Troisième partie

Mise en œuvre et et implantations

Chapitre 6

Plateforme de test de la composition de services Web

Sommaire

6.1 Introduction

La plateforme de test, décrite dans ce chapitre, a pour but de tester l'orchestration de services Web et de fournir un cadre pour la génération automatique de tests et de leur exécution. Elle met en œuvre notre approche de test détaillée dans le Chapitre 5. Les fonctionnalités de cette plateforme sont réparties en cinq phases : (i) modélisation de l'orchestration de services Web, (ii) génération de tests abstraits, (iii) concrétisation des tests qui consiste à dériver des tests exécutables qui sont fournis au Framework BPELUnit, (iv) édition et déploiement du service composé à l'aide de l'outil activeBPEL Designer, (v) l'exécution des tests qui est réalisée à l'aide de BPELUnit. Dans cette plateforme, l'architecture de test — représentant le service composé sous test ainsi que son environnement (client et partenaires) — est de nature distribuée : chaque service partenaire, intervenant dans un test, pourrait être simulé par un service testeur alors qu'un ou plusieurs clients pourraient être simulés par un seul service testeur.

La transformation de BPEL en IF et la description de l'outil BPEL2IF ont été publiées dans les deux conférences ECOWS 2008 [162] et NWeSP 2008 [163]. L'outil TestGen-IF a fait l'objet d'une publication dans la conférence DS-RT 2008 [181]. Il a été aussi utilisé dans deux travaux de collaboration qui ont été publié dans les deux conférences SITIS 2008 [182] et FORTE 2009 [183].

Dans ce chapitre, nous présenterons notre plateforme de test de la composition de services. Nous commencerons par décrire les différents outils développés et/ou utilisés dans cette plateforme : le moteur activeBPEL, le Framework de test unitaire BPELUnit, l'outil de transformation BPEL2IF et l'outil de génération de test TestGen-IF. Dans la seconde partie de ce chapitre, nous détaillerons une étude cas — test du service de prêt `loanApproval` — permettant d'illustrer notre approche de test et de montrer l'utilisation de notre plateforme de test. Pour cela, nous avons choisi de détailler trois scenarios de test parmi 17 scénarios probables pour ce service de prêt. Ces trois scénarios seront utilisés pour le test de la gestion de la corrélation des messages, de la gestion des timeouts et de quelques interactions du service composé `loanApproval`. Dans cette étude cas, nous expliciterons les différentes phases de notre approche : la transformation de la description BPEL du service `loanApproval` en spécification IF (par l'utilisation de BPEL2IF), la formulation des objectifs de test associés aux scénarios de test, la génération des tests (par l'utilisation de TestGen-IF), la concrétisation des tests, l'édition des tests et leur exécution dans le Framework BPELUnit.

6.2 Plateforme de test de l'orchestration de services Web

Dans cette section, nous présentons notre plateforme de test de l'orchestration de services Web. Cette plateforme met en œuvre notre approche de test boîte grise qui est a été définie dans le Chapitre 5. Cette approche se place dans le cadre du test de la conformité de l'orchestration de service Web, qui consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence (une description BPEL). Nous considérons ce test de conformité comme étant un test fonctionnel de type boîte grise où on connaît les différentes interactions entre le service composé, son client et ses partenaires.

Cette plateforme a pour but de tester l'orchestration de services Web, de fournir un cadre pour la génération automatique de cas de test et de leur exécution. Elle est illustrée par la Figure 6.1. Les fonctionnalités de cette plateforme peuvent être réparties en cinq phases :

1. la modélisation de l'orchestration de services Web, qui est réalisée par notre outil de transformation BPEL2IF. A partir d'une description BPEL d'un service composé, de sa description WSDL et de celle de ses partenaires, BPEL2IF produit une spécification (temporelle) de l'orchestration de services en langage IF. Cette spécification servira, dans notre approche de test de conformité, comme modèle formel décrivant le comportement du service composé ;
2. la génération des tests abstraits, qui est effectuée par notre outil de génération de test TestGen-IF. Cet outil permet de dériver automatiquement des tests temporisés à partir de la spécification IF et d'un ensemble d'objectifs de test décrivant les propriétés qu'on cherche à vérifier sur le service composé sous test ;
3. la concrétisation des tests, qui consiste à dériver des tests exécutables par l'outil BPELUnit ; cela à partir des tests abstraits générés lors de la phase précédente, et des interfaces WSDL du service composé et de ses partenaires. L'édition de ces tests se fait à l'aide de l'outil BPELUnit (plus précisément de son interface d'édition des suites de test) ;
4. l'édition et le déploiement du service composé (i.e. processus BPEL) sous test à l'aide de l'outil activeBPEL. Ce dernier est un moteur BPEL permettant aussi la gestion et l'exécution des processus BPEL ;
5. l'exécution des tests, qui est réalisée à l'aide de la plateforme de test BPELUnit.

Rappelons ici qu'une architecture de test pour les services composés représente le service composé sous test ainsi que son environnement (client et partenaires). L'architecture de test adoptée dans notre travail est de nature distribuée : chaque service partenaire, intervenant dans le cas de test, pourrait être simulé par un service testeur. Cependant, on pourra simuler plusieurs clients et les regrouper dans un seul service testeur. On se reportera à la Section 5.4 et à [105] pour avoir plus de détails concernant les architectures de test proposées pour le test des services Web.

Dans la section suivante, nous décrivons les outils utilisés dans cette plateforme de test : activeBPEL, BPELUnit, BPEL2IF et TestGen-IF.

6.3 Description des outils

Dans cette section, nous commençons par présenter brièvement le moteur BPEL activeBPEL (que nous utilisons tout au long de notre étude de cas pour l'édition, le déploiement et l'exécution des processus BPEL) ainsi que le Framework BPELUnit de test unitaire de BPEL (que nous utilisons pour l'exécution des tests dérivés par notre méthode de test). Ensuite, nous décrirons les deux outils : BPEL2IF et TestGen-IF, que nous avons développé pour mettre en œuvre notre méthode de transformation de BPEL en IF (décrite dans le Chapitre 4), respectivement pour implanter notre algorithme de génération de cas de test (défini dans la Section 5.5.2).

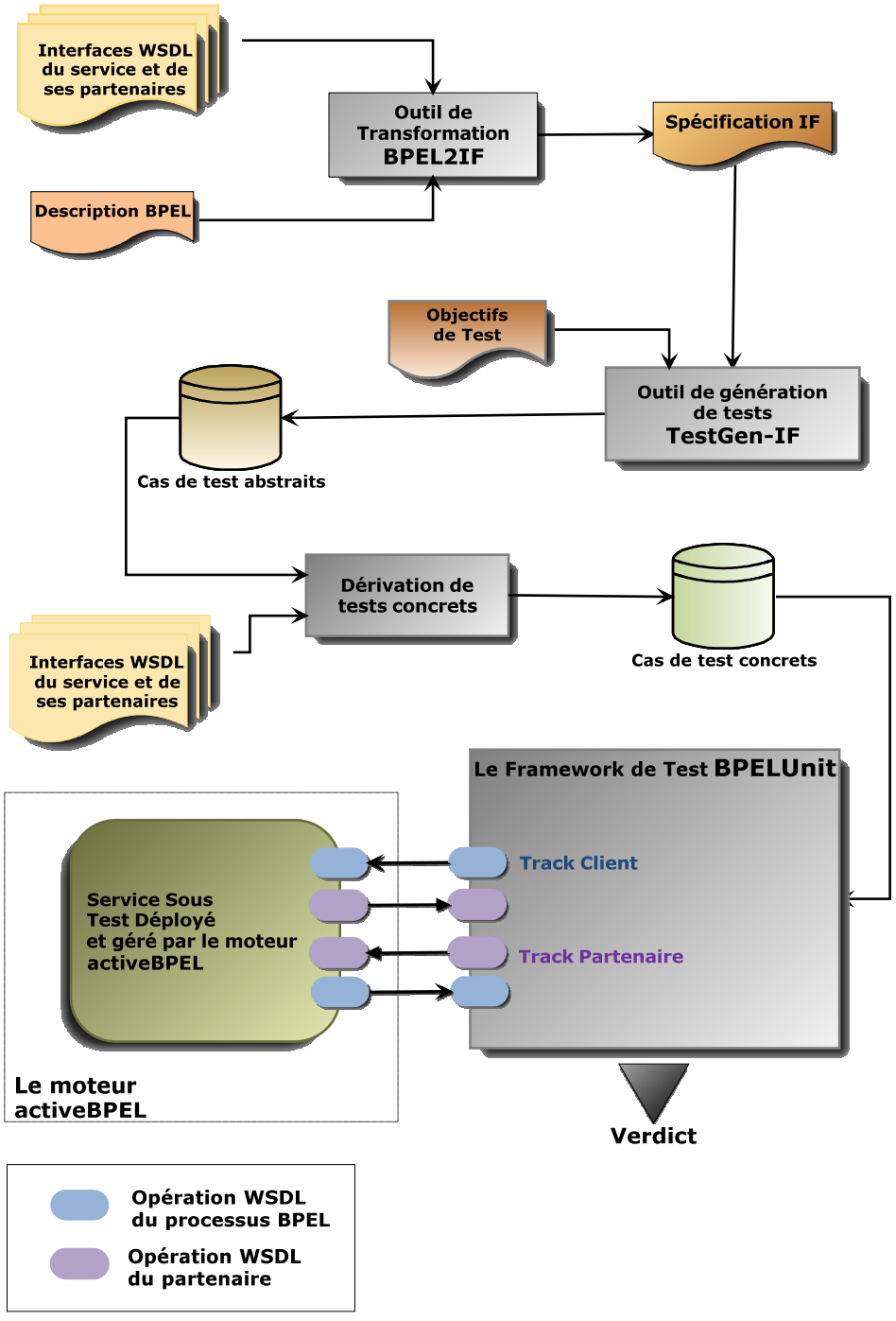

FIGURE 6.1 – Plateforme de test de l'orchestration de services Web

6.3.1 Le moteur activeBPEL

Un moteur d'orchestration BPEL, appelé moteur BPEL, est un outil d'exécution d'une orchestration décrite en BPEL. Il organise les envois/réceptions de messages et les appels de services Web selon l'orchestration décrite dans les processus BPEL. Il existe de nombreux moteurs BPEL (libres ou commerciaux) : ActiveBPEL de Active Endpoints [1], BPEL Process Manager d'Oracle [2], WebSphere Studio d'IBM [3], BPEL Maestro de Parasoft [4], Apache ODE [5].

Dans ce travail, nous utilisons le moteur activeBPEL étant donné qu'il intègre facilement le Framework de test BPELUnit — que nous décrivons dans la section suivante. activeBPEL est doté d'un serveur d'applications sous licence libre et reposant sur le serveur Tomcat [6]. Il est doté d'outils de conception et de déploiement de processus BPEL (e.g. activeBPEL Designer). Il possède une interface d'administration permettant la gestion et le déploiement des services et des processus BPEL ainsi que la création et l'exécution de leurs instances. La Figure 6.2 décrit l'architecture globale [7] du moteur activeBPEL — que nous ne détaillons pas ici.

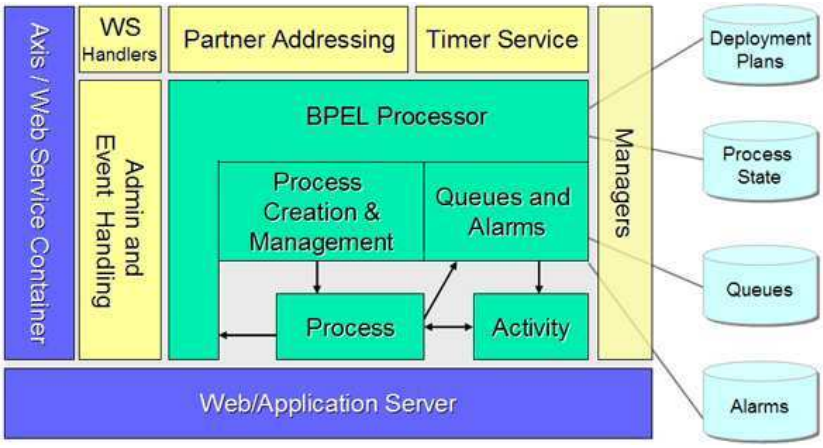

FIGURE 6.2 – Architecture du moteur activeBPEL

6.3.2 Le Framework de test BPELUnit

BPELUnit [8] est un Framework de test unitaire de BPEL qui a été proposé par Mayer et al. [106]. Dans ce Framework, le service composé sous test, noté SST, est isolé de son environnement réel. Il est effectivement déployé dans un serveur d'applications (e.g. Tomcat) alors que le client et les services partenaires du service SST sont simulés respectivement par BPELUnit comme un *Track client* et des *Tracks partenaires*. Lors de l'exécution des tests, tous ces Tracks seront exécutés simultanément comme étant des services indépendants.

