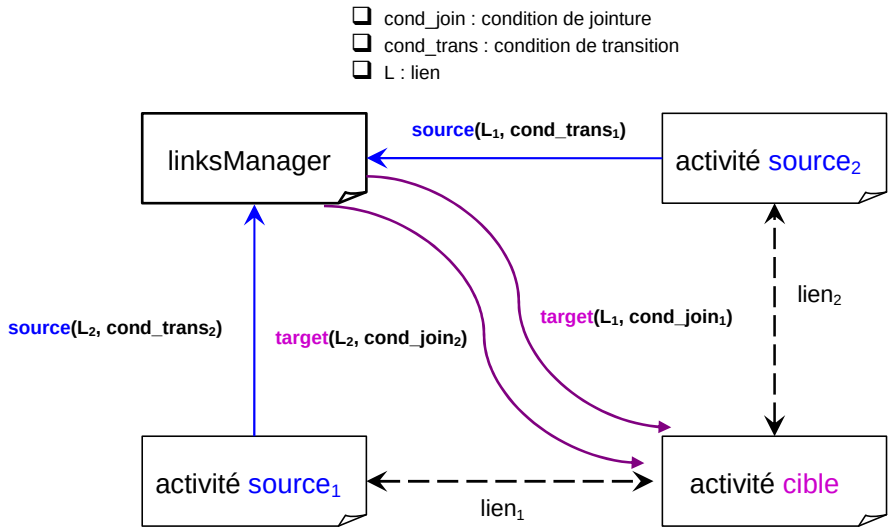

FIGURE 4.14 – Synchronisation des liens

Nous donnons ci-dessous un exemple de transformation d'une activité basique : `<exit>`.

Exemple 4.8 (Transformation d'une activité basique : `<exit>`)**.** L'activité `<exit>` est décrite en BPEL par :

```
<exit />
```

Elle est transformée en un processus IF non interruptible (cf. Fig. 4.15) ayant un seul état `inOutState` contenant une seule transition urgente qui assigne `true` à la variable `stopProcess` et envoie un signal `done` au processus parent. Le processus IF ainsi que l'automate temporisé de l'activité `<exit>` sont représentés dans la Figure 4.15 ci-dessous.

```
process exit(0) ;
    fpar parentId pid ;
    state inOutState #start ;
        deadline eager ;
        provided true ;
            task stopProcess := true ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

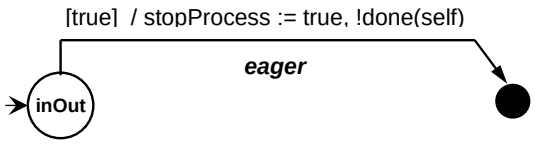

FIGURE 4.15 – Le processus `<exit>`

4.3.6 Activités de communication

Les activités de communication sont utilisées dans les échanges de messages entre un processus BPEL, son client et ses partenaires. Ce sont les activités `<receive>`, `<reply>` et `<invoke>`. Elles sont transformées en un processus IF utilisant des actions de communications (émission et réception de signaux). Dans notre méthode de transformation, toutes les actions de réception sont paresseuses et ne bloquent jamais l'écoulement du temps. Elles sont associées à une échéance `lazy` (cf. § 4.2.2).

Note 4.3. Nous appelons un processus de communication tout processus décrivant une activité de communication ou les éléments `<onMessage>` et `<onEvent>`.

Nous donnons ci-dessous la transformation respective de deux activités de communication : `<receive>` et `<invoke>` synchrone.

Exemple 4.9 (Transformation de <receive>)**.** Le processus IF résultant de la transformation de l'activité `<receive>` suivante est illustré dans la Figure 4.16.

```
<receive partnerLink="pl" portType="pt" operation="op" variable="v" />
<variables>
    <variable name="v" messageType="mess"/>
</variables>
```

t_1 est une transition paresseuse (`lazy`) qui attend la réception d'un signal `mess` et envoie un message de terminaison `done` au processus parent. t_2 est une transition urgente (`eager`) permettant d'interrompre l'exécution du processus `<receive>` en cas de réception d'un signal `terminate`.

Exemple 4.10 (Transformation de <invoke> synchrone)**.** Le processus IF résultant de la transformation de l'activité `<invoke>` synchrone suivante est illustré dans la Figure 4.17 ci-dessous :

```
<invoke partnerLink="pl" portType="pt" operation="op"
                        inputVariable="iv" ouputVariable="ov" />
<variables>
    <variable name="iv" messageType="mess1"/>
    <variable name="ov" messageType="mess2"/>
</variables>
```

t_1 est une transition urgente (`eager`) qui envoie un message $mess_1$ et passe à l'état intermédiaire `inter`. t_2 est une transition urgente (à la différence de toutes les transitions de réception des autres processus décrivant une activité de communication) qui reçoit immédiatement un signal $mess_2$ et envoie un message de terminaison `done` au processus parent. t_2 et t_4 sont des transitions urgentes permettant d'interrompre l'exécution du processus `<invoke>` en cas de réception d'un signal `terminate`.

Note 4.4. Toutes les actions de réception des processus des activités ou éléments de communication (e.g. `<receive>`, `<onMessage>`) sont paresseuses (associées à une urgence `lazy`) à l'exception de l'action de réception d'un processus `<invoke>` synchrone qui est urgente.

```
process receive(0) ;
    fpar parentId pid ;
    var v messType ;
    var pl plType ;
    state inOutState #start ;
        deadline lazy ;
        input mess(pl,v) ;
            output done(self) to parentId ;
                stop ;
        deadline eager ;
        input terminate() ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

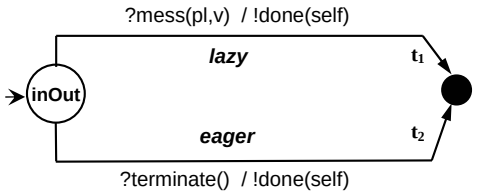

FIGURE 4.16 – Le processus <receive>

4.3.7 Activités structurées

Les activités structurées décrivent l'ordre d'exécution de leurs sous-activités [1]. Les activités <sequence>, <while>, <if> et <repeatUntil> fournissent un contrôle séquentiel. L'activité <flow> décrit une exécution concurrente et permet la synchronisation de ses activités. L'activité <pick> permet un choix contrôlé par l'arrivée d'événements.

Nous présentons ci-dessous la transformation de chacune de ses activités structurées. Dans la suite, nous appelons sous-processus le processus de toute sous activité.

Activité sequence

L'activité <sequence> est transformée en un processus IF (illustré par la Figure 4.18 ci-après) pouvant exécuter séquentiellement les processus des sous-activités de <sequence>. Le processus <sequence> crée, dans l'ordre d'apparition, le processus de chacune de ces sous-activités (par l'action `fork` sp_i dans les transitions t_1 et t_2) et attend à chaque fois sa terminaison (sp_i `?done(self)` de t_2) avant de créer le processus de la sous activité suivante. Il se termine normalement quand son dernier sous-processus termine normalement son exécution (sp_n `?done(self)` de t_3). Le processus <sequence> est interrompu par la réception d'un signal `fault` (dans t_4) ou `terminate` (dans t_5). Dans ce cas, il arrête son exécution et applique une terminaison forcée à son sous-processus actif.

```
process invoke(0) ;
    fpar parentId pid ;
    var iv mess1Type ;
    var ov mess2Type ;
    var pl plType ;
    state inState #start ;
        deadline eager ;
            output mess(pl,iv) via {toLink}0 ;
                next interState ;
        deadline eager ;
        input terminate() ;
                next outState ;
    endstate ;
    state interState ;
        deadline eager ;
        input mess(pl,ov) ;
            next outState ;
        deadline eager ;
        input terminate() ;
                next outState ;
    endstate ;
    state outState ;
        deadline eager ;
            output done(self) to parentId ;
                stop ;
    endstate ;
endprocess ;
```

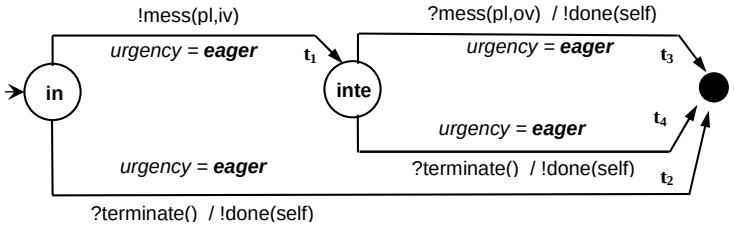

FIGURE 4.17 – Le processus <invoke> synchrone

Activité if

L'activité `<if>` définit une liste ordonnée de choix conditionnels permettant de sélectionner une seule activité à exécuter [1]. Le processus `<if>` (présenté dans la Figure 4.19) sélectionne une de ses branches dans l'ordre de leur apparition si sa condition est satisfaite (comme dans les transitions t_3 et t_4). Ensuite, il crée le processus de la sous activité associée à cette branche et attend sa terminaison (sp_j `?done(self)` de t_5). Le process `<if>` peut être aussi interrompu par la réception d'un signal `fault` (comme dans t_6) ou `terminate` (comme dans t_7)). Dans ce cas, si aucune branche n'a été choisie il termine immédiatement son exécution, sinon la terminaison forcée est appliquée à son sous-processus décrivant l'activité associée à la branche déjà choisie. Notons que les deux transitions t_1 et t_2 permettent de parcourir les conditions associées aux branches de `<if>` jusqu'à la satisfaction de l'une d'elles.

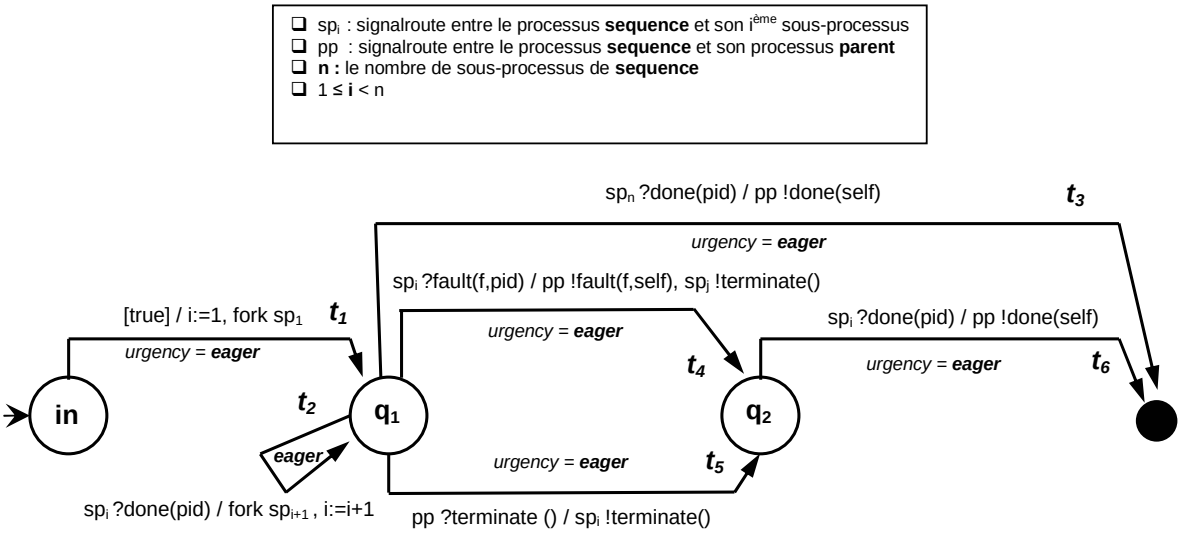

FIGURE 4.18 – Le processus <sequence>

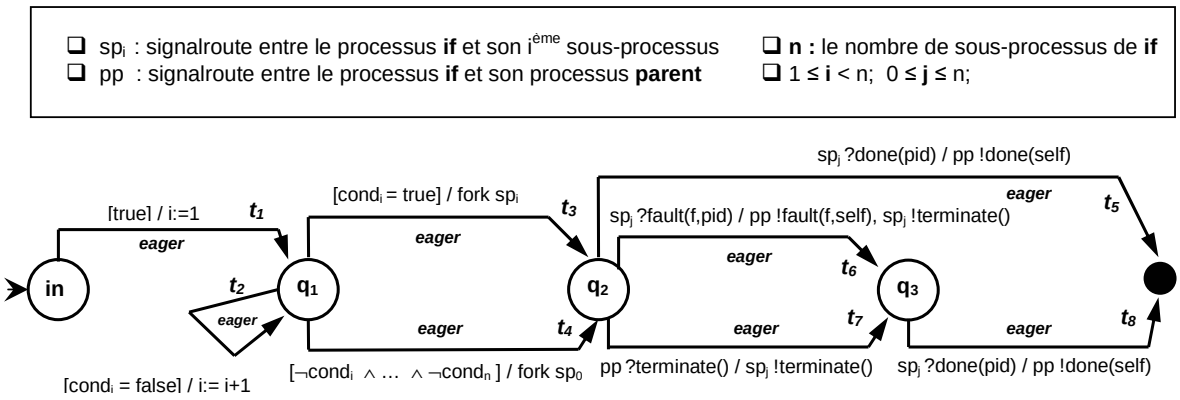

FIGURE 4.19 – Le processus <if>

Activités while et repeat until

Les activités `<while>` et `<repeatUntil>` permettent une exécution répétée de leur sous activité [1]. Elles sont transformées en un processus IF. Le processus `<while>` est illustré par la Figure 4.20 ci-après. A chaque itération, le processus `<while>` crée le processus de sa sous-activité (son sous-processus) tant que sa condition est satisfaite (comme dans les transitions t_1 et t_3). Le processus `<repeatUntil>` crée répétitivement le processus de sa sous-activité jusqu'à la satisfaction de sa condition. Les processus `<while>` et `<repeatUntil>` attendent avant chaque itération la terminaison de leur sous-processus (par l'action `sp ?done(self)` de t_3). Ces deux processus se terminent normalement dès que leur condition est évaluée respectivement à `false` (comme dans t_4) et à `true`. Leur itération peut être interrompue par un signal `fault` (comme dans t_5) ou `terminate` (comme dans t_6), et cela sera suivi par une terminaison forcée de leur sous-processus.

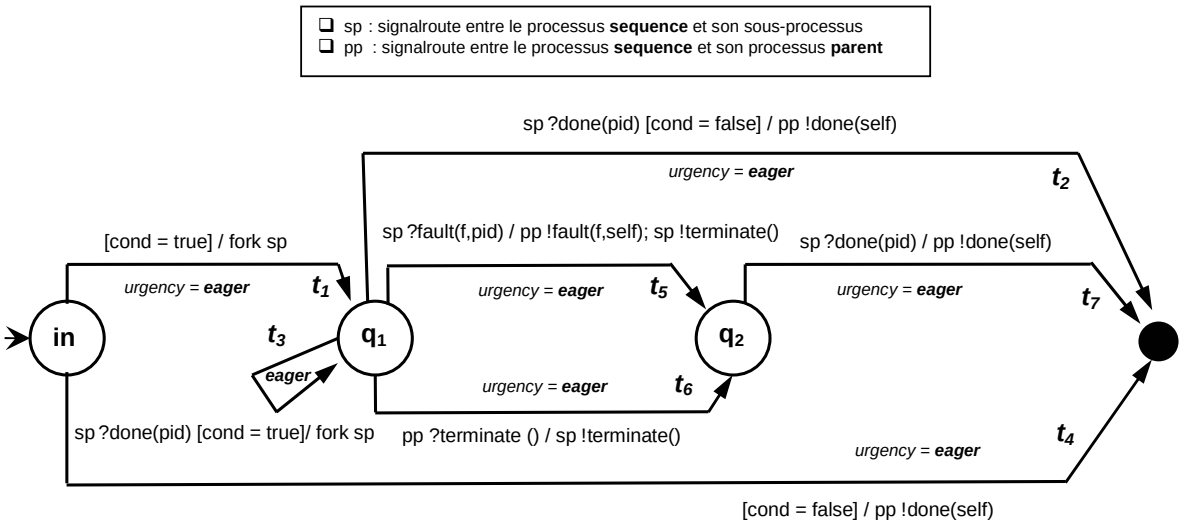

FIGURE 4.20 – Le processus `<while>`

Activité flow

L'activité `<flow>` permet une exécution concurrente d'un ensemble d'activités [1]. Le comportement d'une activité `<flow>` est décrit par un processus IF qui crée simultanément le processus de ses sous-activités. Chaque sous-processus est associé à deux variables booléennes (relatives à leurs états de création et de terminaison) : `start` et `end`. Le processus `<flow>` gère ainsi la liste de ses sous-processus et leurs états. La synchronisation des liens est effectuée par le processus `linksManager` qui est décrit dans la Section 4.3.4. Le processus `<flow>` se termine quand tous sous-processus terminent normalement leur exécution. En recevant un signal `terminate` ou `fault`, le processus `<flow>` arrête son exécution et force la terminaison de tous ses sous-processus actifs.

Activité pick

L'activité <pick> attend l'occurrence d'un événement et exécute par conséquent l'activité qui y est associée [1]. Ce comportement est décrit en IF par un processus (représenté dans la Figure 4.21 ci-dessous) qui attend l'arrivée d'un événement, crée ensuite le processus de la sous-activité associée (comme dans la transition t_3) et attend sa terminaison (comme dans t_5). L'événement <onMessage> est décrit par un process <receive> car il est considéré comme étant une réception de message (InterEnv ?mess(pl,v) dans t_3). L'événement <onAlarm> est considérée comme une attente d'une échéance ou l'écoulement d'une certaine période ; Il est alors décrit par un process <wait> (garde temporelle [c=d] dans t_3).

La terminaison du processus <pick> est similaire à celle de l'activité <if> (cf. § 4.3.7). Notons que les actions de réception d'un signal de communication ne sont jamais urgentes et elles sont associées à l'échéance lazy (cf. § 4.2.2). A la différence, les transitions du processus <onAlarm> sont toutes urgentes et sont associées à l'échéance eager.

