Testerman (la probe Selenium assurant la communication avec Selenium RC) et Selenium IDE (la possibilité de rajouter facilement autres formateurs). L'accès libre au code des deux logiciels a d'ailleurs permit de développer une solution propre.

Il s'est avéré que Testerman et Selenium sont des outils très puissants pour l'automatisation de tests. Le formateur rend la combinaison des deux logiciels plus tentante et aide donc à améliorer les infrastructures ou les environnements de tests en général. Ce sont les petits outils liant les maillons de la chaîne d'automatisation qui la rendent efficace et utilisable.

Dans le cas de simples test d'interface le formateur fait un très bon travail. Le code généré représente un cas de test complet sans qu'on doive y toucher à la main après. Ceci dit, si on ne veut tester que des sites web, l'utilisation de Testerman est probablement superflue. Nombreux autres frameworks (pour lesquelles des formateurs Selenium existent) font aussi bien l'affaire. Mais Testerman se prête très bien dès qu'il s'agit de tester des « canaux » différents en même temps. Dans le cadre d'Anevia, les machines à distance ont plusieurs interfaces (web, ssh, SOAP) et les tests peuvent même comporter des vérifications au niveau du réseau local. Pour ces cas, le formateur Testerman ne fournit que le squelette, certes pratique, de base du cas de test automatisés.

Le développement des logiciels libres est toujours en mouvement. Pendant que j'ai effectué mon stage, plusieurs versions de Selenium RC/IDE sont sorties et Testerman a été également mis à jour. L'équipe Selenium a décidé récemment de changer la façon d'interagir avec le navigateur. Aujourd'hui deux possibilité existent : l'ancienne approche était d'injecter du code Javascript dans un page web préparé pour effectuer les actions utilisateur (c'est ce que fait Selenium RC). Le nouveau système, appelé WebDriver, permet de communiquer *directement* avec le navigateur en utilisant sa propre interface (ce qui est différent pour chacun d'entre eux). L'avantage est une API plus orientée objet supportant plus de navigateurs [8]. WebDriver est plus simple à utiliser et communique directement

avec le navigateur sans passer par Selenium RC. Pour faire fonctionner WebDriver avec Testerman, il faudra probablement créer une nouvelle probe. Le formateur Testerman doit être adapté, mais pas énormément, puisque l'interaction avec le navigateur sera fait par la probe. Il est envisageable de créer un formateur pour chaque approche de Selenium en espérant que Testerman supportera WebDriver dans le futur.

Le logiciel développé porte déjà ses fruits. Grande satisfaction m'est apportée par son utilisation sur place. Dès la première version opérationnelle, les développeurs d'Anevia l'ont adopté pour créer des nouveaux tests. Le formateur a atteint un stade mature et peut être utilisé pour chaque test d'interface web.

Je suis d'avis que le code généré est propre. Pendant l'écriture de ce mémoire, j'ai beaucoup lu dans le code source de Selenium/Testerman et je connait maintenant leur fonctionnement assez bien. L'étude du standard TTCN-3 a également donné des idées pour une implémentation propre. Tout au long du processus, j'étais en contact avec le développeur de Testerman et ses conseils ont influencé le développement pour le mieux. Je n'ai non seulement développé le logiciel mais je l'ai aussi utilisé et documenté. L'alternance entre développement, test (utilisation) et documentation a permis une évolution continue et tranquille du formateur au bénéfice de moi-même, de l'entreprise et de tout utilisateur futur.

Chapitre 6

Code source

Le code source et tout autre matériel (documentation, vidéos) se trouvent sur le CD joint.

Bibliographie

[1] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[2] Jens Grabowski, Dieter Hogrefe, Györgyb Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3) :375–403, June 2003.

[3] European Telecommunications Standards Institute. Methods for Testing and Specification (MTS) ;The Testing and Test Control Notation version 3 ;Part 1 : TTCN-3 Core Language (ETSI ES 201 873-1, V4.2.1). *Intellectual Property*, 1 :1–277, 2010.

[4] European Telecommunications Standards Institute. `http://www.ttcn-3.org/`, August 16 2011.

[5] Sébastien Lefèvre. `http://www.testerman.fr/`, August 12 2011.

[6] Gerard Meszaros. *XUnit Test Patterns : Refactoring Test Code.* Addison-Wesley, 2007.

[7] Hilmar Schmundt. Surfbrett mit Motor. *DER SPIEGEL*, 34 :111, 2011.

[8] The Selenium Team. `http://seleniumhq.org/`, August 16 2011.