1. `http://www.activevos.com/community-open-source.php`
2. `http://www.oracle.com/technology/products/ias/bpel/index.html`
3. `http://www-01.ibm.com/software/websphere/`
4. `http://www.parasoft.com/jsp/products/home.jsp?product=BPEL`
5. http ://ode.apache.org/
6. http ://tomcat.apache.org/
7. Source : `www.activebpel.org`
8. `http://www.se.uni-hannover.de/forschung/soa/bpelunit/`

Une suite de test en BPELUnit consiste en la description des deux parties (cf. Fig. 6.3) :

1. **section de déploiement** : qui contient toutes les informations de déploiement du service sous test et la spécification des services à simuler, i.e. le nom et la description WSDL du client et des partenaires (cf. Fig. 6.4) ;
2. **section de cas de test** : qui contient la description des cas de test contenue dans la suite de test. Chaque description d'un cas de test contient un seul Track client et un nombre de Tracks partenaires (probablement aucun) (cf. Fig. 6.5).

```
< tes:testSuite >
   <!-- Section de déploiement -->
   <tes:deployment>
   . . .
   </tes:deployment>
   <!-- Section de cas de test -->
   < tes:testCases >
      < tes:testCase name="...">
      . . .
      < tes:testCase >
      .
      .
   </ tes:testCases >
< tes:testSuite >
```

FIGURE 6.3 – Structure d'une suite de test dans BPELUnit

```
<tes:deployment>
   < tes:put name="LoanApproval" type="fixed">
      <tes:wsdl>customerService.wsdl</tes:wsdl>
   </ tes:put >
   < tes:partner name="ApproverPartner" wsdl="loanapprover.wsdl"/>
   < tes:partner name="AssessorPartner" wsdl="loanassessor.wsdl"/>
</tes:deployment>
```

FIGURE 6.4 – Spécification de déploiement dans une suite de test de BPELUnit

```
< tes:testCases >
   < tes:testCase name="LoanApprovalTest" basedOn="" abstract="false" vary="false">
      < tes:clientTrack >
      . . .
      </ tes:clientTrack >
      < tes:partnerTrack >
      . . .
      </ tes:partnerTrack >
      .
      .
   </ tes:testCase >
</ tes:testCases >
```

FIGURE 6.5 – Spécification d'une suite de test dans BPELUnit

Chaque Track contient un ensemble d'interactions, appelées activités, décrivant les actions attendues du service SST ou que doivent effectuer un partenaire/client. Chaque activité correspond à l'invocation d'une opération WSDL. En particulier, dans un Track client, une activité correspond à une opération WSDL fournie par le service SST.

On distingue six types d'activités : `Send Asynchronous`, `Receive Asynchronous`, `Send/Receive Synchronous`, `Receive/Send Synchronous`, `Send/Receive Asynchronous` et `Receive/Send Asynchronous`. Les deux premières activités (i.e. `Send Asynchronous` et `Receive Asynchronous`) correspondent à une interaction simple d'envoi et de réception d'un message. Les deux activités synchrones (i.e. `Send/Receive Synchronous` et `Receive/Send Synchronous`) correspondent à une interaction synchrone bilatérale, plus précisément, envoi/réception synchrone et réception/envoi synchrone. Les paires asynchrones (i.e. `Send/Receive Asynchronous` et `Receive/Send Asynchronous`) sont considérées comme des paires d'interactions simples (asynchrones).

L'édition (ou la création) d'un cas de test dans BPELUnit comporte deux étapes. On doit d'abord sélectionner un ensemble d'opérations que doivent invoquer le cas de test et les ajouter aux Tracks associés comme étant des activités. Un Track peut contenir plus d'une activité comme le montre le Track client de la Figure 6.6 (cf. Lignes 3 et 18). Ensuite, on doit préparer pour chaque activité, les *données XML* à transmettre au service SST et/ou les *conditions de vérification* qui sont utilisées, durant le test, pour vérifier les données envoyées par le SST et reçues par un Track client ou un Track partenaire.

La Figure 6.6 donne un exemple d'un cas de test `case1` composé d'un Track client et d'un Track partenaire. Le Track client (cf. Ligne 2 de la Figure 6.6) contient deux activités synchrones `Send/Receive Synchronous` (cf. Ligne 3 et Ligne 18). L'élément `data` (cf. Ligne 5) à l'intérieur de l'élément `send` de la première activité décrit les données XML à envoyer au service sous test. Quant à l'élément `condition` (cf. Ligne 12) à l'intérieur de l'élément `receive` de la même première activité, il décrit la condition de vérification des données reçues par le Track client en réponse à sa requête précédente. Notons que chaque activité d'un Track est associée à un service, un port et une opération qui font référence à la description WSDL d'un client ou d'un partenaire simulé par ce Track.

Le Framework BPELUnit permet l'édition ainsi que l'exécution des cas de tests. Au début de l'exécution d'un cas de test, le client (simulé par le Track client) initie le test par l'envoi des données (déjà préparées ou décrites) au service sous test (SST). Ce dernier invoque, tout au long du test, les opérations de ses partenaires — qui sont simulés par les Tracks partenaires. Ces partenaires reçoivent les données envoyées par le service SST et les vérifient selon les conditions de vérification qui leurs sont associées. Si tout se passe comme prévu, les partenaires invoqués retournent les données déjà préparées au service SST en réponse à sa requête. Dans le cas contraire et en cas d'erreur, le test se termine immédiatement et l'erreur est signalée au testeur.

6.3.3 L'outil BPEL2IF

BPEL2IF est un outil que nous avons développé en Perl et XML/XSLT, dans le but de transformer la description d'un processus BPEL en une spécification en langage IF. Il met en œuvre notre méthode de transformation de BPEL en IF présentée dans le Chapitre 4.

```
< tes:testCase  name="case1" basedOn="" abstract=" false "  vary=" false ">
   < tes:clientTrack >
      <tes:sendReceive  service=" loan:customerService "  port="customerServicePort"  operation=" request ">
         <tes:send  fault=" false ">
            < tes:data >
               <loan:firstName>Mounir</loan:firstName>
               <loan:name>Lallali</loan:name>
               <loan:amount>6000</loan:amount>
            </ tes:data >
         </ tes:send >
         < tes:receive  fault=" false ">
            < tes:condition >
               < tes:expression >accept</tes:expression>
               < tes:value >'yes'</ tes:value >
            </ tes:condition >
         </ tes:receive >
      </ tes:sendReceive >
      <tes:sendReceive  service=" loan:customerService "  port="customerServicePort"  operation="obtain ">
      . . .
       </ tes:sendReceive >
   </ tes:clientTrack >
   < tes:partnerTrack  name="AssessorPartner">
      <tes:receiveSend  service="loan2:LoanAssessor"  port="SOAPPort" operation="check">
      . . .
       </ tes:receiveSend >
   </ tes:partnerTrack >
</ tes:testCase >
```

FIGURE 6.6 – Exemple d'un cas de test dans BPELUnit

La Figure 6.7 décrit l'architecture de l'outil BPEL2IF qui contient les parties suivantes :

- **scripts de transformation** : Pour chaque version de BPEL (BPELWS 1.1 et WS-BPEL 2.0), est associée un script en Perl (applyTrasform1.1 et applyTrasform2.0) permettant la gestion des fichiers d'entrée/sortie de l'outil ainsi que l'application des règles de transformation de BPEL ;

- **règles de transformation** : A chaque activité des deux versions BPEL est associée à une règle de transformation décrite en langage XSLT [184] ; ajoutant à cela les règles de transformation des types de données décrites dans les fichiers WSDL, des variables et des liens partenaires de BPEL, des corrélations utilisées et des gestionnaires de fautes et d'événements ;

- **Entrées** : La description BPEL du service composé, de sa description WSDL ainsi que celle de ses service partenaires ;

- **Sortie** : La spécification formelle en langage IF de l'orchestration de services Web décrite par le processus BPEL.

Chaque activité de BPEL est transformée en un processus IF par le biais d'une règle de transformation XSLT. Le processus IF principal de l'élément `<process>` crée dynamiquement, à l'aide de l'instruction `fork` de IF, les processus de ses gestionnaires de fautes, d'événements et de son activité principale. Chaque processus IF doit être capable de terminer son exécution à la réception d'un message de terminaison `terminate`, et de propager la faute interceptée par l'envoi d'un message `faute` à son processus parent. Pour plus de détails, on peut se reporter au Chapitre 4.

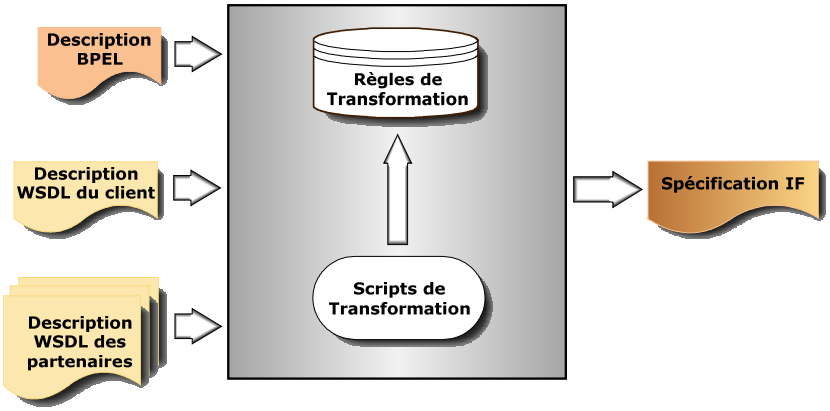

FIGURE 6.7 – Architecture de l'outil BPEL2IF

La transformation d'un processus BPEL se fait à l'aide de la commande suivante : `./applyTrasformX.X [-debug] ? <fchierBPEL> [-service=<fchierWSDL>]`* où :

- `debug` : désigne le mode en exécution interactive ;
- `<fchierBPEL>` : désigne le nom du processus BPEL à transformer ;
- `<fchierWSDL>` : désigne le fichier WSDL d'un partenaire du service composé.

Ci-dessous un exemple d'application de la transformation du processus `loanApproval` ayant deux services partenaires `loanassessor` et `loanapprover` (que nous décrivons dans la Section 6.4.1) :

```
./applyTransform2.0.pl loan/loan_approval.bpel -service=loan/loanapprover.wsdl
-service=loan/loanassessor.wsdl > loan.if
```

6.3.4 TestGen-IF

TestGen-IF est un outil développé en C++ permettant de construire à la volée des test temporisés à partir d'une spécification IF (d'une orchestration de services Web). Il effectue une exploration partielle de l'espace d'états du modèle guidée par un ensemble d'objectifs de test. Cet outil implémente l'algorithme de génération de test temporisés (décrit dans la Section 5.5.2) qui adapte la stratégie Hit-Or-Jump [6] aux automates temporisés de IF [160, 161]. L'implantation de l'outil TestGen-IF se base sur le simulateur de IF qui fait partie de la boîte à outils de IF [159] que nous décrirons brièvement dans la section suivante.

Boite à outils de IF

Rappelons tout d'abord que IF est un langage à base d'automates temporisés communicants [158, 159] qui se situe entre des formalismes de spécification de haut niveau (e.g. SDL [166], LOTOS [63]), et des modèles mathématiques utilisés dans la vérification et la validation des systèmes et protocoles (e.g. les automates temporisés [146]). IF est un langage assez expressif permettant de décrire les concepts existants dans les formalismes de spécification et de gérer les données, le temps et le parallélisme.

IF possède une boîte à outils pour des systèmes distribués, contenant des outils d'analyse statique, des simulateurs et des Model-Checkers. Il possède un environnement de modélisation et de validation de systèmes dont nous décrivons ci-après ses principaux composants :

Composant de transformation syntaxique Ce composant permet la construction d'un arbre syntaxique à partir d'une spécification IF. Cet arbre est constitué d'une collection d'objets C++ représentant tous les éléments syntaxiques figurant dans la spécification IF (e.g. signalroute, signal, state) [158, 159]. Ce composant possède une interface IF/API qui donne accès aux différents objets syntaxiques et offre quelques primitives de manipulation de ces objets ;

Plateforme d'exploration Cette plateforme permet une simulation de l'exécution d'un système IF (ensemble de processus communicants), la gestion du temps et la représentation de l'espace d'états. Elle possède une interface API permettant d'accéder au système de transitions étiquetées (LTS) correspondant à l'exécution de la spécification IF. En plus, cette interface offre quelques primitives pour : (i) la présentation et l'accès aux états et aux transitions, (ii) le parcours du système de transition à l'aide de deux fonctions : *init* et *successor*. La fonction *init* calcule l'état initial du système alors que la fonction *successor* calcule, pour un état donné, l'ensemble des transitions franchissables et celui des états successeurs.