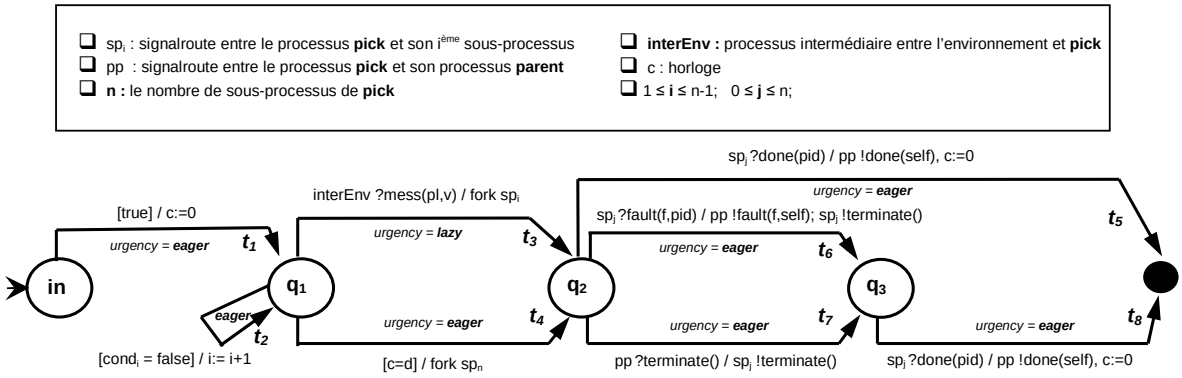

FIGURE 4.21 – Le processus <pick>

4.3.8 Activité scope

L'activité <scope> fournit un contexte influant sur l'exécution de ses activités [1]. Ce contexte inclut des variables, des liens partenaires et des ensembles de corrélation. Il a ses propres gestionnaires de fautes, d'événements, de terminaison et de compensation. Chaque activité <scope> a une activité principale qui décrit son comportement. Elle est décrite par un processus IF qui commence son exécution par créer trois sous-processus associés respectivement à son activité principale (<activity>), à ses gestionnaires d'événements (<eventHandlers>) et à ses gestionnaires de fautes (<faultHandlers>).

Le processus <scope> a ses propres déclarations de variables, de fautes et de corrélations locales. Notons que les éléments <compensationHandlers> ne sont pas pris en considération dans notre travail. Ce processus <scope> se termine normalement si ces deux premiers sous-processus terminent leur exécution sans interruption et sans faute. Quand il est interrompu par la réception d'un signal terminate ou d'un signal fault, il force la terminaison de ses deux premiers sous-processus en propageant un signal terminate et attend leur terminaison. Dans le cas de la réception de fault (interception d'une faute), le processus <scope> propage ce signal fault au processus

de ses gestionnaires de fautes <faultHandlers>. Si ce dernier processus n'arrive pas à traiter la faute interceptée, le processus <scope> propage de nouveau cette faute en envoyant un signal fault à son processus parent englobant (éventuellement un autre processus <scope> ou le processus <process>) et arrête l'exécution de son sous-processus actif (i.e. celui de ses gestionnaires de fautes). La transformation des ensembles de corrélation <correlationSets>, des gestionnaires de fautes et d'événements est décrite dans les sections suivantes.

4.3.9 Corrélation des messages

Les ensembles de corrélation <correlationSets> sont des ensemble de propriétés partagées par tous les messages d'un groupe corrélé [1]. Ils peuvent être utilisés par les activités de communication (<invoke>, <receive>, <reply>) et l'élément <onMessage>. Dans notre transformation de BPEL, la corrélation des messages est gérée de la façon suivante :

- Le processus de l'élément <process> ou d'une activité <scope> est enrichi par une structure de données représentant les ensembles de corrélation et contenant des enregistrements de la forme {name ; status ; properties} où
  - name désigne le nom d'un ensemble de corrélation,
  - status indique le statut actuel de la corrélation (si elle est initialisée ou pas),
  - properties est une table de propriétés contenant le nom et la valeur de chaque propriété de corrélation,
- Chaque processus d'une activité de communication utilisant ces ensembles de corrélation est étendu par deux variables locales :
  - initiate : est une variable indiquant si la corrélation doit être initialisée,
  - cs_name : désigne le nom de l'ensemble de corrélation utilisé,
- Quand la variable initiate est affectée à yes, le processus de communication initie la corrélation cs_name en assignant true à sa variable status associée et définit les propriétés de cette corrélation avec les valeurs des propriétés contenues dans le message échangé ;
- Si l'ensemble de corrélation cs_name a été déjà initialisé (i.e. statut = true) et la variable initiate est affectée à no, le message échangé appartient à la conversation (déterminée par la corrélation) si les valeurs des propriétés de corrélation contenues dans ce message sont identiques à celles de l'ensemble de corrélation définies précédemment ;
- Un message d'erreur de violation de corrélation (correlationViolation) est propagé par le processus de communication dans les trois cas suivants :
  - l'ensemble de corrélation a été déjà initialisé et la variable initiate est affectée à yes,
  - l'ensemble de corrélation n'a pas encore été initialisé et initiate est affectée à no,
  - les valeurs des propriétés de corrélation contenues dans le message échangé sont différentes de celles des ensembles de corrélation.

Exemple 4.11 (Transformation d'une activité de communication avec gestion de la corrélation)**.** L'activité `<receive>` (présentée ci-dessous) utilise un ensemble de corrélation `cs` qu'elle doit initialiser avec les propriétés de corrélation contenues dans son message. Cette initialisation est due à la valeur `yes` de l'attribut `initiate` de `<correlations>`.

```
<receive partnerLink="pl" portType="pt" operation="op" variable="v" >
    <correlations>
        <correlation set="cs" initiate="yes" />
    </correlations>
/receive
```

Le processus IF résultant de la transformation de l'activité `<receive>` est illustré dans la Figure 4.22 ci-dessous. La transition t_1 vérifie en premier si la corrélation n'a pas encore été initialisée (i.e. `status(cs) = false`). Ensuite, elle attend la réception d'un message avant de mettre à jour l'état de cet ensemble de corrélation (par l'action `properties(cs) := properties(mess)` de t_1) et assigne `true` à la variable `status(cs)`. Enfin, elle définit les propriétés de corrélation de `cs` et envoie un signal `done` au processus parent. Si cette corrélation a été déjà initialisée (i.e. `status = true`), la transition t_3 propage un message d'erreur de violation de corrélation en envoyant un signal `fault` au processus parent. La transition t_2 et t_4 arrêtent l'exécution du processus `<receive>` en cas de réception d'un signal `terminate` et envoie un message de terminaison `done` au processus parent.

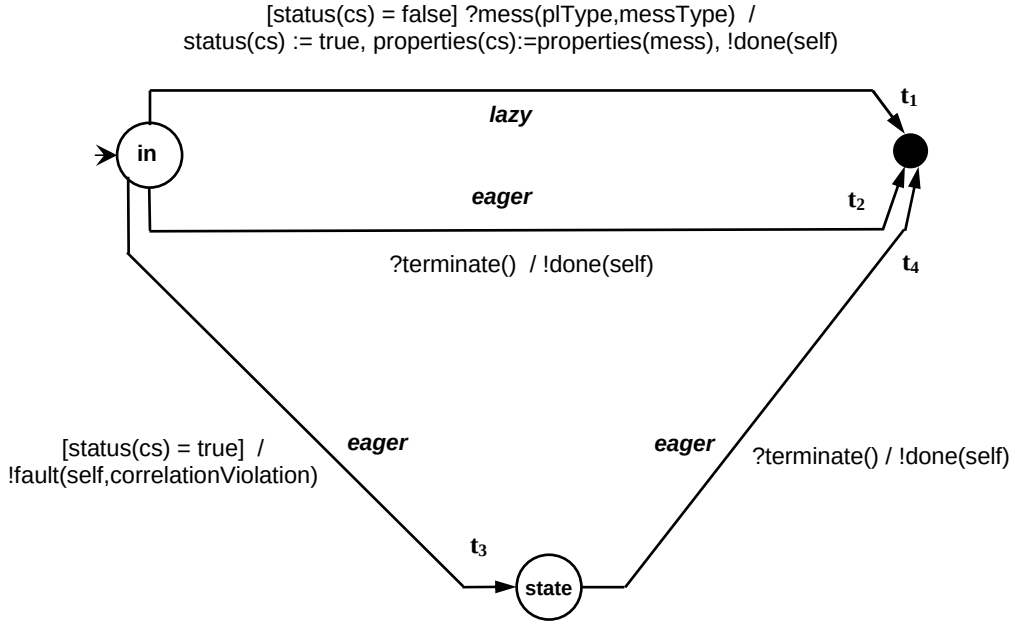

FIGURE 4.22 – Le process `<receive>` avec gestion de la corrélation

4.3.10 Gestionnaires de faute

Les gestionnaires de fautes (<faultHandlers>) d'un processus BPEL ou d'une <scope> est un ensemble de clauses (d'interception de fautes) permettant de définir un comportement en réponse à un type de faute [1].

Ces gestionnaires sont transformés en un processus IF semblable à celui d'une activité <pick> (cf. § 4.3.7) appliquée à une séquence de clauses (<catch> ou <catchAll>) permettant de sélectionner une activité à exécuter en réponse à l'interception d'une faute. Ils sont transformés de la manière suivante :

- chaque clause de la forme <catch faultName="f" ... faultMessageType="fmType"> est transformée en la réception d'un message d'erreur externe défini dans un type de port WSDL [5] ;
- chaque clause de la forme <catch faultName="f" ... > est transformée en la réception d'un message de faute interne : fault(pid,f) ;
- la clause <catchAll> est utilisée pour intercepter toute faute (interne ou externe) non traitée par les clauses <catch>.

Note 4.5. Une activité <invoke> *synchrone* peut être associée à un ensemble de clauses <catch> et <catchAll> afin de traiter les fautes externes envoyées en réponse à une invocation. Cet ensemble de clauses pourra être considéré comme des gestionnaires de fautes externes. Dans le cas de la réception d'un message de faute externe, le processus <invoke> pourra ainsi créer un sous-processus (de gestion de faute) en réponse à ce message.

4.3.11 Gestionnaires d'événement

Un processus BPEL ou une <scope> peut être associé à des gestionnaires d'événements <eventHandlers> qui sont invoqués parallèlement à leur activité principale (<activity>) si un événement se produit [1]. Ces gestionnaires sont transformés de la même manière qu'une activité <pick> (cf. § 4.3.7) où l'événement <onEvent> est décrit par un processus <receive>. La terminaison du process <eventHandlers> est similaire à celle du processus <if> (cf. § 4.3.7).

4.3.12 L'élément process

La description d'un processus exécutable BPEL commence toujours par l'élément racine <process> qui décrit le comportement de ce processus de façon opérationnelle. Il est composé des éléments optionnels suivants : liens partenaires (<partnerLinks>), variables (<variables>), ensemble de corrélation (<correlationSets>), gestionnaires de fautes (<faultHandlers>), d'événements (<eventHandlers>) et de compensation (<compensationHndlers>). Les éléments <compensationHandlers> ne sont pas pris en considération dans notre travail.

Chaque élément <process> a une activité principale <activity> qui décrit son comportement effectif. Il est décrit par un processus IF qui commence son exécution par créer trois sous-processus décrivant le contrôle de flot du processus BPEL — plus précisément son activité principale

(<activity>) et ses gestionnaires d'événements (<eventHandlers>) — et ses gestionnaires de fautes (<faultHandlers>). Le processus <process> contient l'ensemble des variables et des corrélations globales. La transformation de ces éléments optionnels a été détaillée dans les sections précédentes. Le processus <process> se termine normalement si ses deux premiers sous-processus (<activity> et <eventHandlers>) terminent leur exécution sans interruption et sans faute. Il peut être interrompu par la réception d'un signal fault ou par l'activité <exit> qui assigne true à sa variable stopProcess. Dans ces deux cas, il applique la terminaison forcée à ses deux premiers sous-processus en propageant un signal terminate et attend leur terminaison. Dans le cas de l'interception d'une faute, le processus <process> crée le processus de ses gestionnaires de fautes (<faultHandlers>) et attend sa terminaison.

4.3.13 Client et partenaires

Dans notre transformation de BPEL, le client et les partenaires d'un processus sont considérés comme une partie de l'environnement du système. Dans la spécification du langage IF décrite dans [158], la communication entre deux processus IF pourrait être synchrone par rendez-vous (i.e. utilisation des portes de synchronisation comme dans le langage LOTOS (réf lotos)). Dans la version actuelle du simulateur de IF [171, 159, 165], ce type de communication n'a pas encore été implanté Nous ne contentons alors des communications asynchrones entre processus à travers des files d'attentes. Chaque processus IF a sa propre file d'attente associée en entrée pouvant sauvegarder tous les messages provenant des autres processus. La consommation de ces messages prendra un certain temps. Par contre, les messages envoyés par l'environnement aux processus sont traités de manière différente. Ils ne sont pas sauvegardés dans les files d'attentes des processus mais ils sont consommés immédiatement au fur et à mesure de leur réception.

Dans notre transformation, Chaque processus de communication, en particulier celui décrivant les activités <receive> et <invoke> *synchrone*, ou les éléments <onMessage> et <onEvent>, peut recevoir un signal interne (e.g. done, terminate, fault) envoyé par un autre processus ou un message de communication externe provenant de l'environnement. Malheureusement, l'ordre de consommation des messages reçus ne peut pas être garanti ; les signaux contenus dans la file d'attente d'un processus (qui sont envoyés par les autres processus) doivent attendre le traitement des nouveaux messages externes entrants qui sont envoyés par l'environnement. La Figure 4.23 montre le traitement des messages entrants d'un processus IF.

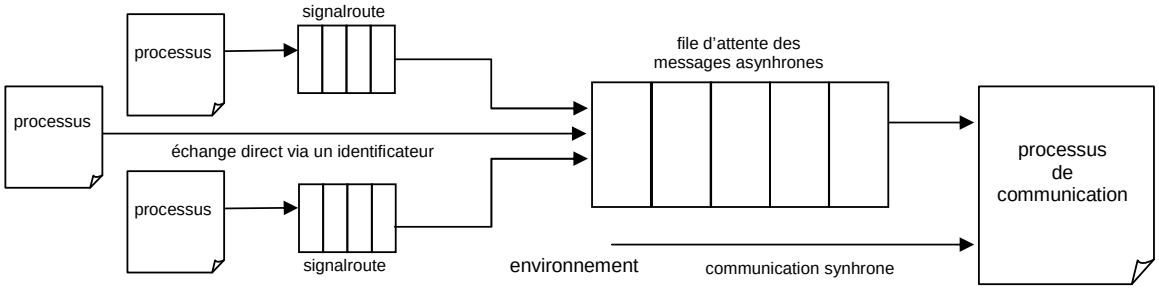

FIGURE 4.23 – Traitement des messages entrants par un processus IF

Tous les messages envoyés à un processus via des routes de communications (`signalroutes`) ou par adressage direct (en utilisant l'identificateur de l'instance active de ce processus) sont tous sauvegardés dans la file d'attente FIFO du processus. La communication entre l'environnement et les processus est synchrone et les messages ne sont pas consommés selon leur ordre d'arrivée.

Pour résoudre ce problème et garantir l'ordre de consommation des messages, nous introduisons un nouveau processus IF intermédiaire appelé `InterEnv`. Chaque message externe est envoyé par l'environnement vers `InterEnv` qui doit le retransmettre au processus destinataire approprié (processus d'une activité de communication). Dans notre transformation, chaque message externe est destiné à un seul processus de communication. Cela permettra à `InterEnv` de bien distribuer tous les messages provenant de l'environnement puisqu'ils ont une seule destination. Les messages retransmis par `InterEnv` seront sauvegardés dans la même file d'attente que les signaux internes envoyés par les autres processus (comme le montre la Figure 4.24) ; Cela permettra de garantir l'ordre de consommation des messages. Notons que ce processus intermédiaire communique avec chaque processus de communication à travers un `signalroute`. Il communique aussi avec l'environnement à travers un seul `signalroute` appelé `fromEnv` permettant de transporter tous les messages externes envoyés par cet environnement.

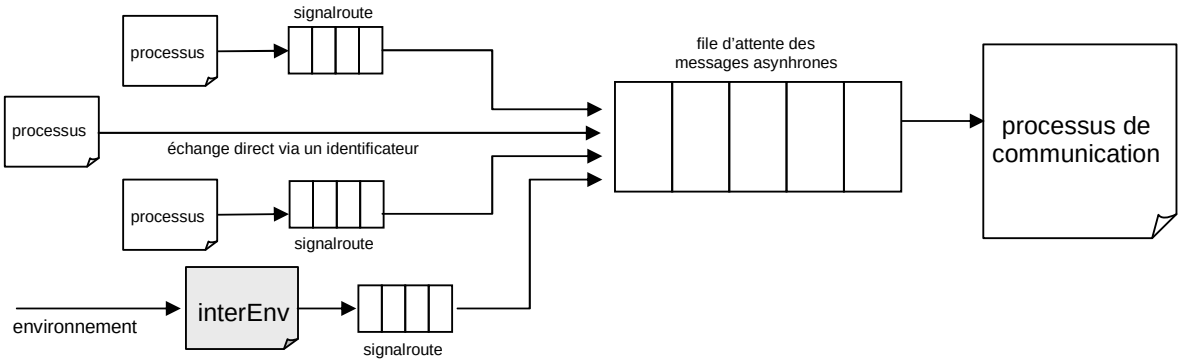

FIGURE 4.24 – Utilisation d'un processus intermédiaire — `InterEnv`

La transformation de BPEL en IF est résumée dans la Table 4.1 ci-dessous.

4.3.14 Synthèse

Dans ce chapitre nous avons présenté le langage IF et plus précisément sa syntaxe et son modèle formel : automate temporisé à échéances. Nous avons détaillé notre méthode de transformation de la description d'un processus BPEL en une spécification IF. Nous avons abordé la transformation de la majorité des concepts de BPEL, à savoir les données, les messages, les liens partenaires, les activités basiques et structurées, la propagation et la gestion des fautes, la gestion des événements, la terminaison et la synchronisation des activités, les activités `<scope>`, la corrélation des messages et la description WSDL du client et des partenaires d'un processus BPEL.

Dans le chapitre suivant, nous allons aborder notre approche de test de l'orchestration de services Web. Nous expliciterons la relation de conformité utilisée pour le test de conformité, notre méthode de génération de tests temporisés abstraits ainsi que la concrétisation de ces tests. Nous décrirons aussi les architectures de tests qui sont bien adaptées à notre approche de test.