Notons enfin qu'un compilateur IF2C a été réalisé à l'aide de l'interface IF/API afin de produire toutes les primitives de simulation (en langage C) pour des spécifications IF permettant ainsi une exploration dynamique du système.

Dans la suite, nous décrirons l'implantation de notre algorithme de génération de test temporisés (qui a été présenté dans la Section 5.5.2) dans l'outil TestGen-IF en utilisant la plateforme d'exploration et le simulateur de IF décrits ci-dessus.

Implantation de l'algorithme de génération de tests temporisés

TestGen-IF se base sur la plateforme d'exploration de IF, en particulier, sur le simulateur de IF. Ainsi, l'interface d'exploration (API) est utilisée pour implanter notre algorithme de génération de test décrit dans la Section 5.5.2. Pour la construction d'un graphe de recherche partielle, nous avons utilisé les primitives offertes par cette API afin d'accéder aux états du système et à ses transitions (actions et paramètres de signaux), ainsi que les primitives d'exploration du système de transitions : les fonctions de calcul de l'état initial (i.e. *init*), de calcul des transitions franchissables et des états successeurs (i.e. *successor*).

A chaque phase d'exploration partielle de l'espace d'états, c.-à-d. recherche d'une transition satisfaisant un objectif de test (Hit) ou atteinte de la profondeur limite d'exploration (Jump), TestGen-IF effectue :

- une construction en parcours en largeur d'abord (éventuellement en profondeur d'abord) de l'arbre de recherche partielle qui est sauvegardé dans une structure de file d'attente (éventuellement de pile) ;
- une sauvegarde des états visités dans une structure de données ;
- une construction à la volée d'une séquence de test partielle à partir de la racine de l'arbre jusqu'à la feuille du Hit ou du saut (Jump) ;
- une mise à jour des structures de données utilisées à chaque phase d'exploration.

En cas de satisfaction de tous les objectifs de test, TestGen-IF construit un cas de test abstrait à partir de toutes les séquences de test partielles, en ne gardant que les actions observables et les intervalles de temps.

Architecture de TestGen-IF

L'architecture de l'outil TestGen-IF est illustrée par la Figure 6.8. L'ensemble des objectifs de test, la spécification IF, la profondeur limite de recherche partielle ainsi que la stratégie de parcours (parcours en largeur d'abord (BFS) ou parcours en profondeur d'abord (DFS)) sont fournis en entrée à TestGen-IF. Ce dernier effectue une exploration partielle — parcours en BFS seulement pour l'implantation actuelle de TestGen-IF — de l'espace d'états du système résultant de la simulation de l'exécution de la spécification IF, et construit à la volée une séquence de test temporisée satisfaisant les objectifs de test.

Pendant la génération de test, lors de la satisfaction d'un objectif de test, seront affichées les informations suivantes : un message `Hit`, la description de l'objectif de test satisfait, et le nombre des objectifs qui restent à satisfaire. L'information `Jump` est affichée à son tour lors d'un Jump (saut). La génération se termine dès la satisfaction de tous les objectifs de test fournis en entrée ou si l'exploration du système a terminé (i.e. s'il n y a plus de transition franchissable à explorer). Dans le premier cas, un cas de test temporisé est obtenu à partir de la séquence de test en la filtrant : en gardant que les événements observables, à savoir les actions de réception et d'émission de messages (i.e. `input` et `output`), les intervalles de temps (i.e. `delay`), l'initialisation et la remise à *zéro* des horloges (i.e. les actions `set` et `reset` portant sur les horloges).

TestGen-IF fournit en sortie deux documents de sortie : un premier document contenant le cas de test temporisé ainsi construit, et un deuxième document contenant toutes les informations de génération : la description de tous les objectifs de test, le nombre de sauts (`Jumps`), la durée de génération, la longueur du cas de test, et le nombre d'états visités. Les tests abstraits construits par TestGen-IF vont servir pour la phase de dérivation des tests concrets qui seront fournis en entrée au Framework BPELUnit (décrit ci-dessus dans la Section 6.3.2).

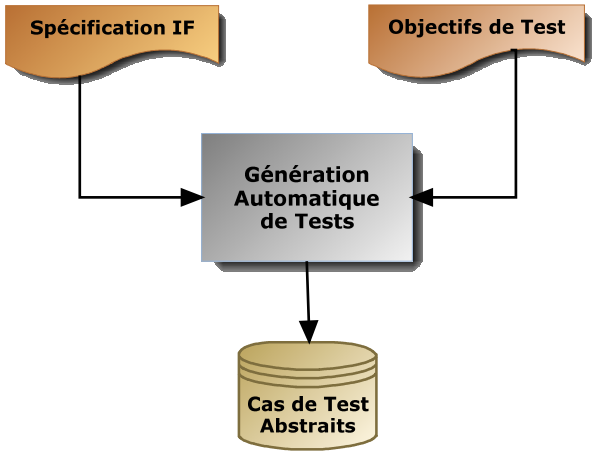

FIGURE 6.8 – Architecture de l'outil TestGen-IF

Dans la section suivante, nous allons détailler la formulation des objectifs de test dans l'outil TestGen-IF et nous donnerons un exemple de formulation. Notons que cette formulation est propre à l'outil TestGen-IF et donc à notre algorithme de génération de cas de test (cf. Section 5.5.2).

Formulation des objectifs de test

Un objectif de test décrit une fonctionnalité particulière de l'implantation sous test. Il est utilisé dans notre méthode pour guider l'exploration (partielle) de l'espace d'états du système. Dans notre formulation, certains objectifs de test peuvent être ordonnés, c.-à-d. ils peuvent être satisfaits selon un ordre donné. L'ensemble des test objectifs à satisfaire, noté *OBJ*, peut contenir alors un sous-ensemble ordonné d'objectifs tel que :

$$\begin{aligned} OBJ &= Obj_{ord} \cup Obj \\ Obj_{ord} &= \{obj_1, obj_2, \ldots, obj_n\} \; avec \; obj_1 < obj_2 < \ldots < obj_n \\ Obj &= \{obj_{n+1}, obj_{n+2}, \ldots, obj_m\} \end{aligned}$$

Chaque objectif de test, noté obj, porte sur une seule transition t du système IF. Il est décrit comme une conjonction de conditions portant éventuellement sur :

- une instance d'un processus *proc* dont l'identificateur est noté par *id* ;
- un état du système pouvant être un état *source* ou *destination* de la transition t ;
- une action du système contenue dans le label de t : un envoi de message (`output`), une réception de message (`input`) ou une action interne (e.g. `informal`, `task`) ;
- une variable *v* du processus *proc* ;
- une horloge *c* du processus *proc* (valeur, état actif/inactif).

Un objectif de test obj est décrit formellement comme suit :

$$\begin{aligned} obj &= cond_1 \wedge cond_2 \wedge \ \ldots \ \wedge cond_j \\ cond_i &= processus : instance = \texttt{\{proc\}id} \mid \\ cond_i &= état : source = \texttt{s} \mid \\ cond_i &= état : destination = \texttt{s} \mid \\ cond_i &= action : \texttt{input signal(parameters)} \mid \\ cond_i &= action : \texttt{output signal(parameters)} \mid \\ cond_i &= variable : \texttt{v = valeur} \mid \\ cond_i &= horloge : \texttt{c = valeur} \mid \\ cond_i &= horloge : \texttt{c est active/inactive} \end{aligned}$$

Ci-dessous, nous donnons un exemple de formulation d'un ensemble d'objectifs de test *OBJ* que nous utiliserons pour la génération de tests temporisés pour le service `loanApproval` (cf. Sect. 6.4.1). Cet ensemble est constitué de quatre objectifs ordonnés. L'objectif obj_1, par exemple, est une conjonction de quatre conditions portant respectivement sur la première instance du processus `sequencereceive14`, l'état `inState` qui est l'état source d'une transition *t*, `outState` qui est l'état destination de *t*, et l'action de réception du signal `creditInformationMessage1` avec les paramètres `{_,{Mounir,Lallali,6000}}`. Enfin, notons que la condition $cond_5$ de l'objectif de test obj_3 porte sur l'horloge $c34$ qui doit avoir la valeur 0 dans l'état `internalState` de la première instance du processus `ifpick34`.

$$OBJ1 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$$obj_2 = \cdots$$

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 0`

$$obj_4 = \cdots$$

Ci-dessous, nous donnons une brève description de l'utilisation de l'outil TestGen-IF.

Utilisation de TestGen-IF

La génération d'une suite de test se fait par le biais de la commande suivante : `./start-generation.sh -f <fichierIF> -d <profLimit> -s <typeParcours> -p <répertoireOBJ> -c <répertoireCasTest>` où

- `<fichierIF>` : désigne la spécification IF décrivant une orchestration de services Web ;
- `<typeParcours>` : désigne le type de parcours de l'espace d'états (BFS ou DFS) ;
- `<profLimit>` : désigne la profondeur maximale de l'exploration partielle ;
- `<répertoireOBJ>` : désigne le répertoire contenant la suite des ensembles d'objectifs de test. Chaque ensemble d'objectifs est décrit dans un fichier en langage C++ et sera utilisé pour la génération d'un seul cas de test ;
- `<répertoireCasTest>` : désigne le répertoire qui devrait contenir la suite de test construite par TestGen-IF.

Ci-dessous un exemple d'application de l'outil TestGen-IF pour la génération d'une suite de test du service composé `loanApproval` (que nous décrivons dans la Section 6.4.1) :

```
./start-generation.sh -f loan.if -d 30 -s bfs -p loan/test-purposes/
-c loan/test-cases
```

6.4 Étude de cas — test du service loanApproval

Dans cette section, nous présentons l'application de notre approche de test au service composé loanApproval, et nous décrirons l'utilisation de notre plateforme de test et de ses différents outils.

6.4.1 Présentation du loanApproval

Dans cette section, nous allons décrire le service loanApproval fourni par active endpoints [9] (le fournisseur du moteur activeBPEL) mais que nous avons modifié pour avoir deux variantes différentes du même service :

- une variante séquentielle : décrit avec un flot séquentiel par le biais de l'activité <sequence> de BPEL et des activités conditionnelles <if> [1]. Ce flot séquentiel du service loanApproval est illustré par la Figure 6.9 ;
- une variante parallèle : décrit avec un flot parallèle par le biais de l'activité <flow> de BPEL et des liens de synchronisation <links> [1]. Ce flot parallèle du loanApproval est illustré par la Figure C.1 en Annexe C.

Pour les besoins de notre cas d'étude et du test des propriétés temporelles de BPEL et sa gestion de corrélations de messages, nous avons aussi étendu les deux versions par l'ajout des corrélations, et d'une activité <pick> qui est munie d'un timeout et servant à attendre la confirmation de la demande du prêt de la part du client. La description BPEL du loanApproval séquentiel (ou variante séquentielle) est utilisée, dans notre cas d'étude, comme étant une spécification de référence de l'orchestration de services, alors que celle du loanApproval parallèle (ou variante parallèle) est utilisée comme une implantation sous test. Nous décrivons ci-dessous ce service de prêt

Description du service de prêt loanApproval est un service de prêt permettant d'accepter ou non un emprunt d'un certain montant. Les clients du service envoient leur demande de prêt, y compris leurs renseignements personnels (on se contente ici du nom et prénom du client) et le montant demandé. En utilisant cette information, le service de prêt peut accepter ou refuser ce prêt. Cette décision dépend du montant demandé et du risque lié au client. Pour de faibles montants de moins de 10000, un traitement simplifié est effectué. Pour des montants plus élevés (i.e. excédant 10000) ou à moyen/haut risque, la demande de prêt nécessite un traitement supplémentaire. Pour chaque demande, le service de prêt peut utiliser les fonctionnalités offertes par deux autres services partenaires : service d'évaluation de risques loanAssessor et un service d'approbation de prêt loanApprover. Pour le traitement des demandes de prêt à faible montant, le service loanAssessor est utilisé pour obtenir une évaluation rapide du risqué lié au client. Le prêt est ensuite accordé si le risque est faible. Dans le cas contraire (i.e. moyen/haut risque) ou si la demande de prêt est de plus de 10000, le service d'approbation loanApprover (éventuellement un expert de prêt) est utilisé pour accorder ou pas ce prêt. Dans tous les cas, le client sera informé de la décision du service de prêt. Dans le cas d'une acceptation de prêt, le service loanApproval attend une confirmation de la part du client pendant une certaine durée. Si cette confirmation est envoyée dans les bons délais au service de prêt , ce dernier envoie au client son accord du prêt, sinon il annule la demande de prêt.