Concept BPEL / WSDL	Language IF
types de messages	types de données complexes (`record`, `array`, `enumeration`, `range`)
expressions booléennes	variables booléennes et contraintes logiques
type des liens partenaires : `<partnerLinkType>`	type complexe : `enumeration` = {`partnerLink name,portType,operation`}
message	`signal`(type de partnerLink, type de message)
ensemble de corrélation : `<correlationSet>`	type complexe : `record = {name,status,propertie}`
lien partenaire : `<partnerLink>`	`signalroute`
Gestion de fautes	propagation d'un signal `fault` et création d'un processus `faultHandlers`
gestion de la terminaison	propagation d'un signal `terminate`, utilisation de la variable `stopProcess` et création d'un processus `terminationHandler`
Gestion de la corrélation des messages	Initialisation des variables de corrélation et ajout de gardes aux transitions des processus des activités de communications
activité basique	processus IF simple n'ayant aucun sous-processus
activité structurée	processus IF complexe créant dynamiquement ses sous-processus
synchronisation des activités	création d'un processus de gestion de liens `linksManager` et échange des messages de synchronisation avec les activités source et cible d'un lien
gestionnaires de fautes : `<faultHandlers>`	processus IF associé à un ensemble de clauses (`<catch>` et `<catchAll>`) et de sous-processus
gestionnaires d'événements : `<eventHandlers>`	processus IF associé à un ensemble d'événements (`<onMessage>` et `<onAlarm>`) et de sous-processus
activité `<scope>`	processus IF complexe ayant quatre sous-processus de flot de contrôle (`<eventHandlers>` et `<faultHandlers>`) et de terminaison (`<terminationHandler>`) et de gestion de fautes (`<faultHandlers>`)
client et partenaires	environnement IF
élément `<process>`	processus IF complexe ayant trois sous-processus de flot de contrôle (`<eventHandlers>` et `<faultHandlers>`) et de gestion de fautes (`<faultHandlers>`)

TABLE 4.1 – Transformation de BPEL en IF

Chapitre 5

Méthode de test de la composition de services Web

.

Sommaire

5.1 Introduction

Les services Web sont des applications auto descriptives, modulaires et faiblement couplées fournissant un modèle simple de programmation et de déploiement d'applications. La composition sert à combiner des services Web (appelés partenaires) pour former de nouveaux services dits composés dont l'exécution implique des interactions avec ces services partenaires en faisant appel à leurs fonctionnalités.

Le test de services Web, en particulier de services composés, pose de nouveaux défis étant donné les caractéristiques spécifiques de la composition de services décrite en BPEL telles que la gestion des transactions de longue durée, la complexité des données et de leur transformation, la gestion des exceptions et la corrélation des messages.

Dans notre approche de test, la description BPEL d'un service composé est considérée comme étant une spécification de référence de la composition. Cette approche s'intéresse au test de conformité des services composés et l'automatisation de la génération de tests. Ce test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. Il est considéré dans notre approche comme étant un test fonctionnel de type boîte grise où les interactions entre le service composé et ses différents partenaires sont connues.

Notre approche est constituée de quatre phases : (i) modélisation de la composition de services, (ii) génération automatique des tests temporisés abstraits, (iii) concrétisation des tests abstraits par rapport à une architecture de test, (iv) exécution des tests et émission de verdicts. La modélisation temporelle de la composition de services — à des fins de génération de cas de test — a été largement détaillée dans le Chapitre 3. Elle a servi dans la transformation de description d'un processus BPEL en une spécification formelle en langage IF (i.e. système d'automates temporisés) qui a fait l'objet du Chapitre 4. La génération automatique de tests est guidée par un ensemble d'objectifs de test décrivant certaines exigences de conformité (propriétés fonctionnelles et/ou temporelles). Elle se base sur un algorithme de génération qui adapte la stratégie Hit-Or-Jump [6] aux automates temporisés de IF, et construit à la volée un cas de test en effectuant une exploration partielle de l'espace d'états.

Dans le contexte du test des services composés, une architecture de test représente le service composé sous test et son environnement constitué de son client et de ses partenaires (simples ou composés). Le client utilise le service offert par ce service composé via une interface WSDL. Pour la concrétisation et l'exécution de tests, nous considérons deux types d'architecture de tests : architecture centralisée vs. distribuée. La concrétisation des tests consiste à dériver des tests temporisés concrets à partir des tests abstraits générés automatiquement à partir d'une spécification formelle de la composition de services. Ces tests concrets définissent les comportements des testeurs et sont décrits en termes d'interactions avec le service sous test, des délais d'attente et des messages SOAP échangés.

Dans ce chapitre, nous allons présenter notre méthode de test de la composition de services Web et ses différentes phases ; Nous détaillerons, en particulier, les deux phases de génération et de concrétisation des tests en prenant en considération une architecture de test. Pour les besoins de ce test de conformité, nous proposerons deux architectures de test pour le test des services composés (architecture centralisée vs. distribuée) que nous adapterons pour le test de plusieurs instances d'un service composé en vue de tester la corrélation des messages. Ensuite, nous définirons la relation inclusion des traces temporisées qui sera utilisée comme relation de conformité.

Notre approche de test a été publiée dans la conférence ECOWS 2008 [162] et a fait l'objet, en partie, d'un livrable du projet WebMov [1] [172].

5.2 Les besoins de test de la composition de services Web

Un service Web peut être lui même composé d'autres services Web ce qui entraîne plus de difficulté au niveau du test étant donné les caractéristiques spécifiques des services composés. Dans cette section, nous allons résumer les besoins de test d'une composition de services Web.

Automatisation du test Un test manuel peut s'avérer non complet et parfois non efficace. Il peut même être erroné dans le cas de systèmes de plus en plus complexes. Dans le cas du test d'un service composé isolé en simulant ses services partenaires, ces derniers risquent d'être décrits de façon incomplète surtout si l'approche de test n'est pas systématique ou n'a pas été automatisée. Notre méthode de test peut être alors utilisée pour aider et guider les différentes étapes de test effectuées manuellement (e.g. génération de cas test, exécution de tests). L'automatisation permet d'améliorer l'efficacité, la précision et la reproductibilité de tests.

Test boîte grise Le test d'un service Web peut être vu comme un test boîte noire de son implantation. Les cas de test sont générés à partir de son interface WSDL [5] considérée comme étant une spécification de ce service. Nous pouvons aussi appliquer cette approche de test à la composition de services Web en ne considérant que l'interface WSDL du service composé. Mais, nous n'allons pas nous contenter de simples échanges — entre le service composé et son client — pour tester l'implantation d'un service composé. Un cas de test pour notre approche de test boîte grise est plus complexe et devrait contenir aussi les interactions du service composé avec ses partenaires qui sont généralement simulés.

Caractéristiques de la composition de services Web Une composition de services Web, et en particulier une description BPEL, se distingue des autres applications ou systèmes par les caractéristiques suivantes : (i) des interactions synchrones et asynchrones, (ii) complexité des données et de leur transformation (iii) gestion des fautes et des événements, (iv) gestion des transactions, (v) corrélation des messages, etc. Il est important de prendre en compte toutes ces caractéristiques lors du test de cette composition.

Dans la section suivante, nous allons décrire les différentes étapes de notre méthode de test de la composition de services Web.

5.3 Méthodologie de test d'un service composé

L'IEEE [173] définit le test comme : « le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus. ». L'étape de test est très importante dans le cycle de développement des systèmes et des applications. En plus, l'utilisation de services Web pour les applications commerciales, distribuées ou critiques nécessite un besoin croissant d'approches de test efficaces afin de détecter les défauts et d'évaluer la qualité de ces services.

1. `http://webmov.lri.fr/`

Les auteurs de [82] ont proposé deux approches de test des services Web composés :

Approche boîte noire Dans cette approche, l'implantation d'un service composé est testée sans aucune connaissance de sa structure interne, y compris ses interactions avec ses partenaires. Pour cela, une spécification formelle ou informelle (décrivant la composition de services) est utilisée pour la génération de tests. Cette approche est plus adaptée pour le test des scénarios ou pour le prototypage rapide des services Web [174].

Approche boîte blanche Dans cette approche, le testeur connaît la structure interne du service Web sous test. BPEL étant est un langage exécutable, une description BPEL peut être considérée comme le code source d'une composition de services Web. Dans ce cas, pour la génération des suites de test, plusieurs critères de couverture structurelle basée sur le code peuvent être appliqués (couverture des activités, des chemins, des conditions, etc.).

Nous ajoutons à cela notre approche de test **boîte grise** où les interactions entre le service composé et ses différents partenaires sont connues à la différence du test boîte noire où elles sont inconnues puisque elles sont internes à cette boîte noire (service sous test).

Le présent travail s'intéresse au test de conformité de la composition de services Web et à l'automatisation du test. Ce test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. Nous considérons ce test de conformité comme étant un test fonctionnel de type boîte grise.

En général, la description d'un processus BPEL peut être considérée soit comme une spécification de référence d'une composition de services Web, soit comme une spécification exécutable de cette composition (i.e. code source). Dans notre travail, nous la considérons comme une spécification de référence d'un service composé. Cette description BPEL peut être générée à partir : (i) d'une description formelle/informelle de la composition en termes de flot de données et de contrôle (e.g. BPMN [111], UML [80]), (ii) et de la description des interfaces WSDL du service composé et de ses partenaires [5].

Notre méthode de test est illustrée par la Figure 5.1. Elle est constituée de quatre phases décrites ci-dessous : phase de modélisation temporelle de la composition de services Web, phase de génération de tests abstraits, phase de concrétisation des tests et phase d'exécution des tests.

Phase de modélisation. Afin de dériver des tests à partir d'une description BPEL, nous avons besoin d'un modèle formel décrivant le comportement du processus BPEL (i.e. composition de services Web). Pour cela, nous proposons d'utiliser le langage intermédiaire IF [165] et plus précisément son modèle de systèmes d'automates temporisés communicants comme modèle formel [160, 161]. Pour cela, nous utiliseront la méthode de transformation de BPEL en IF qui a été décrite dans le Chapitre 4. Cette méthode de transformation se base principalement sur notre modélisation temporelle de la composition de services Web (par des machines WS-TEFSM) qui a été largement détaillée dans le Chapitre 3.

Phase de génération des tests Abstraits. Dans cette phase, nous générons automatiquement des tests temporisés à partir : (i) de la spécification formelle en IF de la composition de services, (ii) et d'un ensemble d'objectifs de test décrivant des propriétés à vérifier sur le SST. Notre algorithme de génération de test utilise une stratégie d'exploration partielle de l'espace d'états qui est guidée par ces objectifs de test. Cet algorithme ainsi que notre relation de conformité seront explicités ci-dessous dans la Section 5.5.

Phase de concrétisation de tests. Cette phase permet de dériver des tests temporisés concrets à partir : (i) des tests abstraits générés lors de la seconde phase, (ii) et des interfaces WSDL du service composé et de ses partenaires. Ces tests concrets doivent être adaptés à l'une de nos deux architectures de test que nous décrirons dans la Section 5.4. Ils définissent les comportements des testeurs et sont décrits en termes d'interactions avec le service sous test, des délais d'attente et des messages (SOAP) échangés. .Ils seront utilisés dans la phase d'exécution pour tester si le SST est conforme à la spécification de référence. Cette concrétisation fera l'objet de la Section 5.6.

Phase d'exécution de tests. Cette phase correspond à la mise en œuvre des tests et de leur exécution. Elle consiste à vérifier la conformité d'une implantation d'un service Web composé par rapport à sa spécification de référence. Elle se termine par l'obtention d'un verdict.

Dans la section suivante, nous allons présenter deux architectures afin de générer les cas de test de conformité des services Web composés.

5.4 Architecture de test

Dans le contexte du test de conformité des services Web composés en approche boîte grise, l'architecture de test représente le service sous test (SST) et son environnement qui est constitué du client du SST et de ses partenaires (qui peuvent être simples ou composés). Ce client peut être une application ou une implantation qui utilise le service offert par ce SST via une interface WSDL.

Dans ce contexte du test des service composés, il est nécessaire selon les auteurs de [105, 175] de simuler ces partenaires pour les raisons suivantes :

- pendant la phase du test, certains services partenaires peuvent être en cours de développement ;
- certains services partenaires peuvent être développés et gérés par d'autres équipes ou entreprises. Dans ce cas, il est parfois impossible d'obtenir leurs implantations et par conséquent de mettre en œuvre l'environnement du test ;
- même si nous avons l'implantation des partenaires, il est parfois plus utile d'utiliser des simulations de ces partenaires pour dériver avec moins d'effort plus de scenarios de test ;
- certains tests nécessitent parfois des conditions de test qui sont très difficiles à reproduire ;
- si le test dépend d'un état du système plus complexe (e.g. l'état des partenaires), il peut être difficile de le mettre en œuvre d'autant plus si le reste du système est en cours de développement.

Notons que pour simuler le client d'un SST, nous utiliserons l'interface WSDL de ce SST. Pour simuler un des partenaires du SST, nous utiliserons l'interface WSDL de ce partenaire.

Les auteurs de [105] ont proposé deux architectures pour le test d'un processus BPEL (approche boîte blanche) : (i) architecture de test centralisée (notée par ATC), (ii) architecture de test distribuée (notée par ATD). Dans ce travail, nous proposons d'adapter ces deux architectures à notre méthode de test (i.e. test de conformité d'un service composé en approche boîte grise) en prenant en compte l'aspect temporel.

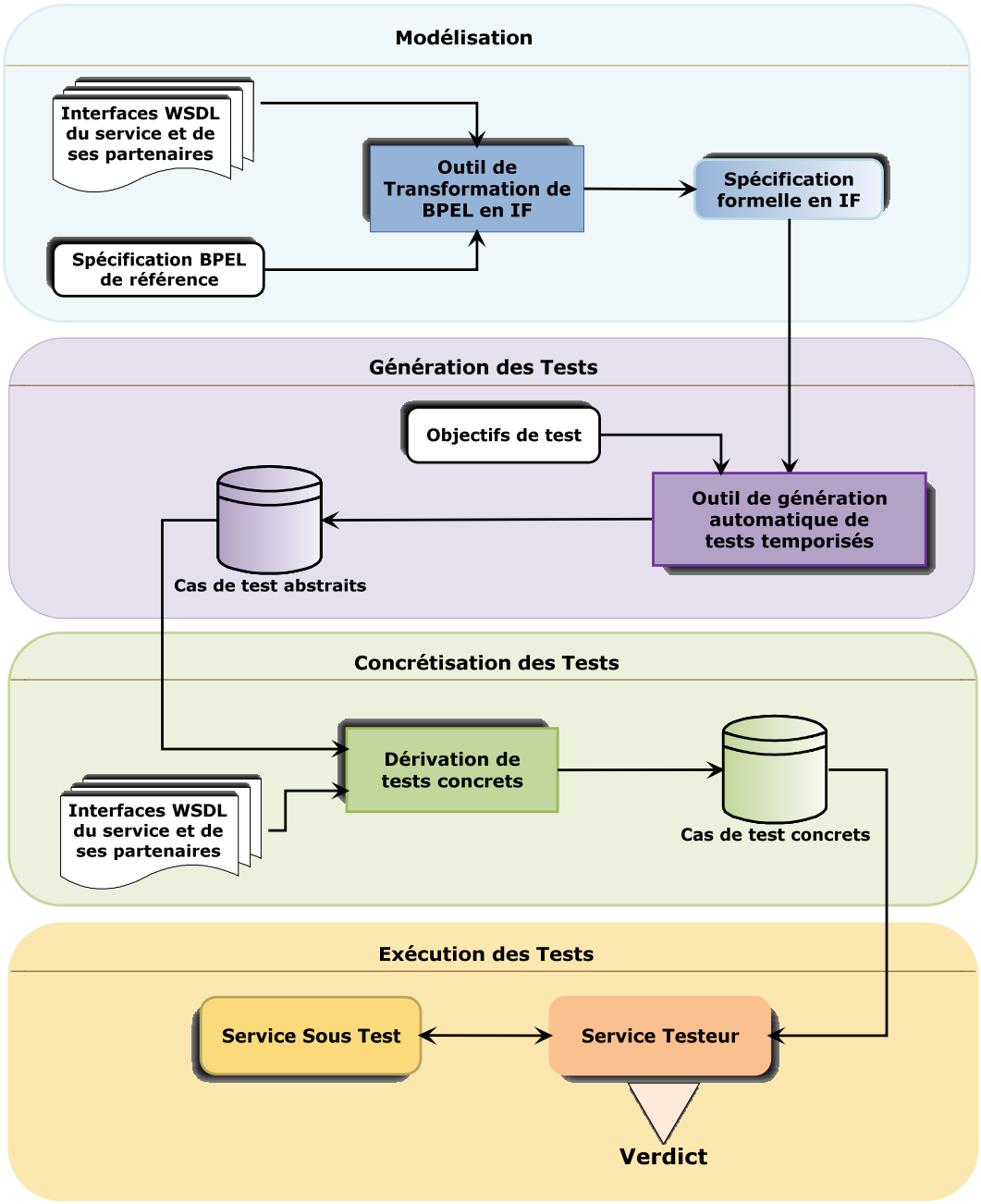

FIGURE 5.1 – Méthode de test de la composition de services Web

Dans ce chapitre, nous considérons un service sous test SST ayant deux services partenaires notés respectivement par SP1 et SP2. Le modèle abstrait de ce SST est illustré dans la Figure 5.2.

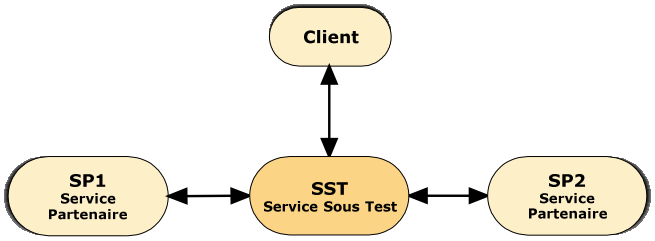

FIGURE 5.2 – Modèle abstrait du service sous test SST

Dans la suite, nous allons présenter ces deux architectures.

5.4.1 Architecture de test centralisée

Dans cette architecture, le client et les partenaires du SST sont simulés par un seul service testeur (composé) noté par ST qui interagit avec le SST par envoi et réception de messages SOAP. Ce ST est contrôlé par un autre service appelé service contrôleur qui est noté par SC permettant de gérer l'exécution du test et d'émettre un verdict. En particulier, il contrôle le début et l'arrêt de l'exécution du ST. L'architecture de test centralisée associée au SST (décrit dans la Figure 5.2) est représentée dans la Figure 5.3 où ST est un testeur composé qui simule à la fois le client et les deux partenaires SP1 et SP2. Le comportement du ST est considéré comme une composition (séquentielle ou parallèle) des comportements du client et des partenaires du SST qui sont décrits par le cas de test. En ignorant les relations temporelles et de causalité entre ces partenaires, nous risquons de ne pas détecter certains types d'erreurs dans le SST. Ces dépendances implicites doivent être conservées dans le ST.

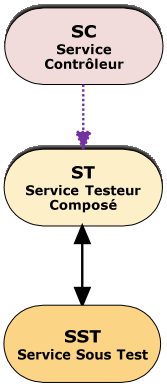

FIGURE 5.3 – Architecture de test centralisée pour SST

Nous décrirons ci-dessous l'architecture de test distribuée qui est plus adaptée dans le cas où la composition des comportements est plus complexe et par conséquent le SC est difficile à maintenir.

5.4.2 Architecture de test distribuée

Dans cette architecture, certains services partenaires peuvent être simulés individuellement par un simple service testeur ST quand d'autres partenaires peuvent être regroupés dans d'autres STs composés (et donc plus complexes). Comme pour l'architecture de test centralisée, ces STs sont contrôlés par un même service contrôleur SC. L'architecture de test distribuée associée au SST (décrit dans la Figure 5.2) est représentée dans la Figure 5.4 où ST0 simule le client, et ST1 et ST2 simulent respectivement les deux partenaires SP1 et SP2.

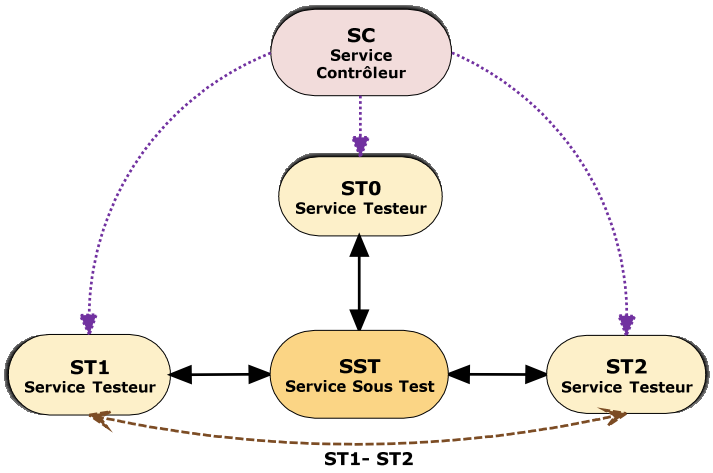

FIGURE 5.4 – Architecture de test distribuée pour SST

Ces STs doivent également considérés les dépendances implicites entre les partenaires du SST. [105] propose trois méthodes pour maintenir ces dépendances dans les STs : (i) l'introduction de liens de coordination entre les STs interdépendants, (ii) l'introduction d'un composant de coordination centralisée, (iii) les deux (i) et (ii). Cette coordination est vraiment nécessaire pour le test d'un SST. Dans la Figure 5.4, ST1−ST2 dénote un lien de dépendance entre les deux testeurs ST1 et ST2.

Dans la section suivante, nous avons étendu ces deux architectures pour le test de la corrélation de messages où nous avons besoin de tester au même moment au moins deux instances du SST.

5.4.3 Test de plusieurs instances d'un service sous test

BPEL fournit un mécanisme de gestion de corrélation de messages permettant de lier les messages échangés aux instances de processus pour lesquelles ils sont destinés. Rappelons qu'un processus BPEL a ses propres ensembles de corrélation qui sont constitués de l'ensemble des propriétés partagées par tous les messages échangés dans un groupe corrélé [1]. Ces messages sont liés à une instance de processus BPEL grâce à leurs valeurs de corrélation. Afin de tester la corrélation des messages et leur utilisation, il est nécessaire de tester plusieurs instances d'un même SST. Pour cela, le client simulé doit invoquer et déclencher l'exécution de plus qu'une seule instance de ce SST. Deux instances sont largement suffisantes pour notre test de la corrélation des messages.

Dans le cadre du test de la corrélation des messages, nous présentons ci-dessous deux exemples d'architecture de test du SST décrit dans la Figure 5.2 : architecture centralisée vs. architecture distribuée. Ces deux architectures sont illustrées respectivement dans la Figure 5.5 et la Figure 5.6.

Dans la Figure 5.5, un seul testeur composé ST est utilisé. Il simule le client et les partenaires du service SST. Pour tester la corrélation des messages, ST initie et interagit avec les deux instances SST_A et SST_B.

Dans la Figure 5.6, le client du SST est simulé par un testeur ST0 et les deux partenaires sont simulés respectivement par deux testeurs ST1 et ST2. Le testeur ST0 initie à la fois les deux instances SST_A et SST_B. De leur côté, les deux testeurs ST1 et ST2 peuvent simuler une ou plusieurs instances d'un partenaire du SST selon que les deux instances SST_A et SST_B invoquent ou pas la même instance de chaque partenaire.

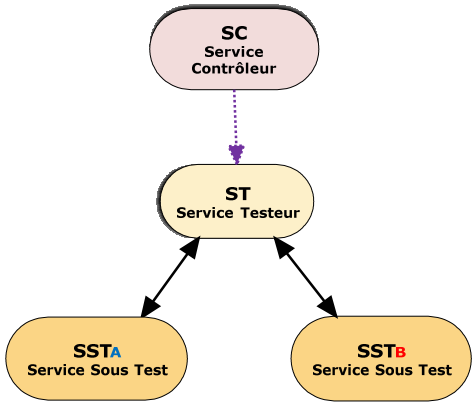

FIGURE 5.5 – ATC pour le test de corrélation de SST

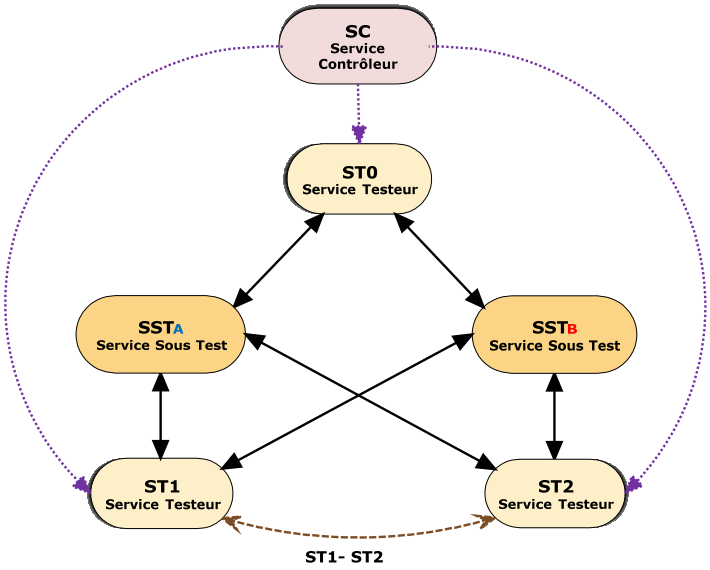

FIGURE 5.6 – ATD pour le test de corrélation de SST

Remarque 5.1. Un service testeur ST peut simuler un ou plusieurs services partenaires (SPs) d'un service sous test (SST) mais il diffère de ces services partenaires [105] ; D'une part, il peut contenir la structure de contrôle du test et/ou le comportement de plusieurs SPs. D'autre part, il est moins complexe qu'un SP étant donné qu'il ne contient qu'une partie de ses interactions qui sont décrites dans le cas de test.

Dans la section suivante, nous allons définir les différents types d'interaction entre un service sous test (SST), son client et ses partenaires. Ces types seront utilisés dans la phase de concrétisation des tests abstraits.

5.4.4 Types d'interactions entre un service sous test et son environnement

En se mettant du coté d'un client ou d'un partenaire d'un service sous test (SST), nous pouvons catégoriser les interactions selon trois modes d'interaction [176] :

1. **interaction simple** : réception ou envoi d'un message ;
2. **interaction synchrone bilatérale** :
   a. envoi/réception synchrone : un client envoie une requête au SST et reçoit une réponse immédiate ;
   b. réception/envoi synchrone : un partenaire est invoqué par le SST et envoie immédiatement sa réponse ;
3. **interaction asynchrone bilatérale** :
   a. envoi/réception asynchrone : un client envoie une requête au SST et attend sa réponse. Cette interaction est équivalente à une suite de deux interactions simples : envoi et réception d'un message ;
   b. réception/envoi asynchrone : un partenaire est invoqué par le SST et envoie sa réponse. Cette interaction est équivalente à une suite de deux interactions simples : réception et envoi d'un message.

En se mettant du coté du SST, l'interaction asynchrone peut être associée à un timeout. Dans ce cas, le SST envoie une requête à un partenaire et attend jusqu'à ce qu'il reçoive une réponse, ou jusqu'à ce qu'un certain délai (timeout) soit atteint.

Note 5.1. Les messages échangés dans une interaction synchrone doivent correspondent à la même opération WSDL (et donc au même type de port et type de lien partenaire) [5].

Après avoir introduit les types d'interactions, nous présentons dans la section suivante les types d'erreurs qui peuvent être détectées par un testeur dans le cadre du test de conformité d'un service composé (approche boîte grise).

5.4.5 Types d'erreurs

Du point de vue d'un service testeur, le test du SST est réalisé en lui fournissant une série de messages SOAP en entrée (probablement après un intervalle de temps), et en observant ses réponses (i.e. ses messages SOAP en sortie). Le verdict de test se fait à partir de ces réponses ou de leur absence durant une certaine période (intervalle de temps). Dans ce cadre, les erreurs dans le SST peuvent être classées en trois types [105, 106] :

1. Contenu incorrect d'un message de sortie ;
2. Absence de message de sortie : un message de sortie attendu n'a pas été émis par le SST ;
3. surplus de message de sortie : un message de sortie non attendu est émis par le SST.

Pour illustrer le troisième type d'erreurs, nous pouvons donner le cas d'un SST qui devrait attendre un message pendant une certaine durée et répondre en conséquence sinon déclencher une alarme. En envoyant un message au SST dans des délais non permis, le testeur s'attend à ne rien recevoir. Si le SST prend en compte le message reçu hors délai et répond en envoyant un message non attendu, alors nous sommes dans la situation d'un surplus de message de sortie.

5.5 Génération automatique de tests temporisés

Dans le cadre du test de la composition de services Web, nous nous intéressons au test de conformité et à l'automatisation de la génération de tests temporisés guidée par un ensemble d'objectifs de test. Nous introduisons dans cette section la terminologie et les concepts nécessaires pour aborder le test de conformité et la génération automatique de tests. Nous présentons ensuite notre relation de conformité et notre algorithme de génération des cas de test. Cet algorithme est fondé sur un système de processus communicants (i.e. composition d'automates temporisés de IF) et est guidé par un ensemble d'objectifs de test décrivant les exigences de conformité (exigences fonctionnelles et/ou temporelles). Pour plus de détails concernant le système et l'automate temporisé de IF, vous pouvez vous référer au Chapitre 4 et en particulier à la Section 4.2.2.

5.5.1 Test de conformité

Dans notre travail, le test de conformité consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence. C'est un test fonctionnel de type boîte grise où le comportement interne de l'implantation sous test (SST) n'est pas à priori connu à l'avance. Seul, le comportement externe de cette implantation est testé par interaction avec son environnement (client et partenaires de SST). Nous construisons les tests à la volée à partir de la spécification et de l'ensemble des objectifs de test. Nous avons défini une relation de conformité entre systèmes IF de processus communicants nommée *inclusion de traces temporisées* que nous présenterons ci-après afin de l'utiliser dans notre test de conformité des services Web composés.

Rappelons qu'un système de IF est une composition d'automates temporisés. Sa sémantique (ou son modèle opérationnel) a été décrite par un système de transitions étiquetées noté par $[SYS] = (S, s_0, \Gamma, \Rightarrow)$. Chaque automate est un tuple $TA = (Q, Act, X, T, q_0)$ où Q est un ensemble fini d'états, Act est un ensemble fini d'actions, X est un ensemble de variables incluant les variables discrètes et les horloges, T est un ensemble de transitions et q_0 est l'état initial. Chaque transition est étiquetée par un ensemble de gardes g, un ensemble d'actions $a \in Act_\tau$ (où Act_τ étend l'ensemble Act par des actions internes) et une échéance $u \in U$. L'échéance d'une transition est de type *eager*, *lazy* ou *delayable*(cf. § 4.2.2). Elle est utilisée pour contrôler l'écoulement du temps.

Dans la suite, le système de IF décrivant la spécification et l'implantation d'un service composé est noté par SYS, plus précisément par respectivement SYS_S et SYS_I. Soient $s, s' \in S$ deux états sémantiques de SYS. $s \stackrel{l}{\Rightarrow} s'$ dénote une transition de SYS ; En particulier, une transition *temporelle* d'un système IF est notée par $s \stackrel{d}{\Rightarrow} s \oplus d$, et respectivement une transition *discrète* par $s \stackrel{a}{\Rightarrow} s'$. $a \in Act$ désigne une action observable d'une transition discrète $s \stackrel{a}{\Rightarrow} s'$. d est un intervalle de temps entre deux actions observables qui désigne un écoulement du temps de d unités par une transition temporelle $s \stackrel{d}{\Rightarrow} s \oplus d$.

Relation de Conformité

La conformité est définie ici comme une relation entre un système modélisant l'implantation et celui modélisant la spécification. Elle est définie ici en termes de traces ou d'observations permises par le test boîte grise des services composés. Nous nous plaçons dans une situation de test où nous pouvons qu'envoyer des interactions vers une boîte grise à tester (i.e. l'implantation d'un service composé SST), et analyser les réponses fournies par ce SST. Nous définissons dans cette section la notion de traces temporisées d'un système de processus communicants ainsi qu'une relation de conformité adaptée à ces systèmes. Nous commençons par introduire quelques concepts utilisés pour décrire formellement cette relation de conformité.

$\Gamma = Act \cup \mathbb{R}_+$ désigne un ensemble d'étiquettes observables : une action observable $a \in Act$ ou un intervalle $d \in \mathbb{R}_+$. $\Gamma_\tau = Act_\tau \cup \mathbb{R}_+$ étend l'ensemble Γ en incluant les actions internes non observables (notées par τ). Dans la suite, $s, s_0, s_1, ..., s_n \in S$ dénotent des états sémantiques de $[SYS]$.

Définition 5.1 (Séquence temporisée observable)**.** Soit $Seq(\Gamma) = (\Gamma)^*$ l'ensemble de toutes les séquences temporisées finies qui sont définies sur Γ. Une séquence temporisée $\sigma \in Seq(\Gamma)$ est une suite d'actions observables a ou d'intervalles de temps d. Une séquence vide est notée par $\sigma_\varepsilon \in Seq(\Gamma)$.

Soient $\Gamma' \subseteq \Gamma$ un sous ensemble de Γ et $\sigma \in Seq(\Gamma)$ une séquence temporisée définie sur Γ. $\pi_{\Gamma'}(\sigma)$ dénote la projection d'une séquence σ sur l'ensemble Γ' obtenue en supprimant de σ toutes les actions n'appartenant pas à Γ'. Soit $\sigma = \sigma_1.\sigma_2.\cdots.\sigma_n$ une séquence temporisée construite par la concaténation des séquences σ_i. La notation $s \stackrel{\sigma}{\Rightarrow}$ est utilisée pour indiquer qu'il existe un état s_n tel que $s \stackrel{\sigma_1}{\Rightarrow} s_1 \stackrel{\sigma_2}{\Rightarrow} s_2 \cdots \stackrel{\sigma_n}{\Rightarrow} s_n$.

Dans notre modélisation d'un processus BPEL par un système IF (cf. § 4.3), chaque activité temporelle de BPEL (e.g. `<wait>`) est modélisée par un processus IF utilisant sa propre horloge locale c_i en l'activant par une initialisation à *zéro* au début de son exécution et en la désactivant à la fin de son exécution (cf. Exemple 4.7). Cette horloge est utilisée dans les gardes temporelles des transitions du processus IF et permet de décrire des délais d'attente ou des timeouts.

Dans ce travail, nous considérons non seulement la réception d'un message (notée par $?m$), l'émission d'un message (notée par $!m$) et l'écoulement du temps comme étant une action observable mais aussi l'activation et la désactivation des horloges locales. Nous notons l'activation d'une horloge c par $c^{\uparrow}$ et sa désactivation par $c^{\downarrow}$. Ces deux actions sont prises en considération lors de la phase de génération de tests abstraits. Elles seront utilisées en particulier lors de la phase de concrétisation de tests (cf. § 5.6).

Nous introduisons ci-dessous des propriétés importantes des systèmes temporisés : les états atteignables et les systèmes déterministes.

Définition 5.2 (État atteignable)**.** Un état s est dit *atteignable* s'il est atteint à partir de l'état initial s_0 tel que : $\exists\, \sigma \in Seq(\Gamma) \mid s_0 \stackrel{\sigma}{\Rightarrow} s$.

Dans la suite, $Att(SYS)$ désigne l'ensemble de tous les états atteignables d'un système SYS.

Définition 5.3 (Système déterministe)**.** Un système SYS est dit *déterministe* si :

$$\forall\, s,s',s'' \in Att(SYS),\ \forall a \in Act_\tau,\ s \stackrel{a}{\Rightarrow} s' \ \wedge\ s \stackrel{a}{\Rightarrow} s'' \Longrightarrow s' = s''$$

Nous pouvons maintenant introduire la définition d'une trace observable temporisée considérée comme une séquence observable temporisée obtenue à partir de l'état initial du système.

Définition 5.4 (Trace temporisée observable)**.** Une trace temporisée observable σ' est une séquence observable définie par : $\sigma' = \pi_\Gamma(\sigma) \mid \sigma \in Seq(\Gamma_\tau) \ \wedge\ s_0 \stackrel{\sigma}{\Rightarrow}$. L'ensemble des traces temporisées d'un système SYS est défini par : $Traces(SYS) = \{\pi_\Gamma(\sigma) \mid \sigma \in Seq(\Gamma_\tau) \ \wedge\ s_0 \stackrel{\sigma}{\Rightarrow}\}$.

Hypothèse 2. Nous supposons que la spécification d'un service composé et son implantation sous test (SST) peuvent être décrites par des systèmes IF *déterministes* définis sur le même ensemble d'actions et notés respectivement par SYS_S et SYS_I.

Dans notre méthode de test des services Web composés, nous proposons une relation de conformité temporisée notée par $\preccurlyeq_{Traces}$. Cette relation est définie comme l'inclusion de traces temporisées où chaque trace observable de la spécification SYS_S doit être obligatoirement une trace observable de l'implantation SYS_I. Elle exprime que SYS_I est conforme à SYS_S si SYS_I fait au moins ce qui est prévu par SYS_S ; en d'autres termes, SYS_I doit avoir au minimum tous les comportements prévus par la spécification SYS_S, mais a la liberté de faire plus.