9. Á télécharger sur cette page : http://www.activebpel.org/samples/samples-3/BPEL_Samples/doc/index.html

Dans la suite, nous décrirons le processus BPEL de la variante séquentielle du service `loanApproval` puisqu'il nous servira en tant que spécification de référence de l'orchestration de services décrite par le processus `loanApproval`. La deuxième variante (parallèle), nous l'utiliserons comme implantation du service de prêt. Notons que le code BPEL des deux variantes du `loanApproval` est complètement détaillé en Annexes A et B. Seul le flot de contrôle de la première variante est donné dans ce chapitre et illustré par la Figure 6.9, celui de la deuxième variante est illustré par la Figure C.1 en Annexe C.

Description du processus BPEL du `loanApproval` séquentiel L'activité principale est une activité `<sequence>` décrivant un flot séquentiel (cf. Fig. 6.9). Un client envoie la demande de prêt au service composé `loanApproval` qui est reçue par l'activité `<receive>`. Cette même activité initialise l'ensemble des corrélations `loanIdentifier` défini dans la description WSDL du `loanApproval` (cf. Ligne 18, AnnexeA). Cet ensemble de corrélation qui est constitué de deux propriétés `clientLastName` et `clientFirstName` (cf. Ligne 160, Section D.1, Annexe D). Ce service composé utilise une activité `<if>` pour vérifier si le montant est en deçà de `10000`. Si c'est le cas, il invoque son service partenaire `loanAssessor`, par l'intermédiaire de l'activité `<invoke>` synchrone, pour qu'il évalue le risque lié au client. Cette dernière activité permet au `loanApproval` de recevoir le niveau risque qui est vérifié à l'aide d'une deuxième activité `<if>`. Si ce risque est faible, il assigne `yes` à la variable `approval.accept`. Dans le cas contraire (i.e. risque moyen/haut) ou si le montant du prêt excède `10000`, `loanApproval` invoque, de la même manière que son premier partenaire par le biais d'une activité `<invoke>`, son deuxième service partenaire `loanApprover` qui peut accorder ou pas le prêt au client. Cette activité `<invoke>` reçoit la décision du `loanApprover` dans la variable `approval`. Dans tous les cas, le service composé envoie la décision (contenue dans la variable `approval`) au client par l'intermédiaire d'une activité <reply>. En cas d'acceptation du prêt, vérifiée par l'utilisation d'une troisième activité `<if>`, `loanApproval` s'attend à recevoir la confirmation du client dans un délai bien déterminé (e.g. `5` unités) par l'utilisation de l'activité `<pick>`. Dans le cas d'une absence de confirmation ou d'une confirmation tardive (hors délai ou délai non permis), le processus BPEL termine son exécution. S'il reçoit une confirmation dans les bons délais, et si les valeurs de l'ensemble de corrélation contenues dans le message reçu, par l'élément <onMessage> de <pick>, correspondent aux valeurs courantes de la corrélation `loanIdentifier`, `loanApprover` envoie son accord de prêt au client en utilisant une activité `<reply>`. Dans le cas contraire, l'activité `<exit>` est utilisée pour interrompre l'exécution du `loanApproval`.

6.4.2 Transformation de la description BPEL du `loanApproval` en IF

La description du `loanApproval` séquentiel, la description de son interface ainsi que celui de ses partenaires sont fournies en entrée à l'outil de transformation BPEL2IF. La spécification résultante de cette transformation contient 18 processus IF : un processus IF décrivant l'élément `<proces>` du processus `loanApproval`, un processus IF pour chacune de ses 16 sous activités, et un processus IF intermédiaire `interProcess` — qui a été décrit dans la Section 4.3.13. Cette spécification contient de la déclaration de : (i) 3 *signaux* internes (i.e. `done`, `fault` et `terminate`), (ii) 8 *signaux* décrivant les messages BPEL échangés entre le service `loanApproval`, son client et ses deux partenaires, (iii) 8 *signalroutes* décrivant les liens partenaires utilisés par le service composé pour interagir avec son client et ses deux partenaires.

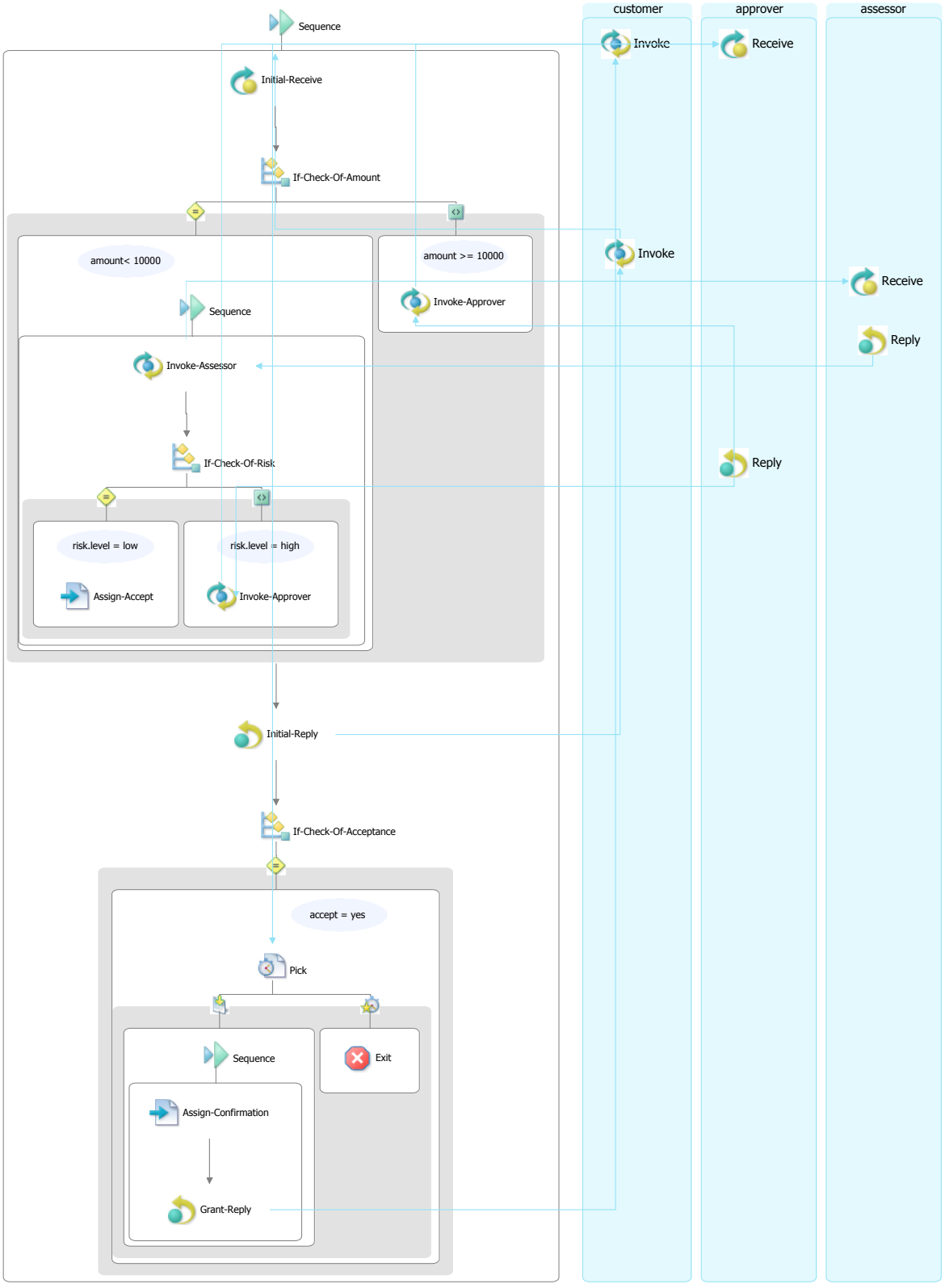

FIGURE 6.9 – flot du service `loanApproval` — variante séquentielle

Le client du `loanApproval`, et les deux partenaires `loanAssessor` et `loanApprover` sont décrits par l'environnement IF (i.e. `env`) qui communique avec les processus IF des activités de communications de BPEL par le biais du processus `interProcess`. Quelques métriques de la spécification IF du service `loanApproval` sont fournies dans la Table 6.1.

Lignes	1010	Processus	18
Signalroutes	8	Signaux	13
États	44	Transitions	96
Horloges	1	Variables Globales	10

TABLE 6.1 – Quelques métriques de la spécification IF du service `loanApproval`

Les types de données sont extraits à partir de la transformation des données WSDL (schémas XML) du client de `loanApproval` et de ses partenaires. Dans notre cas d'étude, nous réduirons les type `String` et `Integer` de BPEL à un intervalle de taille très réduite car deux valeurs ou trois valeurs sont suffisantes pour la génération de cas de test couvrant tous les scenarios de test possibles. Pour exemple, deux valeurs entières du montant du prêt (e.g. `6000` et `12000`), dont une est inférieure à `10000` et l'autre supérieure à `10000`, suffisent pour tester les deux cas possibles. Ajoutant à cela, que deux valeurs de `String` (e.g. `nom1` et `nom2`) suffisent de leur côté pour tester la corrélation des messages qui est définie sur les noms et prénoms du client.

6.4.3 Vérification de la spécification du `loanApproval`

La vérification consiste à établir l'exactitude d'une spécification et/ou d'une implantation d'un système. Rappelons que dans notre approche de test, nous considérons la description BPEL d'une composition de services comme étant une spécification de référence. Avant d'appliquer notre méthode de test à base de modèle formel, nous devons nous assurer que cette spécification vérifie bien les exigences fonctionnelles (et temporelles) de la composition. Cependant, BPEL est un langage exécutable basé sur XML, et par conséquent, nous ne pouvons pas appliquer les méthodes formelles pour la vérification de telles descriptions BPEL. Étant donné que nous avons proposé une modélisation formelle de la sémantique de BPEL et que nous avons par la suite transformé une composition BPEL en une spécification formelle en IF, nous pouvons alors appliquer une méthode de vérification formelle à cette spécification IF. Pour cela, nous utilisons la boîte à outil de IF qui offre une technique de vérification basée sur les observateurs.

Dans cette section, nous décrirons les propriétés attendues (ou à vérifier) par une composition de services. Nous monterons ensuite, à l'aide d'un exemple de vérification formelle d'une propriété, l'application d'une technique de vérification par observateurs à la spécification IF du service composé `loanApproval`. Cette phase de vérification de la spécification du service de prêt `loanApproval` sera suivie dans la suite par une phase de génération de tests temporisés.

Propriétés d'une composition de services Web

Dans la littérature, nous distinguerons quelques types de propriétés que nous pouvons vérifier sur une spécification formelle :

propriété de sûreté qui exprime qu'un événement, sous certaines conditions, ne peut jamais se produire ;

propriété de vivacité qui exprime qu'un événement, sous certaines conditions, finira par se produire ;

propriété de non blocage qui exprime que le système ne se trouvera jamais dans une situation où il ne peut plus progresser ;

propriété d'atteignabilité qui indique qu'un état du système peut être atteint.

Concernant la composition de services Web, les propriétés attendues peuvent être fournies (informellement) ou extraites à partir de la spécification de référence de la composition de services : la description BPEL du processus de composition et celles des interfaces WSDL des services partenaires. Ce sont des propriétés comportementales liées à la composition et qui sont définies en termes :

i d'interactions du service composé avec son client et ses différents partenaires : invocation des partenaires sous certaines conditions, envoi de réponse au client ;

ii de propriétés temporelles du service composé : temps d'attente, timeout.

Dans la section suivante, nous décrivons la vérification des propriétés dans la boîte à outils IF à l'aide de processus observateurs.

Observateurs IF

Dans la boîte à outils du langage IF [159], plus précisément de son extension IFx [185], il est possible de vérifier les propriétés d'un système (ou spécification) en utilisant des observateurs représentés en IF par des processus, et donc par des automates temporisés étendus (cf. § 4.2.2). Ces processus observateurs sont reliés au reste du système par des interactions synchrones, et exécutés en parallèle à ce système. Ils ont toujours la priorité la plus élevée pendant l'exploration du système ; ainsi l'observation d'un événement est déclenchée immédiatement lorsque cet événement se produit. Les processus observateurs sont composés de manière synchrone avec le système. Ils prennent le contrôle après chaque exécution d'une transition atomique de ce système. Ainsi, selon les événements produits par ce système et/ou les conditions vérifiées, les observateurs peuvent exécuter (en run-to-completion) plusieurs actions.

Les propriétés peuvent être décrites à l'aide d'une syntaxe spécifique des observateurs de IF. Ces derniers permettent ainsi l'observation des événements et des états du système (incluant les variables et les horloges), l'arrêt de la génération d'états non pertinents, et la coupure de chemins d'exécution. Ces observateurs utilisent des actions observables du système (e.g. envoi/réception de signaux), et des conditions portant sur les horloges ou les variables.

Après avoir décrit le processus observateur d'une propriété, le simulateur de IF effectue une exploration exhaustive de l'espace d'états du système composé (résultant de la composition synchrone du processus observateur et du système observé). Durant cette simulation, et à chaque étape, le processus observateur vérifie s'il peut observer le comportement attendu. Les processus observateurs de IF sont classés en trois types :

- Observateur *pure* (pure observer), qui décrit une propriété ;
- observateur *de coupure* (cut observer), qui peut guider la simulation du système par la coupure des chemins d'exécution ;
- observateur *intrusif* (intrusive observer), qui peut modifier le comportement du système par l'injection de signaux et la modification des valeurs de données.

Les états d'un processus observateur IF peuvent être de trois types : état *ordinaire* (ordinary state), état *échec* (error state) et état *succès* (success state). Une propriété est alors satisfaite si et seulement si l'état échec du processus observateur est non atteignable durant l'exploration du système composé.

Propriétés attendues du service `loanApproval`

Nous donnons ci-après quelques exemples des propriétés attendues du service de prêt `loanApproval`. Ces propriétés comportementales peuvent être décrites par des propriétés de sûreté, vivacité (bornée) ou de non blocage.

Exemples de propriété de sûreté :

`Prop 1` — Le service `loanApproval` ne doit pas invoquer son service partenaire `loanAssessor` (pour une évaluation de risqué lié au demandeur) avant la réception d'une demande de prêt d'un montant inférieur à `10000` ;

`Prop 2` — Le service `loanApproval` ne doit pas invoquer son service partenaire `loanAssessor` si le montant de la demande du prêt est supérieur à `10000` ;

`Prop 3` — Le service `loanApproval` ne doit pas attendre la confirmation d'une demande de prêt plus que `4` unités de temps ;

`Prop 4` — Le service `loanApproval` ne doit pas accepter une demande de prêt sans consulter le service `loanApprover` si le risque lié au demandeur est de haut niveau ;

Exemples de propriété de vivacité ou de non blocage :

`Prop 5` — A chaque demande de prêt, une décision sera envoyée au demandeur (acceptation ou refus de la demande) ;

`Prop 6` — Le service `loanApproval` doit terminer inévitablement ;

`Prop 7` — Le service loanApproval ne restera pas bloqué à attendre la confirmation du client de sa demande de prêt.

Dans la suite, nous allons expliciter la vérification de la deuxième propriété `Prop 2` sur la spécification IF du service `loanApproval` (cf. § 6.4.2). Pour le reste des propriétés décrites ci-dessus, la démarche est la même et nous les avons vérifié pour l'exemple que nous avons considéré (i.e. service du prêt).

Exemple de vérification de propriétés du service loanApproval

La propriété attendue Prop 2 est décrite par un processus *observateur pure* obs_2 dont le code IF est fournit par la Figure 6.10. Ce processus observateur contient 5 états :

- un état initial idle, permettant de vérifier la réception d'une demande de prêt (cf. Lignes 13–16) ;
- un état input_matched, vérifiant si le montant de la demande de prêt est supérieur ou pas à 10000 (cf. Lignes 18–21) ;
- un état check_output, vérifiant l'invocation : soit du service loanAssessor (cf. Ligne 25) ou du service loanApprover (cf. Ligne 28) ;
- un état de décision decision, pouvant soit mettre fin à la simulation du système en cas de violation de la propriété (cf. Lignes 36–38), soit initier une nouvelle vérification (cf. Lignes 39–41) ou attendre l'invocation d'un des deux services partenaires (cf. Lignes 42–43) ;
- état d'échec err, notifiant un échec de simulation et arrêtant la simulation du système composé.

La vérification de cette propriété Prop 2 à l'aide du simulateur IFx [185] passe par deux étapes :

i La génération du code du simulateur loan.x du système composé de la spécification IF du service loanApproval (fournie par le document loan.if) et de l'observateur obs_2 (décrit dans le document prop2.oif) à l'aide de la commande "if2gen -obs prop2.oif loan.if" ;

ii L'exécution du simulateur du système composé (avec une stratégie en largeur d'abord par exemple) à l'aide de la commande "./loan.x -bfs -q states -t transitions".

La simulation du système composé, et donc la vérification de la propriété Prop 2, ne produit aucun échec. Le simulateur IFx termine alors complètement l'exploration de l'espace d'états du système composé en explorant 10960 états et en franchisant 21331 transitions. La Table 6.2 explicite quelques métriques de cette simulation.

Propriété	Nombre d'états	Nombre de transitions	Durée de simulation (sec)	Résultat
Prop 2	10960	21331	1	Validée

TABLE 6.2 – Quelques métriques de la vérification de la propriété Prop 2 du service de loanApproval

Après avoir abordé la vérification de la composition du service loanApproval, nous allons expliciter dans la section suivante la génération de tests pour quelques scénarios.

6.4.4 Génération de tests abstraits avec TestGen-IF

Pour le test du service loanApproval, nous avons distingués 17 scénarios possibles incluant le test de corrélation, l'absence de message de confirmation du prêt ou son envoi hors délai ainsi que les autres scénarios relatifs au montant du prêt (inférieur ou supérieur à 10000), au niveau du risque lié au client (faible, moyen/haut) et à la décision du service loanApprover (i.e. yes ou no). Nous résumons dans la Table 6.3 ces différents scénarios.

```
/* Signals of the IF specification*/
signal creditInformationMessage1 (pl_type_customer_request,creditInformationMessageType);
signal creditInformationMessage2 (pl_type_assessor,creditInformationMessageType);
signal creditInformationMessage3 (pl_type_approver,creditInformationMessageType);

pure observer ob_2;
  var pl_type pl_type_customer_request;
  var loan_request creditInformationMessageType;
  var output_message2_matched boolean;
  var output_message3_matched boolean;
  var so t_if_signal;

  state idle #start ;
    match input creditInformationMessage1(pl_type,loan_request);
      nextstate input_matched;
  endstate;

  state input_matched;
    provided loan_request.amount >= 10000;
        nextstate check_output;
  endstate;

  state check_output;
    match output (so);
      if so instanceof creditInformationMessage2 then
        task output_message2_matched := true;
      else
        if so instanceof creditInformationMessage3 then
           task output_message3_matched := true;
        endif
      endif
        nextstate decision;
  endstate;

  state decision #unstable ;
    provided (output_message2_matched = true);
      informal "--Validation Fail!";
        nextstate err;
    provided (output_message3_matched = true);
      informal "--Validation Success!";
        nextstate idle;
    provided (output_message2_matched = false and output_message3_matched = false);
      nextstate check_output;
  endstate;

  state err #error ;
  endstate;
endobserver;
```

FIGURE 6.10 – Processus observateur IF décrivant la propriété `Prop 2` du service `loanApproval`

Dans notre étude de cas, nous avons appliqué notre approche de test au service de prêt `loanApproval` en prenant en compte les 17 scenarios décrits ci-dessus. Dans la suite de ce chapitre, pour illustrer notre cas d'étude, nous avons choisi les trois scénarios 1, 8 et 16. Le premier scénario permet de tester quelques interactions du service `loanApproval` avec ses différents partenaires. Le scénario 8 sert à tester la gestion de la corrélation des messages alors que le scénario 16 est utilisé pour tester la gestion du timeout (délai d'attente d'un message de confirmation de la demande de la part du client). Pour ces trois scenarios, nous décrirons la génération de test abstraits avec l'outil TestGen-IF, y compris la formulation des objectifs de test, la concrétisation de ces tests abstraits, et enfin l'exécution des tests résultants avec la plateforme BPELUnit.

Dans la suite, nous décrirons en premier les trois scénarios 1, 8 et 16 (cf. Table 6.3). Ensuite, nous détaillerons, pour ces trois scénarios, la formulation des objectifs de test ainsi que la génération des cas de test temporisés.

N° scénario	Montant du prêt	Niveau du risque	Décision du loanApprover	État de la corrélation	Absence de message ou Envoi hors délai	Exécution attendue du loanApproval
1	6000	faible	-	correct	-	complète
2	6000	faible	-	violation	-	interrompue
3	6000	faible	-	violation puis correct	-	complète
4	6000	faible	-	-	absence	interrompue
5	6000	faible	-	-	hors délai	interrompue
6	6000	haut	yes	correct	-	complète
7	6000	haut	yes	violation	-	interrompue
8	6000	haut	yes	violation puis correct	-	complète
9	6000	haut	yes	-	absence	interrompue
10	6000	haut	yes	-	hors délai	interrompue
11	6000	haut	no	-	-	complète
12	12000	-	yes	correct	-	complète
13	12000	-	yes	violation	-	interrompue
14	12000	-	yes	violation après correct	-	complète
15	12000	-	yes	-	absence	interrompue
16	12000	-	yes	-	hors délai	interrompue
17	12000	-	no	-	-	complète

TABLE 6.3 – 17 Scénarios de test pour le service loanApproval

Description des trois scénarios

Scénario 1 Un client (e.g. Mounir Lallali) envoie une demande de prêt d'un montant de 6000 au service loanApproval. Comme ce montant est inférieur à 10000, loanApproval invoque son service partenaire loanAssessor pour évaluer le risque lié au client Lallali. loanAssessor répond au service de prêt que ce risque est faible. Par conséquent, le prêt est accordé au client qui est informé par l'envoi d'un message de la part du service de prêt. Le client envoie instantanément une confirmation de sa demande de prêt contenant les bons renseignements à savoir nom et prénom (i.e. Lallali et Mounir) au loanAssessor, qui à son tour lui répond par un message de validation de son prêt.

Scénario 8 Un client (e.g. Mounir Lallali) envoie une demande de prêt d'un montant de 6000 au service loanApproval. Comme ce montant est inférieur à 10000, loanApproval invoque son service partenaire loanAssessor pour évaluer le risque lié au client Lallali. loanAssessor répond au service de prêt que ce risque est de niveau haut. Par conséquent, le service de prêt invoque son deuxième partenaire loanApprover qui doit prendre la décision d'accorder ou pas le prêt au client Lallali. loanApprover répond positivement au service de prêt qui transmet

cette décision au client. Ce dernier envoie à son tour un message de confirmation de sa demande de prêt mais après 2 unités de temps. Entre temps, `loanApproval` a reçu un autre message de confirmation d'une demande de prêt mais avec des données non attendues et donc erronées, c.-à-d. nom et prénom différents de ceux du client Mounir Lallali (e.g. Fatiha Zaidi).

Scénario 16 Un client (e.g. `Mounir Lallali`) envoie une demande de prêt d'un montant de 12000 au service `loanApproval`. Vu que le montant est supérieur à 10000, `loanApproval` invoque directement son partenaire `loanApprover` sans aucune évaluation de risque. C'est à `loanApprover` de prendre la décision d'accorder ou pas le prêt au client `Lallali`. `loanApprover` répond positivement au service de prêt qui transmet cette décision au client. Ce dernier envoie à son tour un message de confirmation de sa demande de prêt mais après 6 unités de temps, et donc hors délai.

Formulation des objectifs de test pour les trois scénarios

Nous décrirons, dans cette section, la formulation des objectifs de test pour les trois scénarios.

Objectifs de test pour le scénario 1 L'ensemble des objectifs de test associé au scénario 1, noté par OBJ_1, est décrit formellement dans la Figure 6.11. C'est un ensemble ordonné composé de quatre objectifs. Les trois premiers objectifs (obj_1, obj_2 et obj_3) désignent un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 6000, de l'évaluation du risque faible par le service partenaire `loanAssessor` et de la confirmation immédiate (la valeur d'horloge `c34 = 0`) de la demande de prêt de `Mounir LALLAI` avec les bons renseignements. Le dernier objectif obj_4 désigne la réception du client de la validation du prêt.

Objectifs de test pour le scénario 8 L'ensemble des objectifs de test associé au scénario 8, noté par OBJ_8, est décrit formellement dans la Figure 6.12 ci-dessous. C'est un ensemble ordonné composé de six objectifs. Les cinq premiers objectifs (obj_1, obj_2, obj_3, obj_4 et obj_5) désignent un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 6000, de l'évaluation du risque de niveau haut par le service partenaire `loanAssessor`, de l'accord du prêt par le service `loanApprover`, de la confirmation immédiate (la valeur d'horloge `c34 = 0`) de la demande de prêt d'un autre client `Fatiha Zaidi`, en fin de la confirmation de la demande de prêt de `Mounir LALLAI` avec les bons renseignements mais une attente de 2 unités de temps (la valeur d'horloge `c34 = 2`). Le dernier objectif obj_6 désigne la réception du client de la validation du prêt.