Définition 5.5 (Relation de conformité $\preccurlyeq$)**.** La relation de conformité $\preccurlyeq_{Traces}$ est définie par :

$$\big(SYS_I \preccurlyeq_{Traces} SYS_S\big) \iff \big(Traces(SYS_S) \subseteq Traces(SYS_I)\big)$$

Note 5.2. La relation de conformité $\preccurlyeq_{Traces}$ est transitive :

$$(SYS_1 \preccurlyeq_{Traces} SYS_2) \wedge (SYS_2 \preccurlyeq_{Traces} SYS_3) \Longrightarrow (SYS_1 \preccurlyeq_{Traces} SYS_3)$$

Inversement, on pourrait exiger que les comportements de l'implantation SYS_I ne soient jamais en désaccord avec ceux de la spécification SYS_S. Nous pouvons maintenant étendre cette relation d'inclusion de traces $\preccurlyeq_{Traces}$ en une relation d'équivalence de traces $\cong_{Traces}$ qui exprime que la spécification et l'implantation doivent avoir les mêmes comportements.

Définition 5.6 (Relation de conformité $\cong$)**.** La relation de conformité $\cong_{Traces}$ est définie par :

$$\big(SYS_I \cong_{Traces} SYS_S\big) \iff \big(Traces(SYS_S) \subseteq Traces(SYS_I)\big) \wedge \big(Traces(SYS_I) \subseteq Traces(SYS_S)\big)$$

Étant donné que notre méthode de test est basée sur le test boîte grise et que nous avons aucune connaissance de l'implantation SYS_I, nous ne pouvons pas utiliser l'équivalence des traces comme relation de conformité. Nous relâchons donc cette relation et nous utilisons l'inclusion des traces temporisées $\preccurlyeq_{Traces}$ définie ci-dessus comme relation de conformité où les traces temporisées sont sélectionnées par des objectifs de test.

Nous donnons ci-dessous un exemple pour illustrer notre relation de conformité.

Exemple 5.1 (Conformité entre spécification et implémentation)**.** Dans cet exemple, les actions de réception sont paresseuses (*lazy*) à la différence des actions d'envoi qui sont urgentes (*eager*). Nous considérons la spécification `Spec` qui est décrite textuellement par : « le système doit attendre la réception d'un message `a` au maximum pendant `4` unités de temps et doit envoyer à la suite un message $\texttt{b}_1$. S'il ne reçoit rien après `4` unités, il doit envoyer un message $\texttt{b}_2$. La durée d'attente de `5` unités est considérée ici comme un *timeout*. ». Cette spécification et les différentes implémentations sont illustrées dans la Figure 5.7. Étant donné la relation de conformité $\preccurlyeq_{\texttt{Traces}}$ définie ci-dessus, l'implantation $\texttt{Impl}_1$ est conforme à la spécification `Spec` car toute trace de `Spec` est une trace de $\texttt{Impl}_1$. $\texttt{Impl}_2$ et $\texttt{Impl}_3$ ne sont pas conformes à `Spec`. La trace $c^{\uparrow}\ ?a\ c^{\downarrow}\ !b_1$ de `Spec` n'est pas une trace de $\texttt{Impl}_2$ car $\texttt{Impl}_2$ après avoir reçu a envoie $\texttt{b}_1$ au bout de `2` à `4` unités de temps ($2 \leq c \leq 4$). Concernant $\texttt{Impl}_3$, la trace $c^{\uparrow}\ 4\ ?a\ c^{\downarrow}\ !b_1$ de `Spec` n'est pas une trace de $\texttt{Impl}_3$ car, d'une part, le message `a` est envoyé après `4` unités de temps, et d'autre part, $\texttt{Impl}_3$ ne peut attendre la réception de ce message `a` que `3` unités de temps au maximum ($c \leq 3$).

Note 5.3. Il existe dans la littérature d'autres relation de conformité : *tioco* [155] (une extension de la relation *ioco* [177] pour les systèmes temporisés), *rtioco* (relation de conformité *tioco* relativisée) [178], relation de bisimulation temporisée [179]. Nous citerons aussi les travaux de Berrada et al. [180] qui ont abordé l'inclusion des traces pour les systèmes temps réel. Berrada et al. ont proposé le concept des *traces temporisées bornées* (timed bound traces) et ont ramené le problème d'inclusion des traces temporisées en une inclusion de traces temporisées bornées. Nous comptons nous inspirer de ces travaux pour étendre notre relation d'inclusion des traces $\preccurlyeq_{Traces}$ décrite ci-dessus.

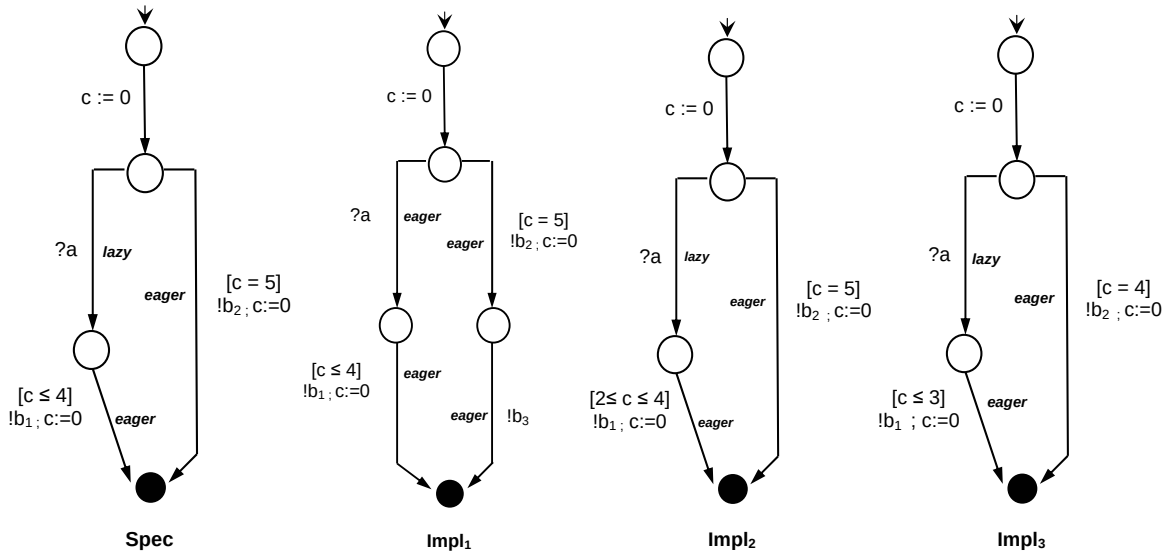

FIGURE 5.7 – Conformité entre Spécification et Implémentations

Objectif de test et cas de test

Un objectif de test décrit une fonctionnalité particulière de l'implantation sous test. Il est utilisé pour dériver efficacement un cas de test en particulier pour des modèles formels de grande taille. Il permet parfois d'éviter le problème d'explosion du nombre d'états car nous prenons en considération qu'une partie du modèle de test dont les transitions satisfont cet objectif. Dans notre méthode de test, un objectif de test est ensemble (ordonné) de conditions logiques où chaque condition est associée à une seule transition d'un processus du système. Chaque condition est une conjonction de contraintes logiques portant éventuellement sur : (i) l'état source et/ou destination d'une transition, (ii) ses actions associées (envoi/réception de signaux ou de messages), (iii) la valeur courante des variables et des horloges locales dans l'état source de cette transition. Actuellement, ces objectifs sont crées manuellement à partir des exigences de conformité (exigences fonctionnelles et/ou temporelles).

Un cas de test temporisé est une trace temporisée observable pouvant valider les exigences de conformité décrites par les objectifs de test. L'exécution de ce cas de test permet de produire trois différents verdicts envisageables : *pass* (réussi), *fail* (échec) ou *inconclusive* (inconcluant). Ce dernier verdict peut indiquer que l'implantation du service sous test est passée par un chemin qui n'était pas décrit par le cas de test.

Dans la section suivante, nous détaillons notre algorithme de génération de test temporisés guidée par un ensemble d'objectifs de test.

5.5.2 Algorithme de génération automatique de test temporisé

Cet algorithme a en entrée une spécification formelle d'un service composé (i.e. un système de processus communicants de IF) et utilise une stratégie d'exploration partielle de l'espace d'état appelée Hit-Or-Jump [6]. Étant donné que cette stratégie Hit-Or-Jump a été définie sur des machines d'états finis étendues (EFSM), nous l'avons bien adapté aux systèmes d'automates temporisés de IF.

Afin d'éviter le problème d'explosion combinatoire du nombre d'états, cet algorithme effectue une exploration partielle — parcours en largeur d'abord (BFS) ou parcours en profondeur d'abord (DFS) — de l'espace d'états du système et construit à la volée une séquence de test temporisée satisfaisant les objectifs de test. Lors de chaque phase d'exploration, nous gardons en mémoire qu'une partie du système construite à la volée. Cet algorithme est illustré dans la Figure 5.8.

À partir de l'état initial du système, nous conduisons une exploration partielle en explorant toutes les transitions franchissables. Chaque phase d'exploration permet de construire un arbre de recherche partielle (noté ARP) et de vérifier la satisfaction des objectifs de test. Cette phase s'arrête dans l'un des deux cas suivants :

- Cas *Hit* : si au moins un objectif est satisfait par une transition t. Dans ce cas, on met à jour la séquence de test et la liste des objectifs de test à satisfaire. Ensuite, on recommence une nouvelle exploration à partir de l'état destination de t ;
- Cas *Jump* : si on atteint la profondeur limite d'exploration sans satisfaire aucun objectif. Dans ce cas, on choisit aléatoirement une feuille de l'arbre ARP construit lors cette phase, on met à jour la séquence de test et on recommence une nouvelle exploration à partir de cette feuille.

L'algorithme se termine si tous les objectifs de test sont satisfaits ou s'il n y a aucune transition franchissable à explorer. En cas de satisfaction de tous les objectifs de test, une séquence de test représentera le chemin construit à la volée à partir de l'état initial du système jusqu'à l'état destination de la transition satisfaisant le dernier objectif de test. Un cas de test temporisé est obtenu à partir d'une séquence de test en la filtrant et en ne gardant que les événements observables, à savoir les actions de réception et d'émission de messages, les intervalles de temps, l'initialisation et la remise à *zéro* des horloges. Ces tests seront utilisés pour vérifier la conformité de l'implantation du service composé à sa spécification par rapport aux exigences décrites par les objectifs de test. Cet algorithme est décrit formellement ci-dessous. Nous l'avons implanté dans l'outil de génération de test temporisé TestGen-IF (cf. § 6.3.4).

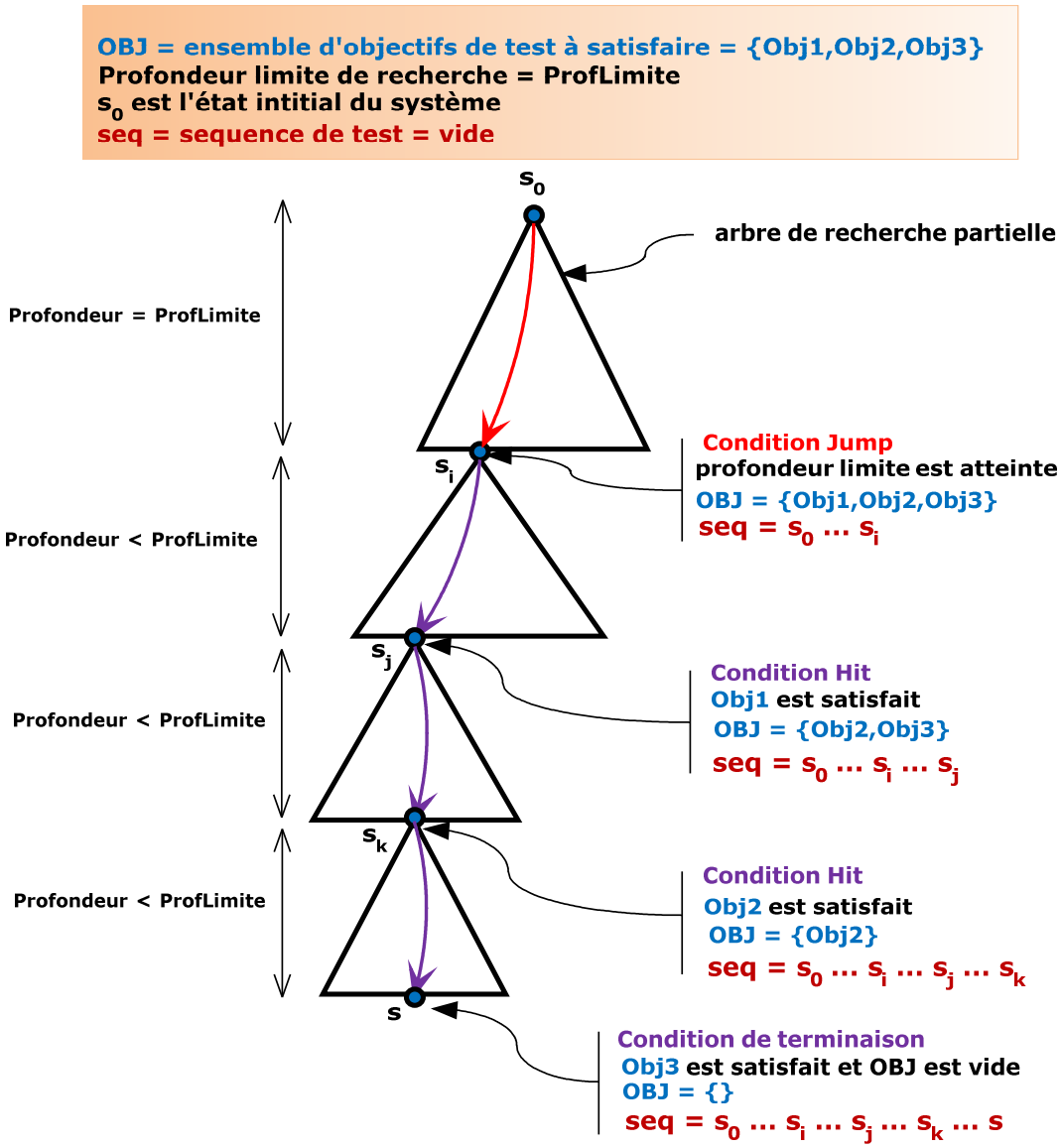

FIGURE 5.8 – Génération automatique de tests temporisés

Algorithme 5.1 : Algorithme de génération automatique de cas de test temporisés

Données :

- Un système représentant la spécification d'un service composé :SYS ;
- L'ensemble des objectifs de test à satisfaire : OBJ ;
- La profondeur limite de recherche partielle : $profLim$.

Résultat : sequence de test temporisée : seq.

Initialisation :

- *SYS est dans l'état initial* s_0 ;
- $OBJ := \{Obj_1, Obj_2, \cdots, Obj_n\}$;
- s_0 *est la racine de l'arbre de recherche partielle (ARP) :* $r := s_0$;
- $ARP = \{r\}$;
- *La séquence de test initiale seq est vide :* $seq := \sigma_\varepsilon$.

Terminaison : *L'algorithme se termine dès la satisfaction de tous les objectifs de test* ($OBJ = \emptyset$.) ou s'il n y a aucune transition franchissable à explorer.

Exécution :

répéter
  répéter
    depuis la racine r, conduire une recherche partielle en explorant toutes les transitions franchissables : transition discrète $s_i \stackrel{a}{\Rightarrow} s_{i+1}$ *ou transition temporelle* $s_i \stackrel{d}{\Rightarrow} s_i \oplus d$
  jusqu'à ⓐ $\vee$ ⓑ $\vee$(*"aucune transition à explorer"*)

  ⓐ **si** *une transition* $t = s \stackrel{l}{\Rightarrow} s_{hit}$ *satisfait au moins un objectif de test :* $\exists k.\ t \models obj_k$ **alors**
    Hit :
      – **pour** *tout* $k.\ t \models Obj_k$ **faire** $OBJ := OBJ \setminus \{Obj_k\}$
      – *concaténer à la séquence de test seq le chemin allant de r à* s_{hit} *:* $r \stackrel{\sigma}{\Rightarrow} s \stackrel{l}{\Rightarrow} s_{hit} \wedge seq := seq.\sigma.l$
      – *mettre à jour l'arbre ARP et sa racine :* $r := s_{hit} \wedge ARP = \{r\}$
    fin Hit

  ⓑ **si** $profLimit$ *est atteinte* **alors**
    Jump :
      – *l'arbre ARP est déjà construit, de profondeur* $profLim$ *et ayant comme racine r ;*
      – *examiner toutes les feuilles de ARP et sélectionner une feuille* s_{jump} *aléatoirement ;*
      – *concaténer à la séquence de test seq le chemin allant de l'état r à* s_{jump} *:* $r \stackrel{\sigma}{\Rightarrow} s_{jump} \wedge seq := seq.\sigma$
      – *mettre l'arbre ARP et sa racine :* $r := s_{jump} \wedge ARP = \{r\}$
    fin Jump
jusqu'à $(OBJ = \emptyset) \vee$ (*"aucune transition à explorer "*)

si $OBJ = \emptyset$ **alors**
  retourner la séquence de test : seq
sinon
  afficher "échec de génération"

Dans la section suivante, nous allons décrire notre méthode de concrétisation des cas de test abstraits en adéquation avec les deux architectures de test proposées ci-dessus.

5.6 Concrétisation de cas de test

Nous rappelons qu'un cas de test abstrait est une suite d'actions observables : (i) $!m$: envoi d'un message m, (ii) $?m$: réception d'un message m, (iii) $c^{\uparrow}$: activation d'une horloge c, (vi) $c^{\downarrow}$: désactivation d'une horloge c, (v) d : un intervalle de temps entre actions observables.

Nous introduisons en premier l'exemple d'une spécification BPMN [111] d'un service sous test (SST) ainsi que deux cas de test que nous utiliserons comme exemples pour cette phase de concrétisation. Nous détaillons, dans l'exemple 5.2, la spécification du service sous test `SST` dont le modèle abstrait a été décrit ci-dessus dans la Figure 5.2.

Exemple 5.2 (Un service sous test)**.** Nous considérons le service sous test `SST` dont la spécification informelle est représentée en notation BPMN dans la Figure 5.9. Ce `SST` a deux partenaires notés respectivement par `SP1` et `SP2`. L'interaction du `SST` avec son premier partenaire `SP1` est asynchrone utilisant un timeout ($timeout_1$) alors que son interaction avec son deuxième partenaire `SP2` est synchrone. `SST` est initié par la réception d'un message (ou une requête) `a1`. Il invoque ensuite son premier partenaire `SP1` et attend sa réponse. Si après `6` unités de temps ($timeout_1$) il ne reçoit aucune réponse de la part de `SP1`, alors il envoie un message d'erreur `b4` à son client et arrête son exécution. Dans le cas contraire (réception d'un message `a2`), il invoque de manière synchrone son deuxième partenaire `SP2` et reçoit instantanément soit une réponse (i.e. message `a3`) ou un message d'erreur `a4`. S'il reçoit `a3`, il envoie une réponse à la requête de son client (i.e. message `b3`) et termine normalement son exécution, sinon il envoie un message d'erreur `b4` et arrête son exécution.

Remarque 5.2. Dans notre transformation de BPEL en IF décrite dans le Chapitre 4, chaque message BPEL est transformé en un `signal` IF de la forme `signal mess(plType,messType)` où `plType` est un type de lien partenaire et `messType` est un type de message (cf. § 4.3.2). `plType` permet de déterminer la source ou la destination du message échangé (client ou partenaire). Dans la suite de ce chapitre, nous notons — par simplification — ces messages par `a`,`b`, ...

Nous avons utilisé la spécification précédente pour générer deux cas de test abstraits qui sont décrits dans l'exemple 5.3 et qui seront utilisés comme exemples pour illustrer notre méthode de concrétisation des tests abstraits.

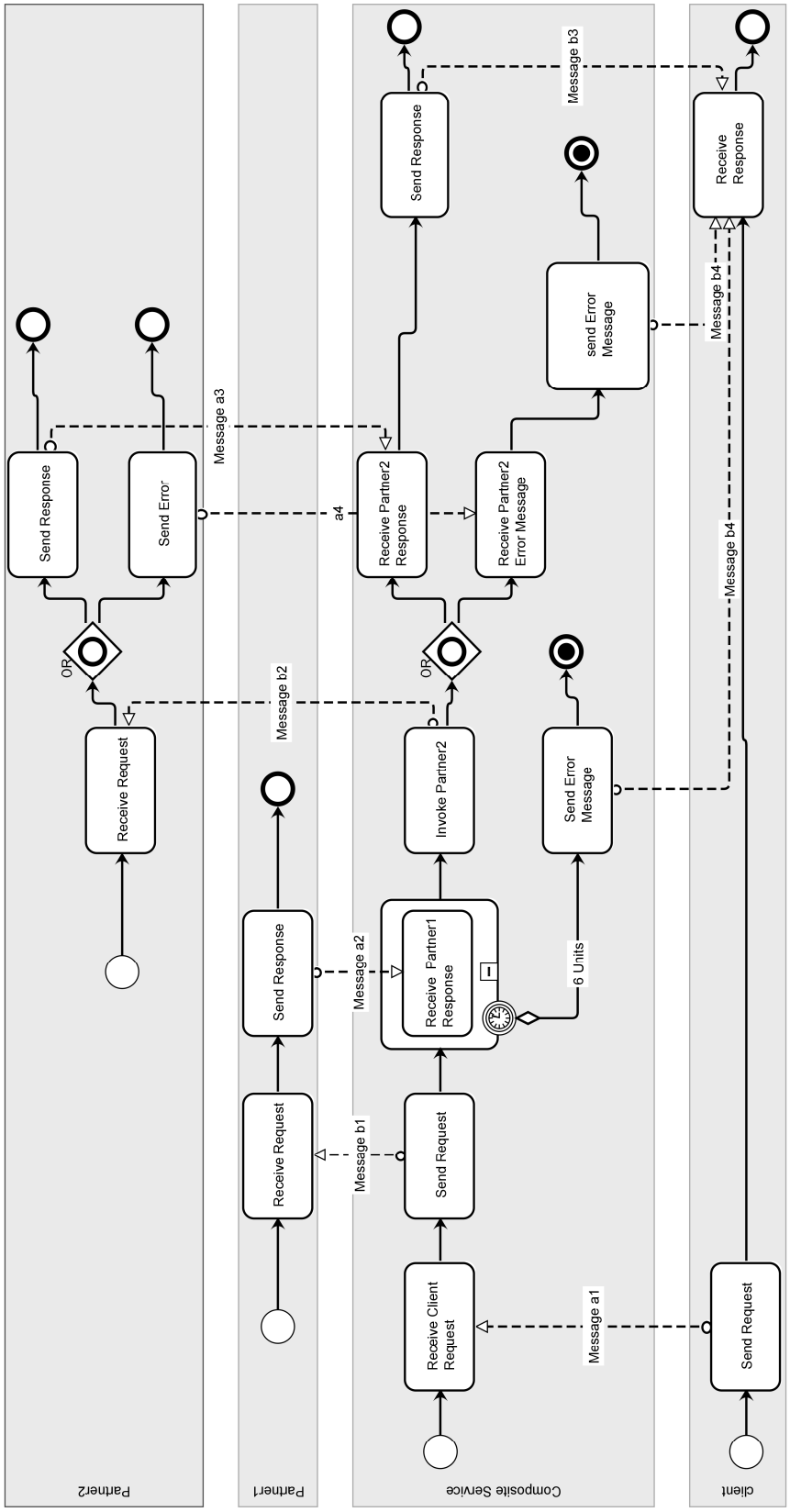

FIGURE 5.9 – Spécification informelle BPMN d'un service sous test SST

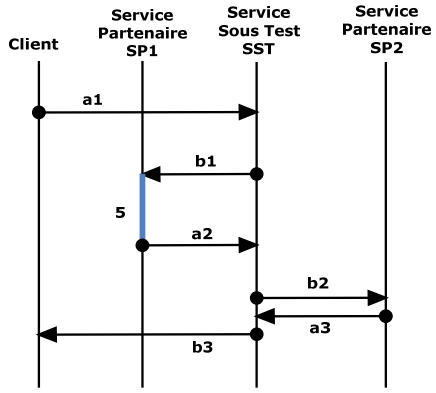

FIGURE 5.10 – Cas de test abstrait seq_1

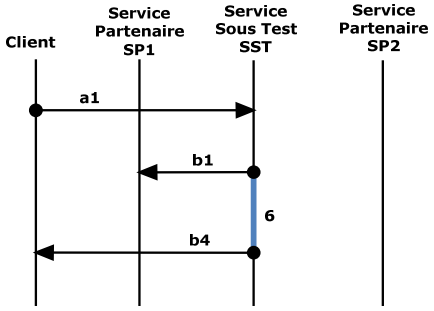

FIGURE 5.11 – Cas de test abstrait seq_2

Exemple 5.3 (Cas de tests abstraits)**.** Nous présentons deux cas de tests, notés respectivement par seq_1 et seq_2, qui sont illustrés respectivement dans la Figure 5.10 et la Figure 5.11 où :

$$seq_1 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 5 \;\; ?a2 \;\; c_1^{\downarrow} \;\; !b2 \;\; ?a3 \;\; !b3$$
$$seq_2 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 6 \;\; c_1^{\downarrow} \;\; !b4$$

seq_1 décrit le comportement du `SST` suivant : « `SP1` répond au `SST` après un écoulement de temps de `5` unités (inférieur au $timeout_1 = 6$), de son coté `SP2` répond instantanément au `SST` par un message `a3`. Enfin, `SST` termine son exécution par l'envoi d'un message `b3` en réponse à la requête de son client. » Dans seq_2, `SP1` se comporte différemment car il ne répond pas à l'invocation du `SST`. Par conséquent, ce dernier envoie un message d'erreur `b4` à son client après une attente de `6` unités ($timeout1$).

Nous pouvons maintenant détailler notre approche de concrétisation de tests abstraits.

5.6.1 Approche de concrétisation

Cette approche consiste à transformer un cas de test abstrait en un test décrivant : (i) le comportement du service testeur composé dans le contexte d'une architecture de test centralisée, (ii) le comportement de chaque service testeur dans le contexte d'une architecture de test distribuée, (iii) les messages d'entrée du service sous test (SST), (iv) les messages de sortie attendus. Ces comportements sont décrits en termes d'attentes et d'interactions telles qu'elles ont été définies dans la Section 5.4.4.

Notons que nous utilisons notre propre syntaxe pour exprimer les tests concrets mais il est possible de les exprimer dans un langage standard tel que TTCN-3 [95]. Nous nous sommes inspirés de la description des services testeurs de la plateforme BPELUnit [106] que nous avons étendu et adapté au test boîte grise et au test des propriétés temporelles de BPEL.

Nous explicitons dans les sections suivantes, la dérivation — à partir d'un cas de cas de test abstrait — des interactions du client d'un SST et de ses partenaires, de leurs délais d'attente et des messages échangées.

Dérivation des interactions de testeurs

A partir d'un cas de test abstrait, nous pouvons extraire les interactions qui ont été décrites dans la section 5.4.4, en particulier les interactions simples ou bilatérales synchrones. Les interactions bilatérales asynchrones peuvent être considérées comme une suite d'interactions simples et d'attentes que nous allons aborder lors de la dérivation des délais d'attente. Nous transformons ces interactions en un comportement du client ou des partenaires du SST de la manière suivante :

envoi/réception synchrone c'est une suite de deux actions successives $(?m1, !m2)$ où les deux messages $m1$ et $m2$ correspondent à la même opération WSDL [5]. Cette interaction est transformée en une action `SendReceive(m1,m2)` qui envoie une requête `m1` au SST, reçoit immédiatement une réponse `m2` et vérifie son contenu par rapport au contenu du message attendu. Si ce dernier est erroné ou si elle ne reçoit aucun message, `SendReceive` affecte `fail` à la variable `verdict` du contrôleur de test SC (par l'action `SetVerdict(fail)`) et arrête l'exécution du testeur (par l'action `Stop`).

réception/envoi synchrone c'est une suite de deux actions successives $(!m1, ?m2)$ où les deux messages $m1$ et $m2$ correspondent à la même opération WSDL. Cette interaction est transformée en une action `ReceiveSend(m1,m2)` qui attend la réception d'un message `m1` et vérifie son contenu. Si ce dernier est erroné ou si elle ne reçoit aucun message, alors elle exécute successivement `SetVerdict(fail)` et `Stop`, sinon elle envoie un message synchrone `m2` au SST.

envoi d'un message c'est une action $?m$ qui est transformée en une action `Send(m)` qui envoie un message `m` au SST.

réception d'un message c'est une action $!m$ qui est transformée en une action `Receive(m)` qui reçoit un message `m` et vérifie son contenu. Si ce contenu est erroné ou si elle ne reçoit aucun message, `Receive(m)` exécute successivement `SetVerdict(fail)` et `Stop`.

Les messages envoyés par les testeurs sont tout ceux qui sont contenus dans le cas de test abstrait et censés être reçus par le service sous test (SST). La vérification du contenu des messages reçus par ces testeurs se fait par rapport au contenu des messages attendus qui sont censés être envoyés par le SST.

Dérivation des délais d'attente

Les interactions entre un service sous test (SST), son client et ses partenaires peuvent être asynchrones et probablement associées à un *timeout* (comme dans l'exemple 5.2 entre le service `SST` et son premier partenaire `SP1`).

Du point de vue du testeur, nous pouvons distinguer deux types d'attente :

1. une attente avant l'envoi d'un message (une réponse) au SST. C'est le cas du testeur `ST1` qui simule le premier service partenaire `SP1` dans la Figure 5.10 et qui doit attendre `5` unités de temps avant d'envoyer son message `a2` au `SST` ;
2. une attente avant la réception d'un message envoyé par le SST. C'est le cas du testeur `ST1` dans la Figure 5.11 qui doit attendre `6` unités de temps avant de recevoir le message `b4`.

Comme nous l'avons précisé dans la Section 4.3.6, toutes les actions de réception de messages provenant de l'environnement sont paresseuses et ne bloquent pas l'écoulement du temps. Afin d'extraire les délais d'attente, nous utilisons les actions d'activation et de désactivation d'horloges que nous avons considérées comme étant des actions observables (cf. § 5.1).

Étant donné un cas de test *seq*, nous définissons une fonction $delai(seq, c_i)$ qui calcule la somme de tous les intervalles de temps de *seq* figurant entre l'activation d'une horloge c_i ($c_i^{\uparrow}$) et sa désactivation ($c_i^{\downarrow}$). Cette fonction permet de calculer les délais d'attente de chaque testeur de la manière suivante :

– si l'action $c_i^{\downarrow}$ est précédée dans *seq* par une action de réception d'un message m (i.e. $?m$), $delai(seq, c_i) = d$ détermine un délai d'attente du service testeur (ST) de d unités avant l'envoi de m. Cette attente est transformée en une action `Wait(d)` qui doit être suivie d'une action `Send(m)` ;

– si l'action $c_i^{\downarrow}$ n'est pas précédée dans *seq* par $?m$ et est suivie par une action d'envoi d'un message m (i.e. $!m$), $delai(seq, c_i) = d$ détermine un délai d'attente du ST de d unités avant la réception d'un message m. Durant cette attente, le ST ne doit recevoir aucun message. Cela est traduit par l'utilisation d'une action `Pick(d)` ayant deux sous-actions (`onMessage` et `onAlarm`) et qui attend la réception d'un message ou le déclenchement d'une alarme. La réception d'un message `m` est effectuée par une action `onMessage(m)` qui exécute successivement les deux actions `SetVerdict(fail)` et `Stop` sans aucune vérification du contenu de `m`. Quand à `onAlarm(d)`, elle déclenche une alarme après l'écoulement de `d` unités à partir du début de l'exécution de `Pick` et met fin à l'exécution de cette dernière. Notons que `Pick` doit être suivie d'une action `Receive`.

Remarque 5.3. La simple utilisation des intervalles de temps dans un cas de test sans aucune information sur l'activation et la désactivation des horloges n'est pas suffisante pour déterminer les délais d'attente de chaque testeur.

Dans l'exemple 5.4, nous donnons les délais d'attente calculés à partir des cas de test décrits dans l'exemple 5.3 et que nous utiliserons dans leur concrétisation.

Exemple 5.4 (Délais d'attente)**.** Nous détaillons dans la Table 5.1 les délais d'attente associés aux deux cas de test seq_1 et seq_2. $delai_1$ définit un délai d'attente de `5` unités du service partenaire `SP1` avant l'envoi du message `a2` (cf. Fig. 5.10), et $delai_2$ définit un délai d'attente de `6` unités après lequel le client doit recevoir le message d'erreur `b4` (cf. Fig. 5.11).

Dans la Table 5.2 ci-après, nous résumons l'ensemble des actions décrivant le comportement d'un service testeur.

Délai	Définition	Valeur	Délai d'attente Avant envoi	Délai d'attente Avant réception
$delai_1$	$delai(seq_1, c_1)$	5	✓	
$delai_2$	$delai(seq_2, c_1)$	6		✓

TABLE 5.1 – Délais d'attente des cas de test : seq_1 et seq_2

Action	Description
`Send(m)`	Envoi d'un message `m`
`Receive(m)`	Réception d'un message `m` et vérification de son contenu
`SendReceive(m1,m2)`	Envoi d'un message `m1`, réception d'un message `m2` et vérification de son contenu
`ReceiveSend(m1,m2)`	Réception d'un message `m1`, vérification de son contenu et envoi d'un message `m2`
`SetVerdict(v)`	Affecte `v` à la variable `verdict` du service testeur
`Stop`	Arrête l'exécution d'un testeur
`onMessage(m)`	Attente d'un message `m` suivie d'une notification d'un échec (`fail`) et arrêt de l'exécution du service testeur
`onAlarm(d)`	Déclenchement d'une alarme après un délai d'attente de `d` unités de temps
`Wait(d)`	Attente du service testeur de `d` unités de temps
`Pick(d)`	Déclenchement d'une alarme après une attente de `d` unités de temps ou réception d'un message `m` suivie par la notification d'un échec

TABLE 5.2 – Actions d'un service testeur

Dérivation des messages

Un cas de test abstrait contient l'ensemble d'actions observables (e.g. envoi et réception de messages) et décrit aussi les messages et les données échangées entre le service sous test (SST) et son environnent (i.e. client et partenaires). Du point de vue d'un service testeur (ST), nous pouvons distinguer deux types de messages échangés :

- **Les messages d'entrée** : sont ceux envoyés par le service SST. Le service testeur doit vérifier leurs contenus par rapport aux messages attendus décrits par le cas de test en tant que paramètres des actions d'envoi ;
- **Les messages de sortie** : sont ceux envoyés au service SST par le service testeur et qui sont décrits dans le cas de test en tant que paramètres des actions de réception.

Rappelons qu'un message WSDL peut être constitué de plusieurs parties logiques [5], chacune pouvant être d'un type différent — élément d'un schéma XML ou type (simple ou complexe) d'un schéma XML. Comme nous l'avions mentionné lors de la transformation des messages WSDL utilisés par un processus BPEL (cf. § 4.3.2), chaque message est transformé en un signal IF de la forme *signal mess(plType,messType)* où : (i) *plType* est un type de lien partenaire, i.e. une énumération de la forme *pl.pt.op* où *pl* est un lien partenaire, *pt* est un *portType* et *op* une opération WSDL, (ii) *messType* est un type (complexe) de IF associé au type du message WSDL *mess* et décrivant les données transportées par ce message.

A partir d'un cas de test abstrait, nous pouvons générer l'ensemble des messages attendus par un service testeur et des messages de sortie — envoyés au service SST — de la manière suivante :

- pour une réception d'un message *?mess(pl.pt.op,v)* : *v* est transformé en un arbre XML ayant la même structure que le message WSDL *mess* et qui sera utilisé pour générer le message SOAP de sortie envoyé par le service ST au service SST ;
- pour un envoi d'un message *!mess(pl.pt.op,v)* : *v* est transformé en un arbre XML ayant la même structure que le message WSDL *mess* et qui sera utilisé pour générer le message SOAP attendu par le service ST.

Les message SOAP sont générés à partir des interfaces WSDL, des arbres XML et des liens partenaires (lien partenaire, type de port et opération). Les messages d'entrée du service testeur doivent être comparés aux messages attendus. Nous pouvons effectuer cette comparaison de deux manières différentes :

1. extraire l'arbre XML du message SOAP reçu par le testeur et le comparer à celui du message attendu ;
2. générer le message SOAP du message attendu et le comparer au message SOAP reçu par le testeur.

Toutes les données d'un messages attendu ne sont pas pertinentes ou nécessaires pour le test de conformité. Dans ce cas, nous limitons cette comparaison à certaines données qui sont décrites ou sélectionnées par des expressions XPath [58] portant sur les arbres XML.