Objectifs de test pour le scénario 16 L'ensemble des objectifs de test associé au scénario 16, noté par OBJ_16, est décrit formellement dans la Figure 6.13 ci-dessus. C'est un ensemble ordonné composé de trois objectifs (obj_1, obj_2 et obj_3) désignant un envoi de message au service de prêt, plus précisément, envoi de la demande de prêt de `Mounir LALLAI` avec un montant de 12000, de l'accord du prêt par le service `loanApprover`, et de la confirmation hors délai (la valeur d'horloge `c34 = 6`) de la demande de prêt de `Mounir LALLAI` avec les bons renseignements.

$$OBJ_1 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequenceinvoke20}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input riskAssessmentMessage{_,{low}}`

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 0`

$$obj_4 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$cond_1 = processus : instance =$ `{sequencereply43}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `output confirmationMessage`

FIGURE 6.11 – Objectifs de test du scénario 1

Génération de tests abstraits pour les trois scénarios de test

Les objectifs de test des trois scénarios (1, 8 et 16) ainsi que la spécification IF (résultante de la transformation du processus `loanApproval` et des interfaces WSDL) ont été fournis en entrée à l'outil TestGen-IF. En conséquence, trois cas de test ont été dérivés. Ils sont composés d'actions observables qui ont été décrites dans la Section 5.5.1 : (i) la réception d'un message (`?message`), (ii) l'émission d'un message (`!message`) (iii) l'écoulement du temps entre deux actions observables (`delay`), l'activation et la désactivation des horloges locales (actions `set` et `reset`). Ces trois cas test abstraits sont présentés, respectivement, par la Figure 6.14, la Figure 6.15 et la Figure 6.16.

Dans la Table 6.4, nous donnons quelques métriques concernant la génération de cas de test pour les trois scénarios.

$OBJ_8 = Obj_{ord} = \{obj_1, obj_2, obj_3, obj_4, obj_5, obj_6\}$

$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequencereceive14}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditInformationMessage1{_,{Mounir,Lallali,6000}}`

$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequenceinvoke20}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input riskAssessmentMessage{_,{high}}`

$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{elseinvoke30}0`
$cond_2 = état : source =$ `receive`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input approvalMessage{_,{yes}}`

$obj_4 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Fatiha,Zaidi}}`
$cond_5 = horloge :$ `c34 = 0`

$obj_5 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$
$cond_1 = processus : instance =$ `{ifpick34}0`
$cond_2 = état : source =$ `internalState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `input creditConfirmationMessage{_,{Mounir,Lallali}}`
$cond_5 = horloge :$ `c34 = 2`

$obj_6 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$
$cond_1 = processus : instance =$ `{sequencereply43}0`
$cond_2 = état : source =$ `inState`
$cond_3 = état : destination =$ `outState`
$cond_4 = action :$ `output confirmationMessage`

FIGURE 6.12 – Objectifs de test du scénario 8

$$OBJ_16 = Obj_{ord} = \{obj_1, obj_2, obj_3\}$$

$$obj_1 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$$cond_1 = processus : instance = \texttt{\{sequencereceive14\}0}$$
$$cond_2 = état : source = \texttt{inState}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input creditInformationMessage1\{\_,\{Mounir,Lallali,12000\}\}}$$

$$obj_2 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4$$
$$cond_1 = processus : instance = \texttt{\{elseinvoke28\}0}$$
$$cond_2 = état : source = \texttt{receive}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input approvalMessage\{\_,\{yes\}\}}$$

$$obj_3 = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \wedge cond_5$$
$$cond_1 = processus : instance = \texttt{\{ifpick34\}0}$$
$$cond_2 = état : source = \texttt{internalState}$$
$$cond_3 = état : destination = \texttt{outState}$$
$$cond_4 = action : \texttt{input creditConfirmationMessage\{\_,\{Mounir,Lallali\}\}}$$
$$cond_5 = horloge : \texttt{c34 = 6}$$

FIGURE 6.13 – Objectifs de test du scénario 16

```
1 ?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
2 !creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
3 ?riskAssessmentMessage{assessor_riskAssessmentPT_check,{low}}
4 !approvalMessage2{customer_loanServicePT_request,{yes}}
5 set c34:=0
6 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
7 reset c34;
8 !confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

FIGURE 6.14 – Cas de test du scénario 1

N° du scénario	Longueur du cas de test	Stratégie d'exploration	Profondeur limite	Nombre d'objectifs de test	États visités	Durée de génération (s)	Nombre de sauts
1	8	BFS	30	4	398	8	0
8	12	BFS	30	6	662	13	0
16	4	BFS	30	3	1140	52	0

TABLE 6.4 – Quelques métriques de la génération de test pour les trois scénarios

Après avoir formulé les objectifs de test et générer les cas de test abstraits pour les trois scénarios, nous allons décrire, dans la section suivante, la concrétisation de ces cas de test selon notre méthode qui a été décrite dans la Section 5.6, mais en tenant en compte de la syntaxe de description des suites de test de la plateforme BPELUnit [106] présentée dans la Section 6.3.2.

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
?riskAssessmentMessage{assessor_riskAssessmentPT_check,{high}}
!creditInformationMessage3{approver_loanApprovalPT_approve,{Mounir,Lallali,6000}}
?approvalMessage{approver_loanApprovalPT_approve,{yes}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
set c34
?creditConfirmationMessage{customer_loanServicePT_obtain,{Fatiha,Zaidi}}
delay=2
?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
reset c34
!confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

FIGURE 6.15 – Cas de test du scénario 8

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,12000}}
!creditInformationMessage3{approver_loanApprovalPT_approve,{Mounir,Lallali,12000}}
?approvalMessage{approver_loanApprovalPT_approve,{yes}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
set c34:=0
delay=5
reset c34
```

FIGURE 6.16 – Cas de test du scénario 16

6.4.5 Dérivation des tests concrets

La dérivation de tests concrets consiste à transformer un cas de test abstrait en un test décrivant les comportements des services testeurs (qui simulent le client et les partenaires du service sous test), les messages d'entrée pour ce service sous test ainsi que les messages de sortie attendus. Dans cette section, nous présentons la concrétisation des deux cas de test abstraits du scénario 1 et 8. Les cas de test concrets de ces deux scénarios sont donnés, respectivement, en Annexe E et Annexe F.

Concrétisation du cas de test abstrait du scénario 1

Pour la concrétisation du cas de test abstrait du scénario 1, nous procédons de la manière suivante. En analysant les paramètres des signaux du cas de test 1 (cf. Fig. 6.14), en particulier des liens partenaires qui sont donnés par le premier champs du premier paramètre de chaque signal et qu'utilisent le service composé `loanApproval` pour interagir avec son client et ses partenaires (i.e. `customer` et `assessor`), nous pouvons constater que seuls deux services testeurs sont nécessaires pour la concrétisation de ce cas de test, à savoir, le service `client` et le service partenaire `loanAssessor`. Nous utilisons alors deux Tracks de BPELUnit : Track `client` et Track `assessorPartner`, pour représenter ces deux services testeurs.

Pour la concrétisation des interactions des testeurs (i.e. activités des Tracks de BPELUnit), nous regroupons les actions de communication (! et ?) selon les liens partenaires. On aura alors, quatre actions pour le Track `client` et deux actions pour le Track `loanAssessor` :

```
Track client :
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
!confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}

Track assessorPartner :
!creditInformationMessage2{assessor_riskAssessmentPT_check,{Mounir,Lallali,6000}}
?riskAssessmentMessage{assessor_riskAssessmentPT_check,{low}}
```

Notons que chaque paire d'actions ont en commun les mêmes liens partenaires, types de port et opérations. Ainsi, nous pouvons déduire que chaque paire d'actions représente une interaction bilatérale synchrone et qui sera transformée en l'une des deux activités synchrones de BPELUnit : `Send/Receive Synchronous` ou `Receive/Send Synchronous`.

Pour exemple, la paire suivante :

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
!approvalMessage2{customer_loanServicePT_request,{yes}}
```

est transformée en une activité `Send/Receive Synchronous`. Une action de réception est transformée en une activité `Send` de BPELUnit alors qu'une action d'envoi est transformée en une activité `Receive` de BPELUnit. Le Track `client` sera alors composé de deux activités `Send/Receive Synchronous`, respectivement, le Track assessorPartner sera composé d'une seule activité `Receive/Send Synchronous`.

Concernant la dérivation des données échangées (entrée/sortie), nous combinons les valeurs des paramètres des signaux figurant dans les cas de test avec les différents types de données définis dans les descriptions WSDL du client et des partenaires du service sous test. Cela, nous permet d'extraire un arbre XML représentant une donnée du test où chaque feuille est associée à une valeur.

Pour générer les données XML envoyées au service sous test par le biais de l'activité `Send` de BPELUnit résultant de la dérivation de l'action de réception suivante :

```
?creditInformationMessage1{customer_loanServicePT_request,{Mounir,Lallali,6000}}
```

nous combinons les valeurs `{Mounir,Lallali,6000}` avec le type de données du message d'entrée (input) de l'opération `request` décrit ci-dessous :

```
<message name="creditInformationMessage">
    <part name="firstName" type=" xsd:string " />
    <part name="name" type="xsd:string" />
    <part name="amount" type="xsd:integer" />
</message>
<operation name="request">
    <input message=" tns:creditInformationMessage " />
    <output message="tns:approvalMessage" />
    <fault name="unableToHandleRequest" message="tns:errorMessage" />
</operation>
```

Ce, qui nous permet de générer les données XML suivantes qui seront fournies à une action `Send` de BPELUnit :

```
1 <data>
2     <firstName>Mounir</firstName>
3     <name>Lallali</name>
4     <amount>6000</amount>
5 <data>
```

D'un autre côté, pour vérifier si les données attendues, nous pouvons associer une activité `Receive` de BPELUnit à une condition de vérification. Ces données sont extraites, de la même façon que les données d'entrée, mais à partir des paramètres des actions d'envoi (`!message`) appartenant au cas de test abstrait.

Pour le cas de test du scénario 1, nous pouvons vérifier si le Track `client` a bien reçu une décision favorable à sa demande de prêt comme c'est indiqué dans le cas de test du scénario 1 (cf. Ligne 4, Figure 6.14). Pour cela, l'activité `Receive` du Track `client`, permettant de recevoir la décision du `loanApproval`, est associée à la condition suivante (cf. Ligne 23, Annexe E) :

```
1 <receive>
2 <condition>
3 <expression >accept</expression>
4 <value>'yes'</value>
5 </condition>
6 </receive>
```

Enfin, Le test concret dérivé du cas de test abstrait du scénario 1 est créé en utilisant l'éditeur de BPELUnit (cf. Fig. 6.17). Il est donné en décrit en Annexe E.

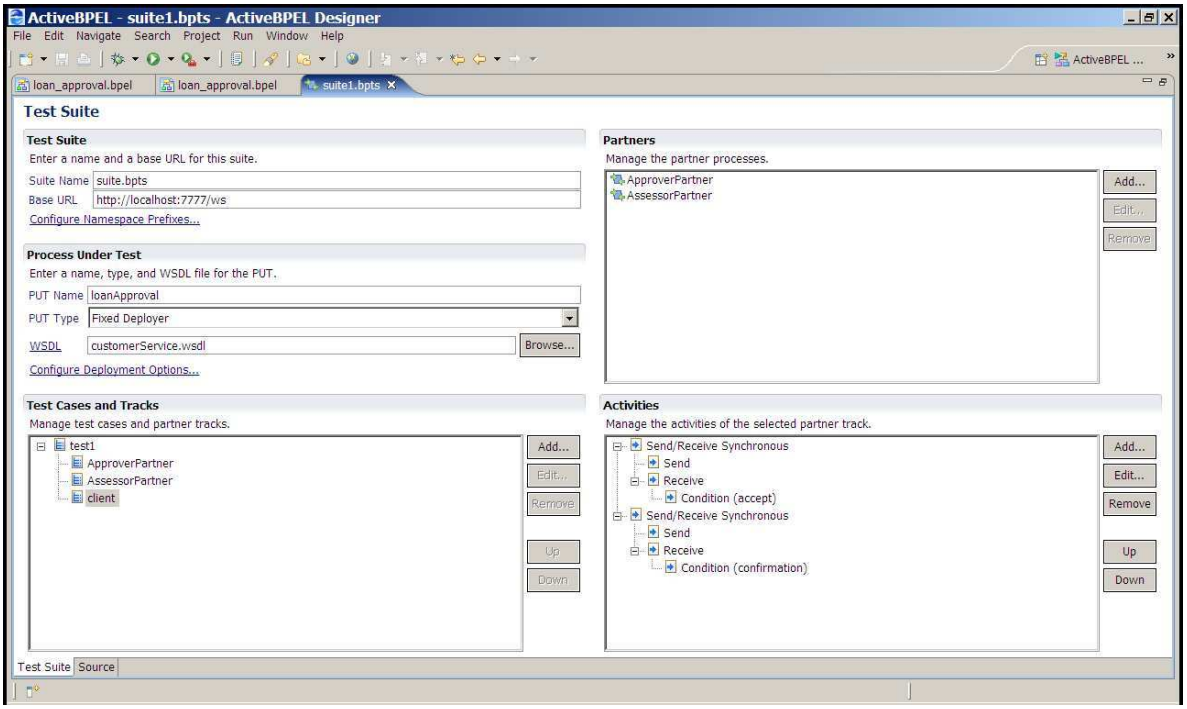

FIGURE 6.17 – L'interface de BPELUnit sous activeBPEL Designer

Concrétisation du cas de test abstrait du scénario 8

Pour la concrétisation du cas de test du scénario 8 (décrit en Figure 6.15), nous procédons de la même manière que pour la concrétisation de celui du scénario 1. Cependant, le cas de test du scénario 8 contient un intervalle de temps de 2 unités. En conséquence, nous appliquons notre méthode de concrétisation des délais d'attente que nous avons définie dans la Section 5.6.1. Notons que dans la Figure, l'action de désactivation de l'horloge c34 (i.e. reset c34) est précédée, dans le cas de test, par une action de réception comme suit :

```
1 set c34:=0
2 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Fatiha,Zaidi}}
3 delay=2
4 ?creditConfirmationMessage{customer_loanServicePT_obtain,{Mounir,Lallali}}
5 reset c34
6 !confirmationMessage{customer_loanServicePT_obtain,{LoanConfirmation}}
```

Pour cette raison, `delay=2` détermine un délai d'attente du Track `client` de 2 unités de temps avant l'envoi du message `creditConfirmationMessage` par une activité `Send`. Dans BPELUnit, nous pouvons différer l'envoi d'un message en utilisant la paramètre `Send Delay` associé à chaque activité `Send`.

Le test concret du scénario 8, édité avec l'interface de BPELUnit, est détaillé en Annexe F.

6.4.6 Exécution des tests avec BPELUnit

Dans cette section, nous présentons l'exécution des cas de test concrets avec la plateforme BPELUnit. Pour cela, nous avons déployé la variante parallèle du service de prêt `loanApproval` — dont le flot de contrôle ainsi que le code BPEL sont donnés respectivement en Annexe B et Annexe C — sous le moteur activeBPEL utilisant le serveur d'applications Tomcat. Notons que les cas de test abstraits qui ont servi à la concrétisation des tests exécutables par BPELUnit, ont été dérivés à partir de la spécification IF de la variante séquentielle du service `loanApproval` qui a été présentée dans la Section 6.4.1 et dont le code BPEL est donné en Annexe C.

Phase d'édition d'une suite de test

Nous pouvons éditer les tests concrets et exécutables à l'aide de l'interface de BPELUnit (cf. Fig. 6.17 ci-dessus). Cette interface permet :

- la saisie des informations liées au service sous test (cf. Fig. 6.18) ;
- la sélection des partenaires interagissant dans le test avec le service sous test (cf. Fig. 6.19) ;
- l'édition des cas de test (cf. Fig. 6.20) ;
- la saisie des caractéristiques d'une suite de test (cf. Fig. 6.21) ;
- l'édition des activités de chaque Track simulant un client ou un partenaire.

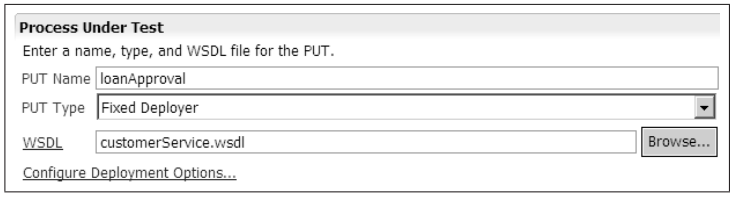

FIGURE 6.18 – Édition d'une suite de test dans BPELUnit — Service sous test

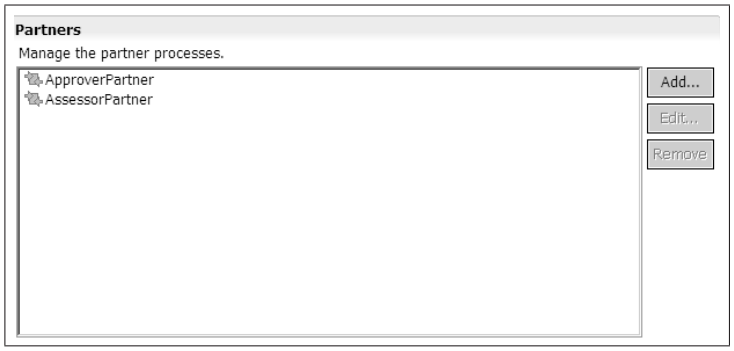

FIGURE 6.19 – Édition d'une suite de test dans BPELUnit — Sélection des partenaires

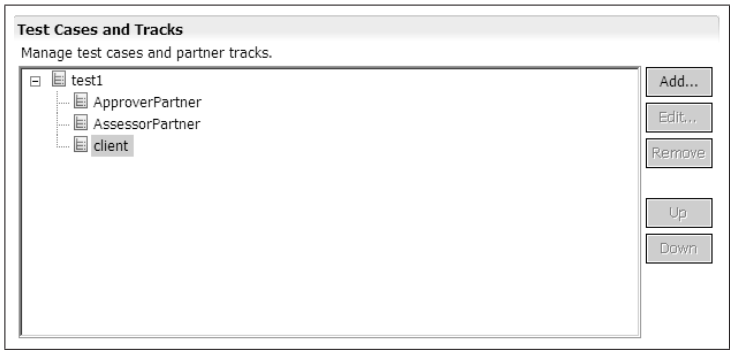

FIGURE 6.20 – Édition d'une suite de test dans BPELUnit — Édition de cas de test

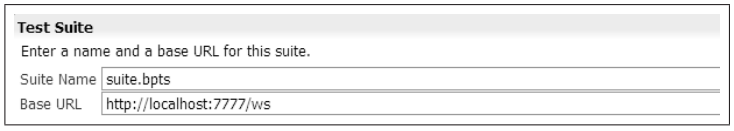

FIGURE 6.21 – Édition d'une suite de test dans BPELUnit — Caractéristiques d'une suite de test

La Figure 6.22 présente l'interface d'édition d'une activité `Send/Receive Synchronous` d'un Track `client` permettant de sélectionner le nom du port et de l'opération WSDL (utilisés pour invoquer le service composé déployé (`loanApproval`)), de saisir les données XML fournies en entrée au service `loanApproval`.

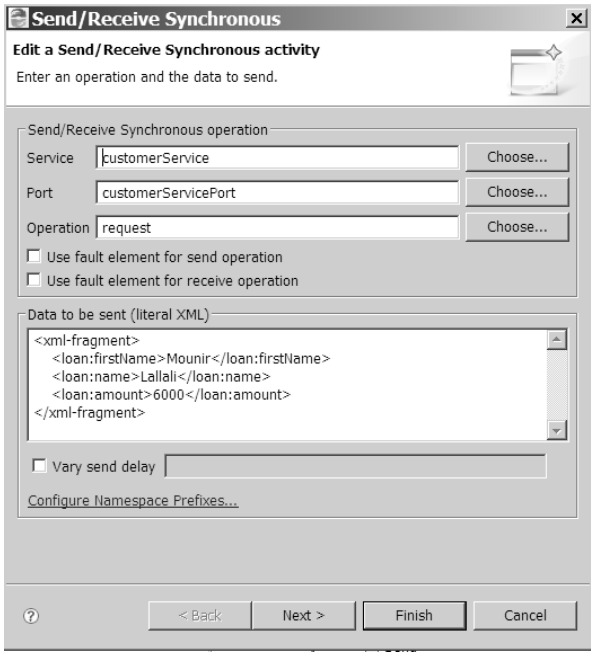

FIGURE 6.22 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Données XML envoyées par `Send`

La Figure 6.23 présente l'interface BPELUnit de saisie des conditions de vérification associée à une activité `Receive`.

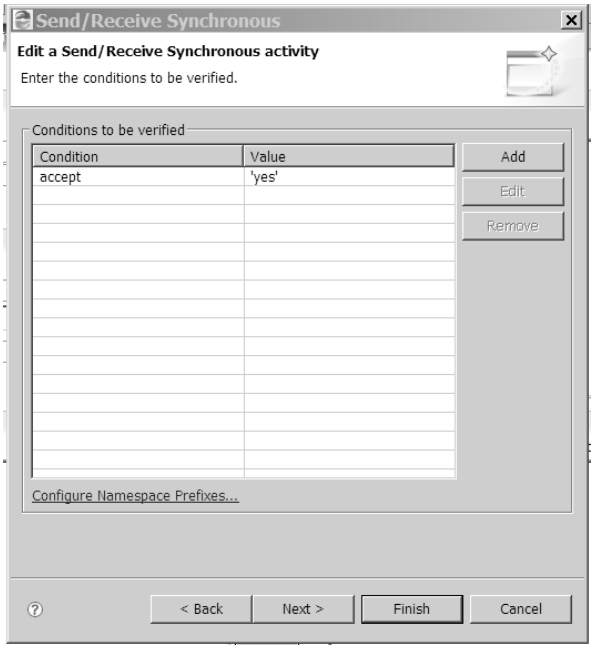

FIGURE 6.23 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Condition de vérification des données reçues par `Receive`

BPELUnit nous a servi pour créer et exécuter les tests concrets des 17 scénarios de test définis dans la Table 6.3. Ces tests concrets ont été dérivés à partir des cas de test abstraits qui ont été générés par l'outil TestGen-IF. Nous donnons comme exemple dans la section suivante, l'exécution du test concret du scénario 1.

Phase d'exécution de la suite de test

Nous exécutons directement les suites de test à partir de l'interface de BPELUnit. Nous pouvons ensuite visualiser les résultats du test, y compris, l'ensemble des activités effectuées, des données envoyées au service sous test ainsi que les données reçues par les Tracks (service testeurs simulant le client ou les partenaires du service sous test). La Figure 6.24 donne un aperçu des résultats de l'exécution du test du scénario 1.

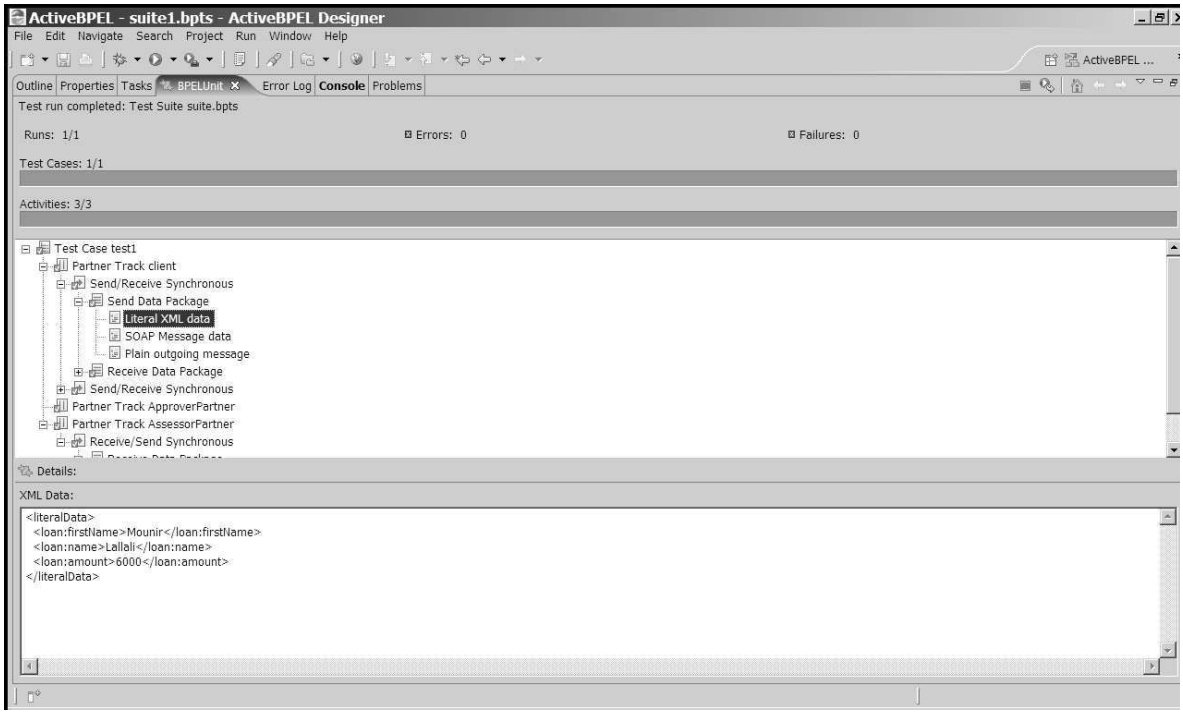

FIGURE 6.24 – Exécution d'une suite de test avec BPELUnit

D'un autre coté, l'interface d'administration des processus actifs de activeBPEL (cf. 6.25), permet la visualisation des instances qui ont été initiées par le Track `client` ainsi que toutes les informations concernant leurs exécutions. Nous l'avons utilisé pour vérifier l'exécution d'une instance du service `loanApproval` déployé (variante parallèle).

La Figure 6.26 donne l'aperçu des informations relatives au comportement du processus `loanApproval` en réaction à l'exécution par BPELUnit du cas de test du scénario 1. Ces informations sont fournies par l'interface activeBPEL.

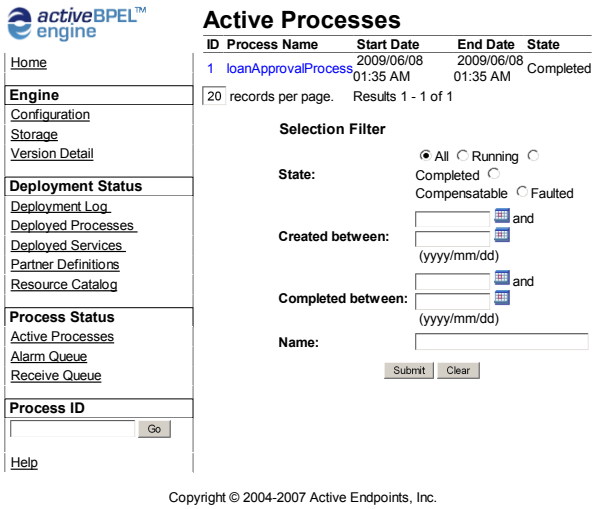

FIGURE 6.25 – Page d'administration de activeBPEL — Gestion des processus actifs

Phase d'émission du verdict

Le verdict de test dépend de l'exécution d'un cas de test avec BPELUnit, i.e. exécution de toutes les activités BPELUnit des Tracks simulant le client et les partenaires du service sous test. Une exécution attendue d'un cas de test engendre un verdict `Pass`. Dans le cas contraire, un verdict `Fail` est émis. Nous résumons, dans les Tables 6.5 et 6.6 et 6.7, l'exécution des cas de test des trois scénarios et les verdict associés.

Dans la Table 6.5, l'émission d'un verdict `Pass` pour l'exécution du cas de test 1 nécessite une exécution complète et réussie de toutes les activités des Tracks `client` et `assessorPartner` simulant, respectivement, le client et le service partenaire `loanAssessor` du service sous test (i.e. variante parallèle du service `loanApproval` décrite dans les Annexes B et C).

Track	Action BPELUnit	Exécution de l'action
`client`	`Send/Receive Synchronous` (1)	réussie
	`Send/Receive Synchronous` (2)	réussie