Description du service contrôleur

Le service contrôleur (SC) permet de gérer l'exécution des services testeurs (STs) et d'émettre un verdict. Il interagit avec ces testeurs par l'intermédiaire de deux messages de contrôle : `startTest` et `stopTest`. Le premier message sert à déclencher l'exécution d'un testeur alors que le deuxième est utilisé pour l'arrêter. SC utilise une variable globale `verdict` qui peut être modifiée par lui même ou par un service testeur. Cette variable est initialisée à `null` et peut être affectée à trois variables éventuelles : `pass`, `fail` ou `inconclusive`.

Un testeur ne peut notifier qu'un `fail` (i.e. affecter `fail` à `verdict`) en exécutant l'une des actions suivantes : **`Receive`**, **`SendReceive`**, **`ReceiveSend`** et **`Pick`** ; Ceci dans l'un des cas suivant : (i) réception d'un message avec un contenu erroné, (ii) absence d'un message attendu, (iii) réception d'un message non attendu (cf. § 5.4.5). Les deux premiers types d'erreur peuvent être détectés par les actions **`Receive`**, **`SendReceive`**, **`ReceiveSend`** alors que le troisième type d'erreur est détecté par l'action **`Pick`** (et plus précisément par son action **`onMessage`**).

C'est au SC de notifier un `pass` en utilisant l'action **`SetVerdict(pass)`** à la fin de l'exécution normale de tous les testeurs, i.e. aucune notification d'un échec (`fail`). Dès la notification d'un échec par un testeur, SC émet `fail` comme étant le verdict du test et arrête l'exécution de tous les testeurs encore actifs — qui n'ont pas encore terminé l'exécution de leurs actions.

Nous illustrons, dans la Figure 5.12 et la Figure 5.13, la description du service contrôleur (SC) pour les deux types d'architecture :

SC pour une architecture de test centralisée Le service contrôleur (SC) initialise sa variable `verdict` à `null`, déclenche l'exécution du service testeur composé (ST) en envoyant un message `startTest` et attend sa terminaison ou la notification d'un échec (`fail`). Dans le cas d'une terminaison, SC notifie un `pass` et l'émet comme verdict du test, sinon il émet un `fail`. L'émission du verdict est effectuée par `Display(verdict)`. Notons que pour ce type d'architecture de test, SC n'a pas besoin d'arrêter l'exécution du testeur ST puisque ce dernier termine lui même son exécution (cf. Exemple 5.14).

SC pour une architecture de test distribuée Dans une architecture distribuée, SC se comporte comme pour une architecture centralisée, à la différence qu'il déclenche plusieurs services testeurs et arrête l'exécution des testeurs actifs en cas d'une notification d'un échec.

```
1. verdict := null ;
2. Send(startTest) to ST ;
3. repeat true until (verdict ≠ null ∨ termination(ST)) ;
4. if termination(ST) then SetVerdict(pass) ;
5. Display(verdict) ;
```

FIGURE 5.12 – Testeur contrôleur pour une architecture centralisée

```
1. verdict := null ;
2. Send(startTest) to all ST_i ;
3. repeat true until (verdict ≠ null ∨ ⋀_i termination(ST_i)) ;
4. if ⋀_i termination(ST_i) then SetVerdict(pass)
   else Send(stopTest) to all active TS_j ;
5. Display(verdict) ;
```

FIGURE 5.13 – Testeur contrôleur pour une architecture distribuée

Dans la suite, nous explicitons la concrétisation des deux cas de test seq_1 et seq_2.

5.6.2 Concrétisation des cas de test seq_1 et seq_2

Concrétisation de seq_1

A partir du cas de test $seq_1 = ?a1 \;\; c_1^{\uparrow} \;\; !b1 \;\; 5 \;\; ?a2 \;\; c_1^{\downarrow} \;\; !b2 \;\; ?a3 \;\; !b3$, nous pouvons dériver le comportement de(s) service(s) testeur(s) pour chaque architecture de test. Chaque action de seq_1 est transformée en une action de(s) testeur(s).

Pour une architecture de test centralisée (ATC), un cas de test concret consiste à définir le comportement du service testeur composé ST illustré dans la Figure 5.14. L'action de réception $?a1$ de seq_1 est transformée en une action d'envoi `Send(a1)` du ST. L'action d'envoi $!b1$ est transformée à son tour en une action `Receive(b1)` du ST qui non seulement attend la réception d'un message b1 mais vérifie aussi son contenu. Dans seq_1, $c_1^{\downarrow}$ est précédée par $?a2$, « $delai(seq_1, c1) = 5$ » détermine alors un délai d'attente de 5 unités de temps avant l'envoi de a2. Cela est traduit par l'utilisation de `Wait(5)` permettant ainsi de différer l'envoi de a2. `Wait` est suivie par l'action `Send(a2)`. Le couple $(!b2, ?a3)$ est transformé en une action `ReceiveSend(b2,a3)` du ST qui attend la réception de b2, vérifie son contenu et envoie a3 au SST. Ce test concret termine son exécution par l'action `Stop`.

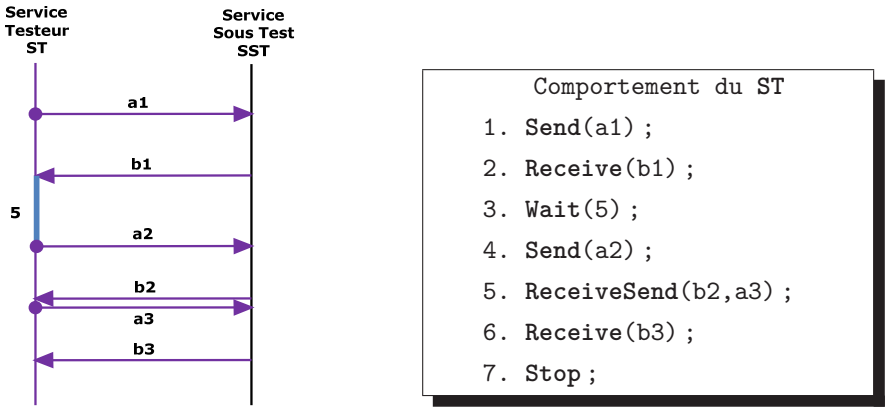

FIGURE 5.14 – Concrétisation de seq_1 pour une architecture ATC

Pour une architecture de test distribuée (ATD), un cas de test concret consiste à définir le comportement des trois services testeurs ST0, ST1 et ST2 simulant respectivement le client et les deux partenaires du SST. Ces testeurs sont illustrés dans la Figure 5.15 et sont décrits comme suit :

- ST0 commence par envoyer a1 au SST (i.e. `Send(a1)`), attend la réception de b3 et vérifie son contenu (i.e. `Receive(b3)`) avant de terminer son exécution ;
- ST1 attend la réception de b1 et vérifie son contenu (i.e. `Receive(b1)`). Ensuite, il attend 5 unités de temps (`Wait(5)`) avant d'envoyer a2 au SST (i.e. `Send(a2)`) et de terminer son exécution ;
- ST2 reçoit b2, vérifie son contenu et répond immédiatement en envoyant a3 au SST (i.e. `ReceiveSend(b2,a3)`) avant de terminer son exécution.

Concrétisation de seq_2

La concrétisation de $seq_2 = ?a1\ c_1^{\uparrow}\ !b1\ 6\ c_1^{\downarrow}\ !b4$ pour une architecture ATC est présentée dans la même Figure 5.16 étant donné que nous simulons un seul service partenaire. Dans seq_2, $c_1^{\downarrow}$ est suivi par $!b4$, « $delai(seq_2, c1) = 6$ » détermine alors un délai d'attente de 6 unités de temps après lequel on doit observer la réception du message b4. Comme nous l'avons mentionné lors de la dérivation des délais d'attente, durant cette attente, ST ne doit recevoir aucun message. Cela est réalisé par l'utilisation de l'action `Pick` et de ses deux sous-actions `onMessage(m)` et `onAlarm(6)`. Cette action `Pick` est suivie par l'action `Send(b4)`. Ce cas de test termine son exécution par l'action `Stop`.

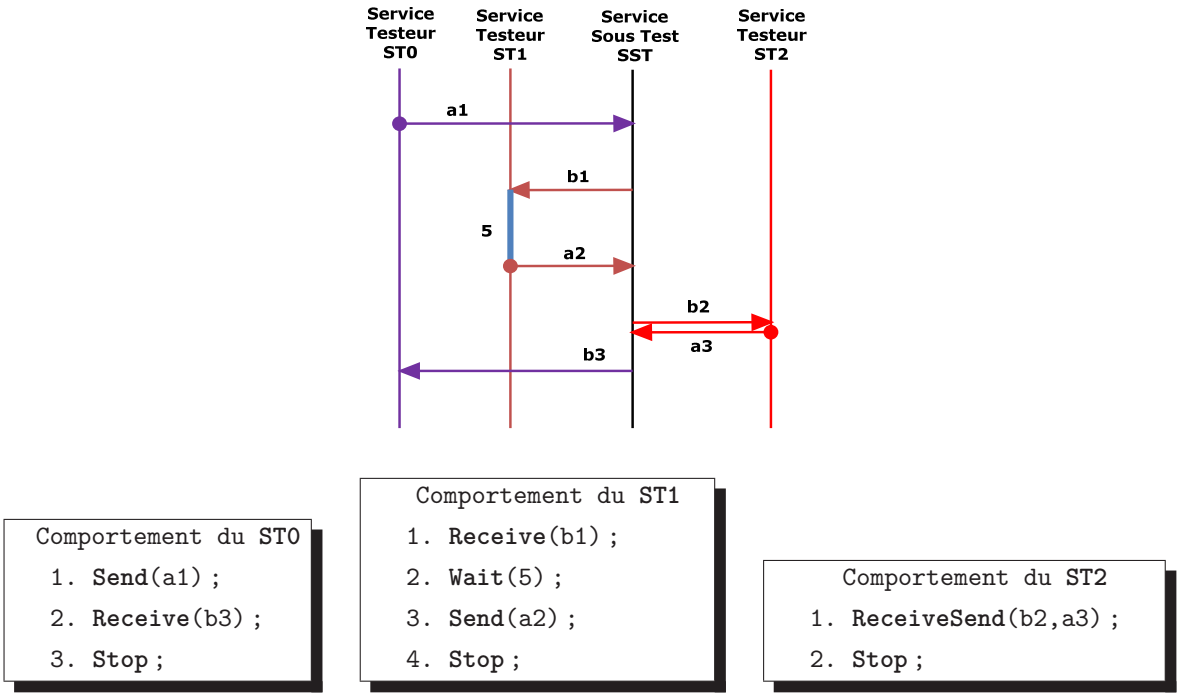

FIGURE 5.15 – Concrétisation de seq_1 pour une architecture ATD

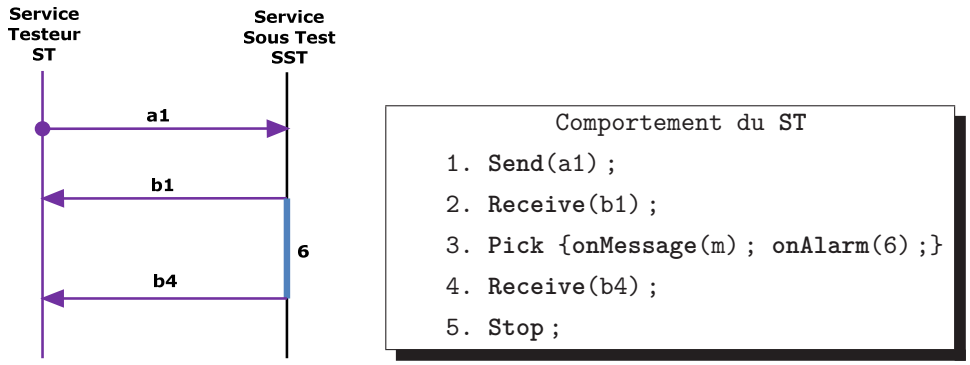

FIGURE 5.16 – Concrétisation de seq_2 pour une architecture ATC

Nous venons de décrire la concrétisation des tests abstraits qui constitue la troisième phase de notre approche de test. La dernière phase d'exécution des tests sera abordée dans le chapitre 6 après la mise en œuvre de notre plateforme de test et la prise en compte de l'architecture distribuée proposée dans ce chapitre.

5.7 Synthèse

Nous venons de présenter notre méthode de test de la composition de services Web. Dans notre approche de test, la description BPEL de cette composition est considérée comme une spécification de référence. Cette approche s'articule sur le test de conformité — d'un service composé par rapport à sa spécification de référence — avec une approche de boîte grise étant donné que les interactions entre le service composé et ses partenaires sont connues.

Notre approche est composée de quatre phases : modélisation formelle de la composition de services, génération automatique des tests temporisés abstraits, concrétisation des tests abstraits par rapport à une architecture de test, exécution des tests et émission de verdicts. La génération automatique de tests est guidée par un ensemble objectifs de test et se base sur une stratégie d'exploration partielle de l'espace d'états. Quant à la concrétisation des tests abstraits, elle consiste à engendrer des tests temporisés exécutables définissant les comportements des testeurs.

Dans le chapitre suivant, nous présenterons notre plateforme de test de l'orchestration de services, y compris nos outils développés : BPEL2IF et TESTGEN-IF, permettant respectivement la transformation de la description d'un processus BPEL en une spécification formelle en IF, et la génération de cas de test temporisés à partir de cette spécification et d'un ensemble d'objectifs de test. Pour illustrer notre méthode de test et l'utilisation de la plateforme de test, nous présenterons un cas d'étude d'un service Web composé. Nous décrirons les différentes phases de test partant d'une spécification de référence de ce service composé (processus BPEL) jusqu'à l'exécution des tests.

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).

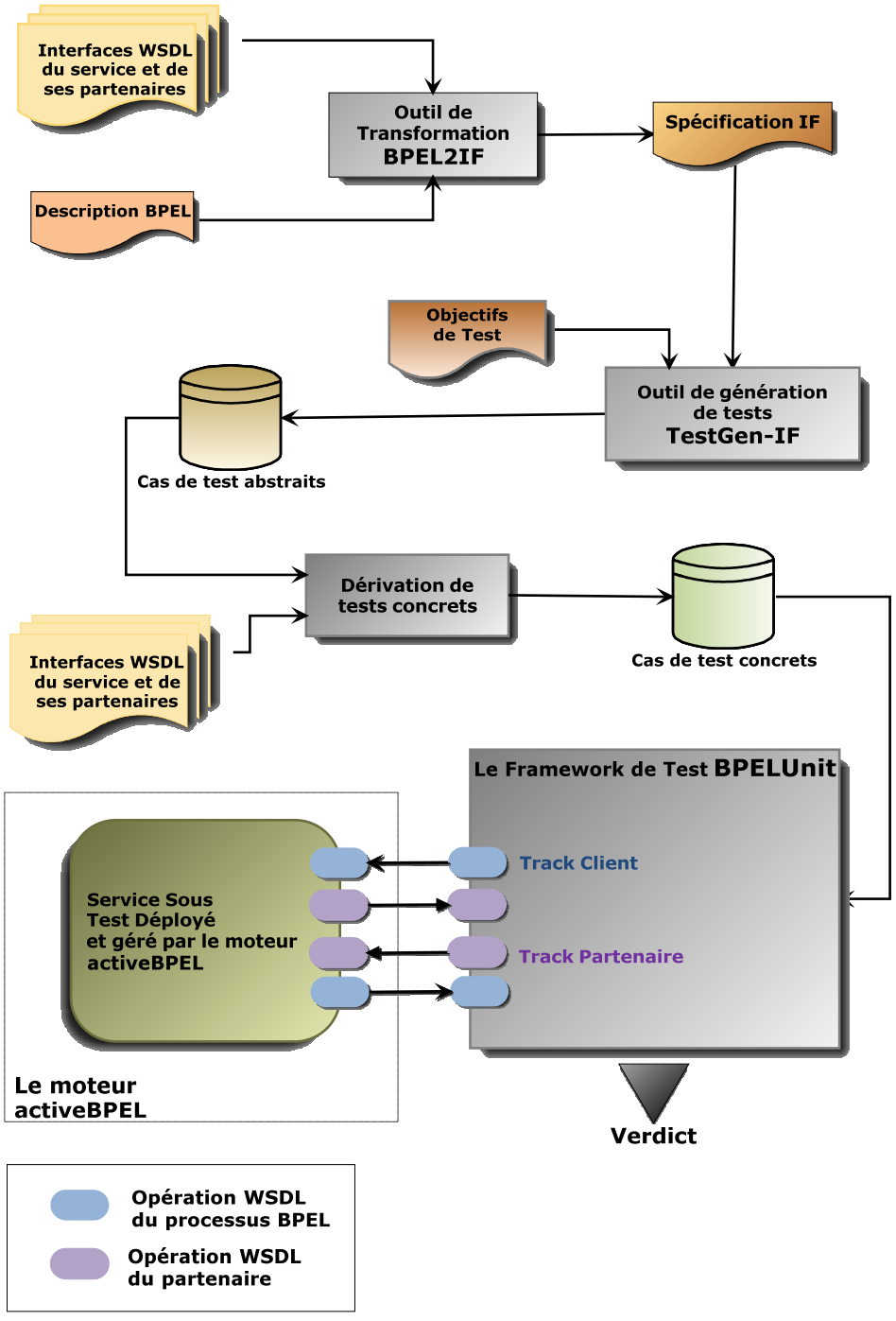

FIGURE 6.1 – Plateforme de test de l'orchestration de services Web

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.

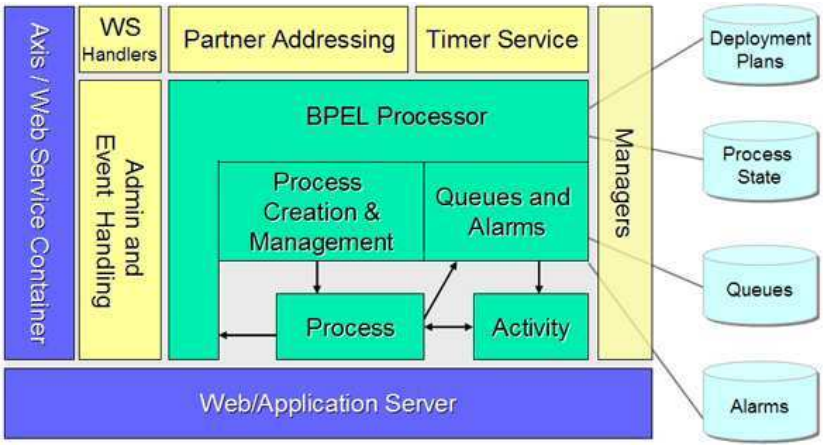

FIGURE 6.2 – Architecture du moteur activeBPEL

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.