`assessorPartner`	`Receive/Send Synchronous`	réussie
Verdict		`Pass`

TABLE 6.5 – Exécution du test du scénario 1

Le cas de test 8 permet de tester la gestion de corrélation du service sous test. La Table 6.5 présente l'exécution de ce cas de test et le verdict émis en conséquence. Un verdict `Pass` est émis, si l'exécution de toutes les activités BPELUnit des Tracks `client`, `assessorPartner` et `approverPartner` est réussie à l'exception de la deuxième activité `Send/Receive Synchronous` du Track `client`. Cette dernière action synchrone envoie un message de confirmation de demande de prêt contenant des données erronées au service sous test qui ne les prend pas en compte. Par conséquent, ce service n'envoie aucun message de validation au client causant ainsi l'échec la deuxième action de réception de `Send/Receive Synchronous`. Au contraire, la troisième action `Send/Receive Synchronous` du Track `client` s'exécute sans aucun échec puisqu'elle envoie les données attendues par le service sous test.

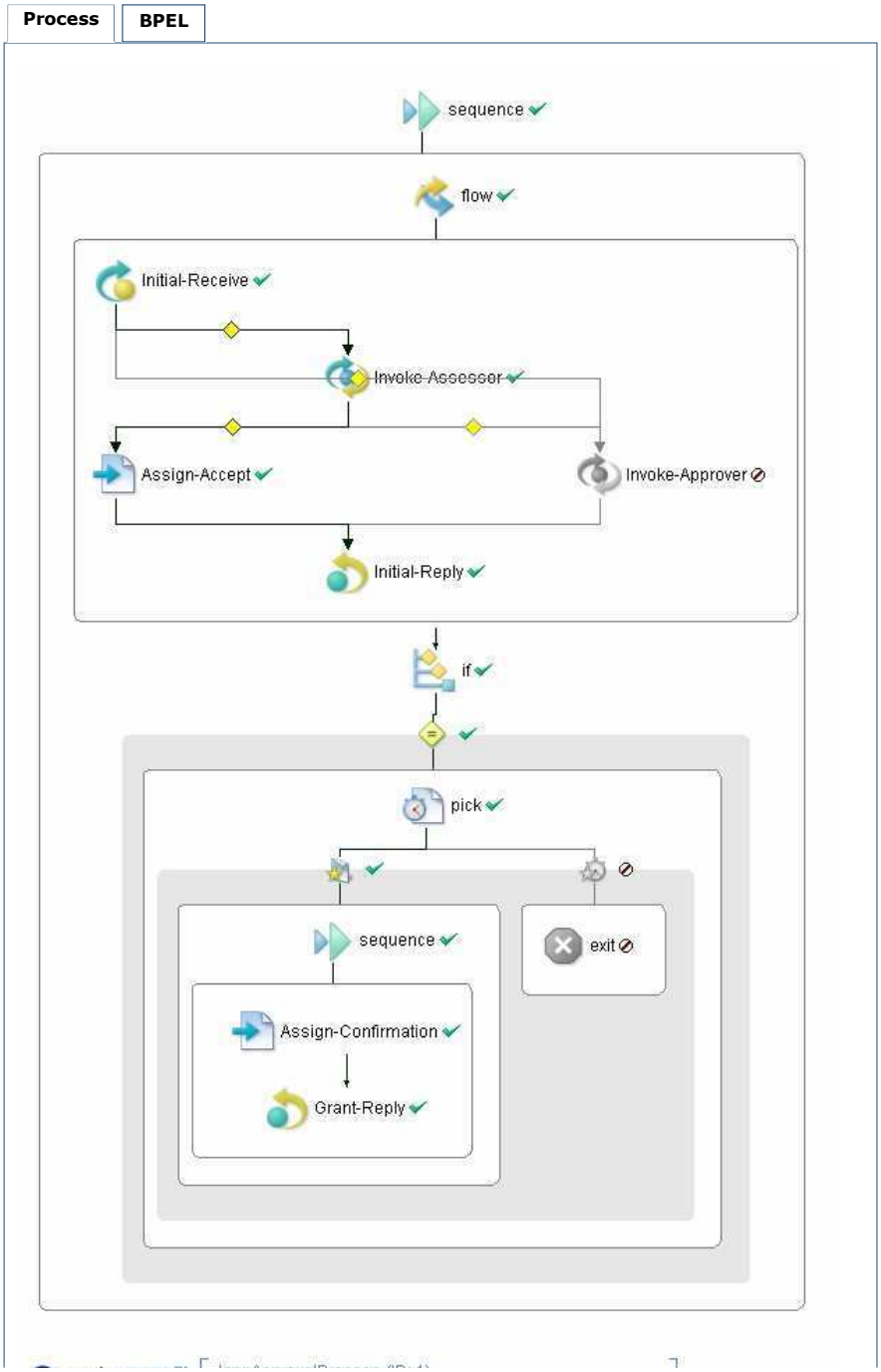

FIGURE 6.26 – Page d'administration de activeBPEL — Affichage de l'exécution du `loanApproval` sous test suite à la l'exécution du cas de test du scénario 1

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	échec de réception
	Send/Receive Synchronous (3)	réussie
assessorPartner	Receive/Send Synchronous	réussie
approverPartner	Receive/Send Synchronous	réussie
Verdict		Pass

TABLE 6.6 – Exécution du test du scénario 8

Le cas de test 16 permet, quant à lui, de tester la gestion des timeouts par le service sous test qui doit déclencher une alarme après une attente de 5 unités de temps d'un message de confirmation d'une demande de prêt (que devrait envoyée son client). L'exécution de ce cas est présentée dans la Table 6.7. Un verdict Pass est émis, si toutes les activités BPELUnit des deux Tracks client et approverPartner terminent leur exécution sans échec, à l'exception de l'activité Send/Receive Synchronous qui doit causer un échec de réception étant donné qu'elle envoie ses données hors délai (i.e. message de confirmation envoyé après 6 unités de temps).

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	échec de réception
approverPartner	Receive/Send Synchronous	réussie
Verdict		Pass

TABLE 6.7 – Exécution du test du scénario 16

Nous avons modifié le code BPEL du service sous test loanApproval en ignorant la gestion des corrélations de messages. Nous avons ensuite exécuté de nouveau le cas de test 8 après avoir déployé la nouvelle version du service sous test. L'exécution de ce cas de test, résumée dans la Table 6.8, n'a pas généré le résultat attendu. Cela s'explique par le fait que le message envoyé par la deuxième activité Send/Receive Synchronous du Track client, a été bien accepté par le service sous test malgré les données erronées qu'ils contiennent. Par conséquence, le verdict émis est Fail qui est dû à la non gestion des corrélations de la part du service sous test.

Track	Action BPELUnit	Exécution de l'action
client	Send/Receive Synchronous (1)	réussie
	Send/Receive Synchronous (2)	réussie
	Send/Receive Synchronous (3)	échec de réception
assessorPartner	Receive/Send Synchronous	réussie
approverPartner	Receive/Send Synchronous	réussie
Verdict		Fail

TABLE 6.8 – Nouvelle exécution du test du scénario 8

Dans la Table 6.9, nous résumons les informations concernant l'exécution des cas de test des 17 scénarios ainsi que les verdict de test associés.

N° cas de test	Exécution du cas de test	Exécution attendue du `loanApproval`	Verdict de test
1	succès	complète	`Pass`
2	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
3	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
4	succès	interrompue	`Pass`
5	le client ne reçoit aucun message de validation du prêt (message de confirmation envoyé hors délai)	interrompue	`Pass`
6	succès	complète	`Pass`
7	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
8	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
9	succès	interrompue	`Pass`
10	le client ne reçoit aucun message de validation du prêt (message de confirmation envoyé hors délai)	interrompue	Pass
11	succès	complète	`Pass`
12	succès	complète	`Pass`
13	le client envoyant des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	interrompue	`Pass`
14	le client envoyant, la première fois, des données erronées pour la corrélation ne reçoit aucun message de validation de prêt	complète	`Pass`
15	succès	interrompue	`Pass`
16	le client ne reçoit aucun message de validation du prêt (message envoyé hors délai)	interrompue	Pass
17	succès	complète	`Pass`

TABLE 6.9 – L'exécution des 17 tests du service `loanApproval`

6.5 Synthèse

Dans ce chapitre, nous avons présenté la plateforme de test qui met en œuvre notre approche de test de la composition de services. Cette plateforme permet la transformation d'une description BPEL en une spécification IF, la génération automatique de tests temporisés, et l'édition et l'exécution des tests concrets. Elle intègre les deux outils que nous avons développés : BPEL2IF (l'outil de transformation de BPEL en IF) et TestGen-IF (l'outil de génération de tests) ainsi que BPELUnit (le Framework d'édition et d'exécution de tests). Pour illustrer notre approche, nous avons présenté une étude de cas : le test du service composé `loanApporval`. Nous avons décrit toutes les phases d'application de notre approche de test au service `loanApproval` (modélisation, génération des tests abstraits, concrétisation des tests, édition et exécution des tests) ainsi que l'utilisation de la plateforme et de ses outils.

Dans la suite du document, nous allons résumer nos travaux effectués et les résultats obtenus, et donner quelques directions envisageables pour la poursuite de ces travaux.