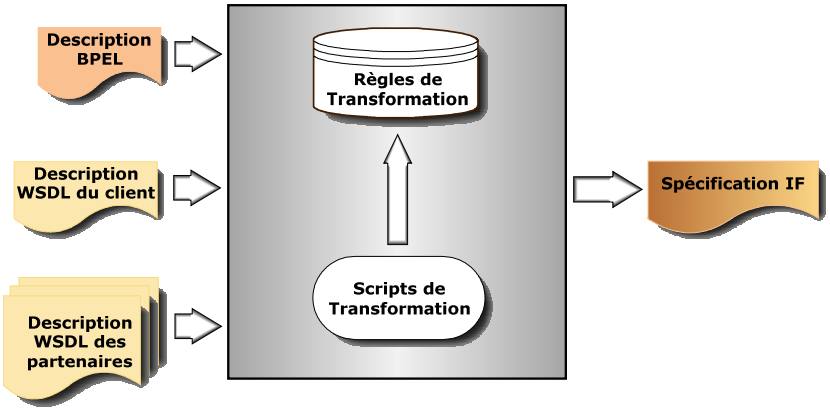

FIGURE 6.7 – Architecture de l'outil BPEL2IF

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).

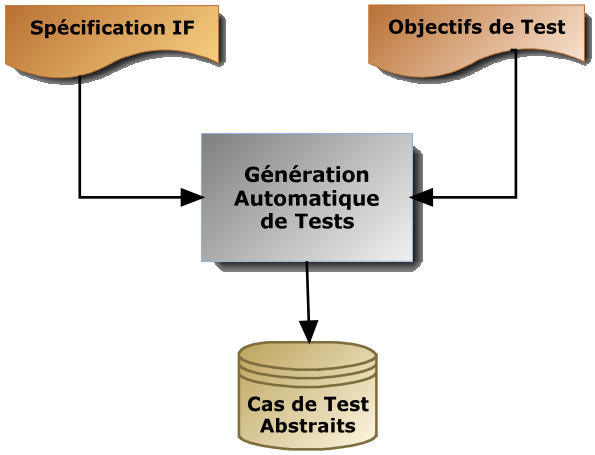

FIGURE 6.8 – Architecture de l'outil TestGen-IF

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.

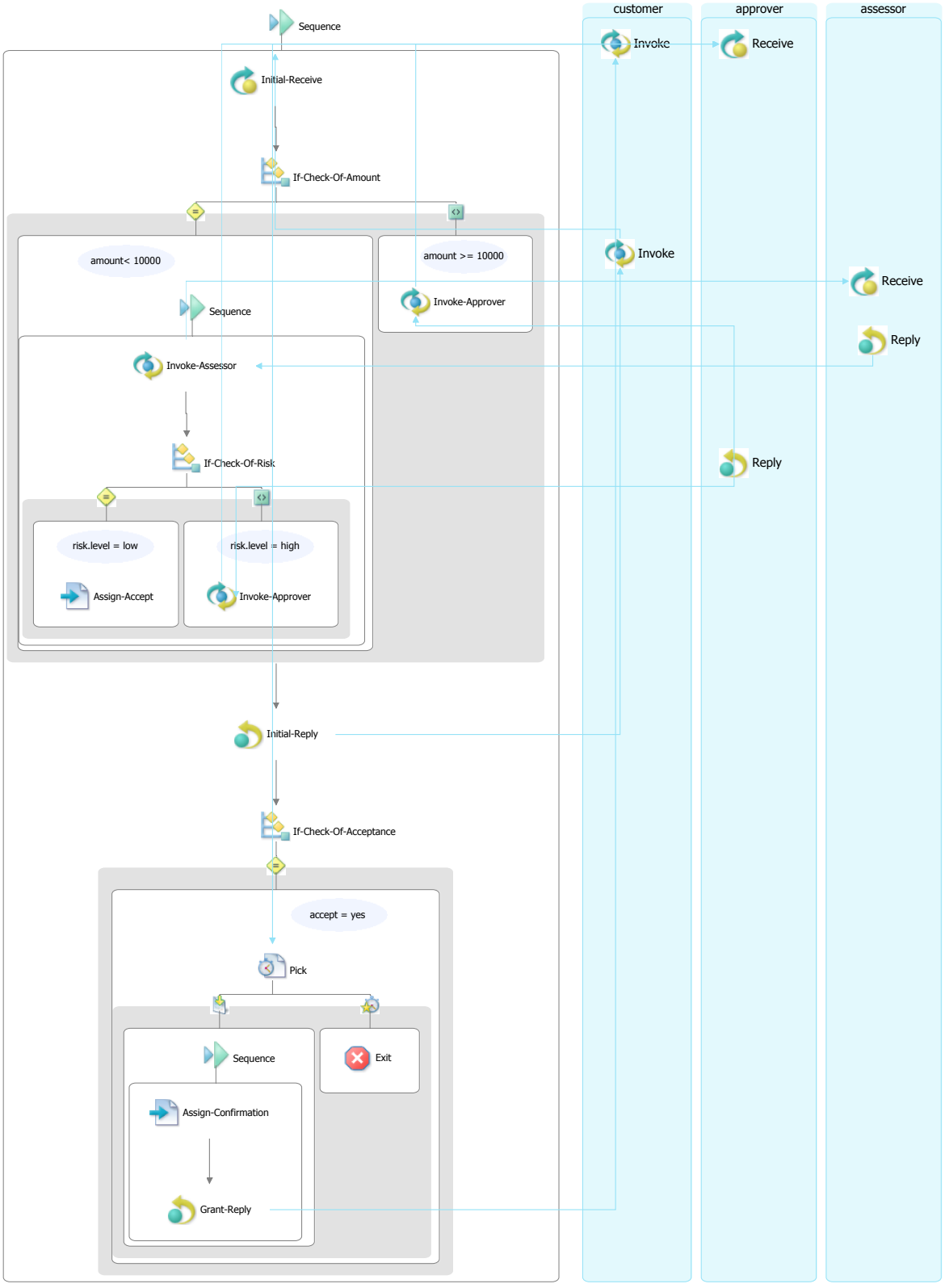

FIGURE 6.9 – flot du service `loanApproval` — variante séquentielle

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.

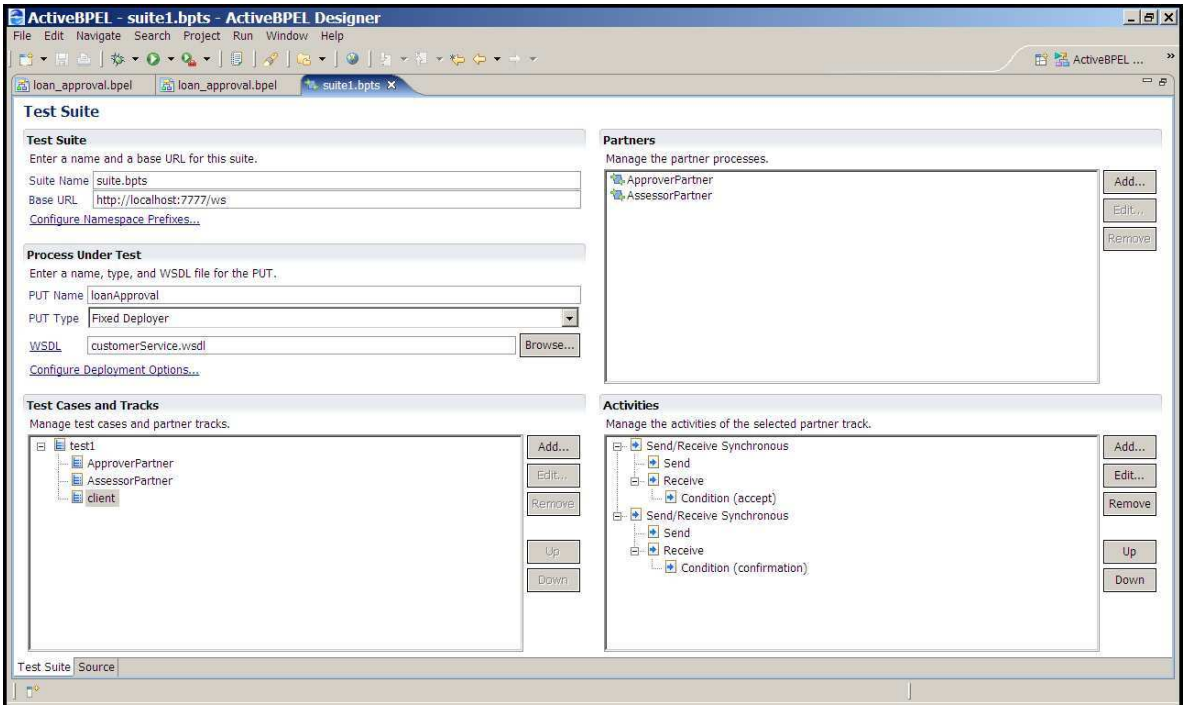

FIGURE 6.17 – L'interface de BPELUnit sous activeBPEL Designer

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.

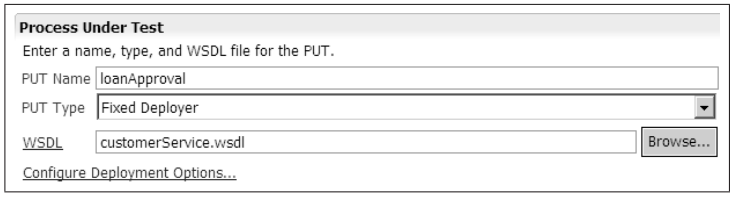

FIGURE 6.18 – Édition d'une suite de test dans BPELUnit — Service sous test

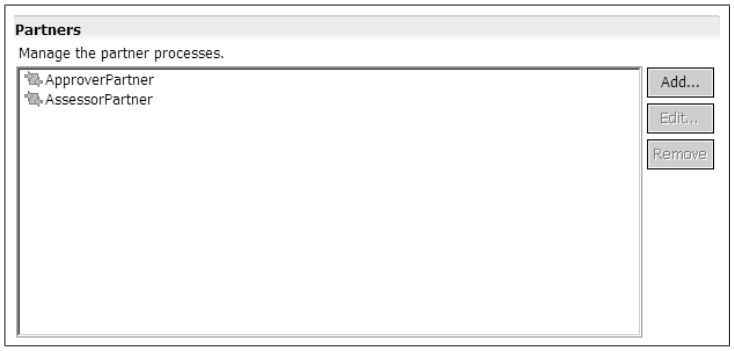

FIGURE 6.19 – Édition d'une suite de test dans BPELUnit — Sélection des partenaires

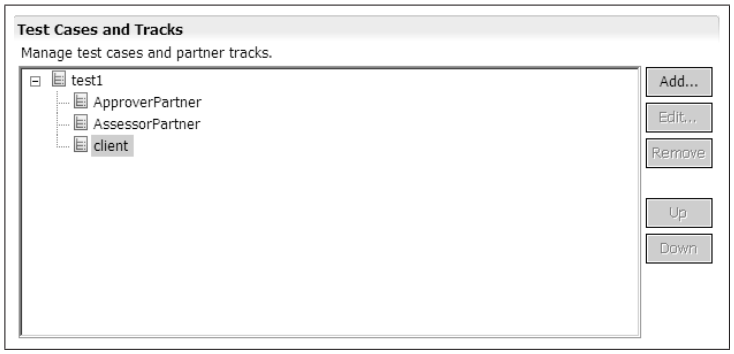

FIGURE 6.20 – Édition d'une suite de test dans BPELUnit — Édition de cas de test

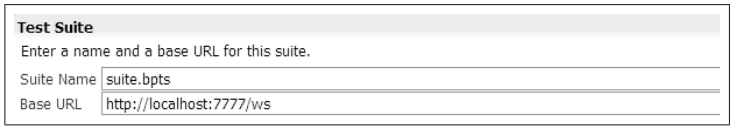

FIGURE 6.21 – Édition d'une suite de test dans BPELUnit — Caractéristiques d'une suite de test

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`.

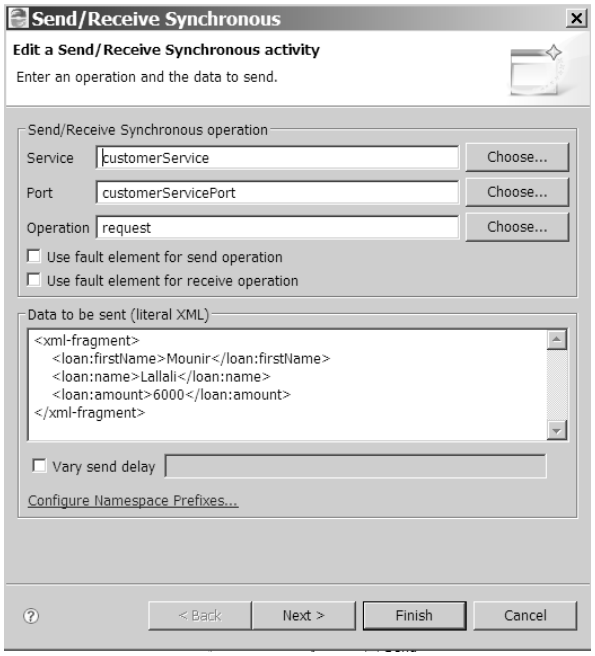

FIGURE 6.22 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Données XML envoyées par `Send`

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

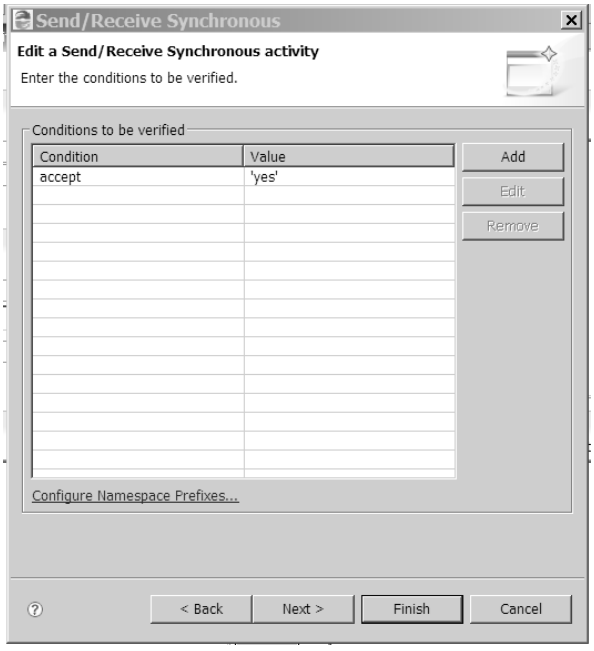

FIGURE 6.23 – Édition d'une activité `Send/Receive` synchrone dans BPELUnit — Condition de vérification des données reçues par `Receive`

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.

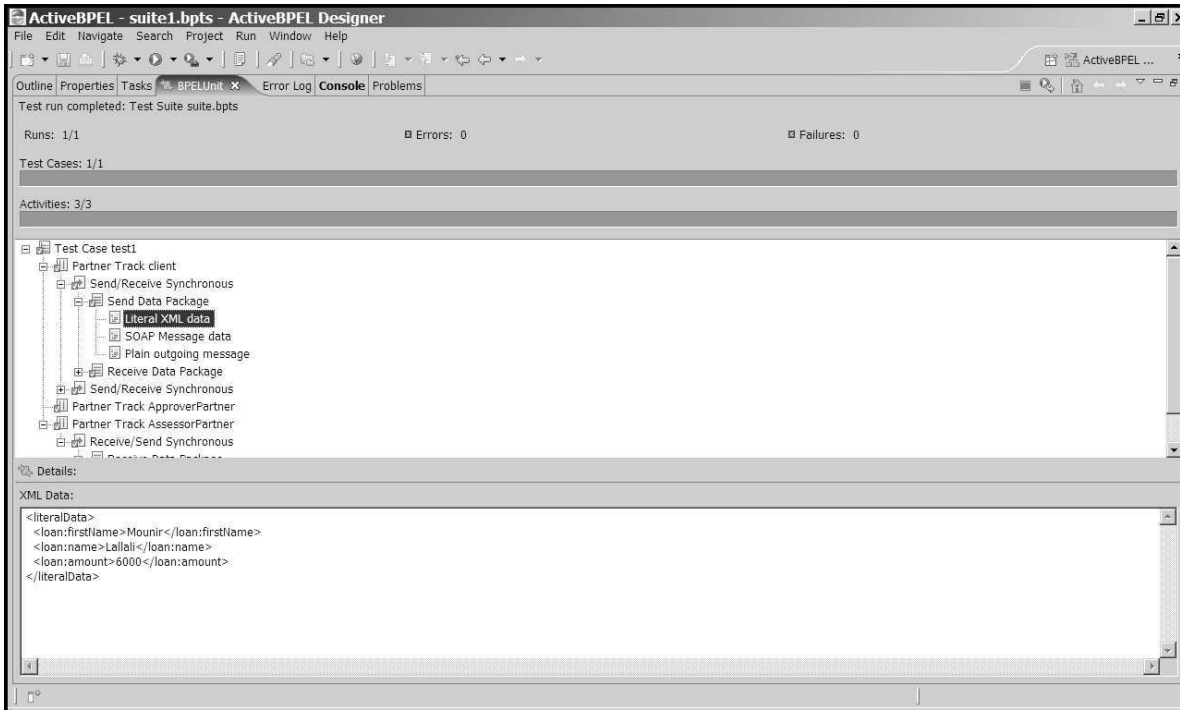

FIGURE 6.24 – Exécution d'une suite de test avec BPELUnit

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.

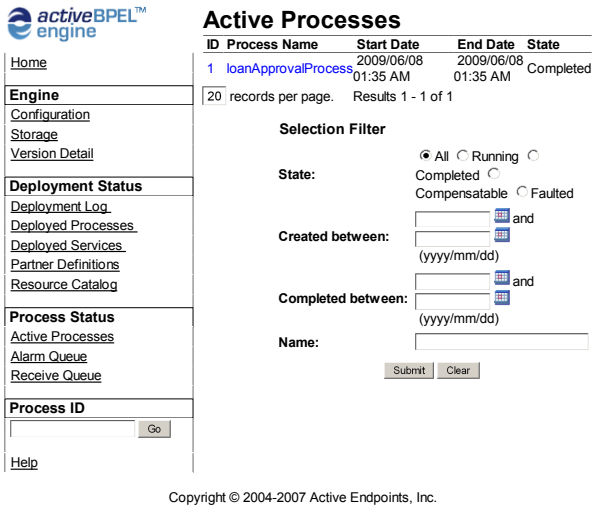

FIGURE 6.25 – Page d'administration de activeBPEL — Gestion des processus actifs

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.

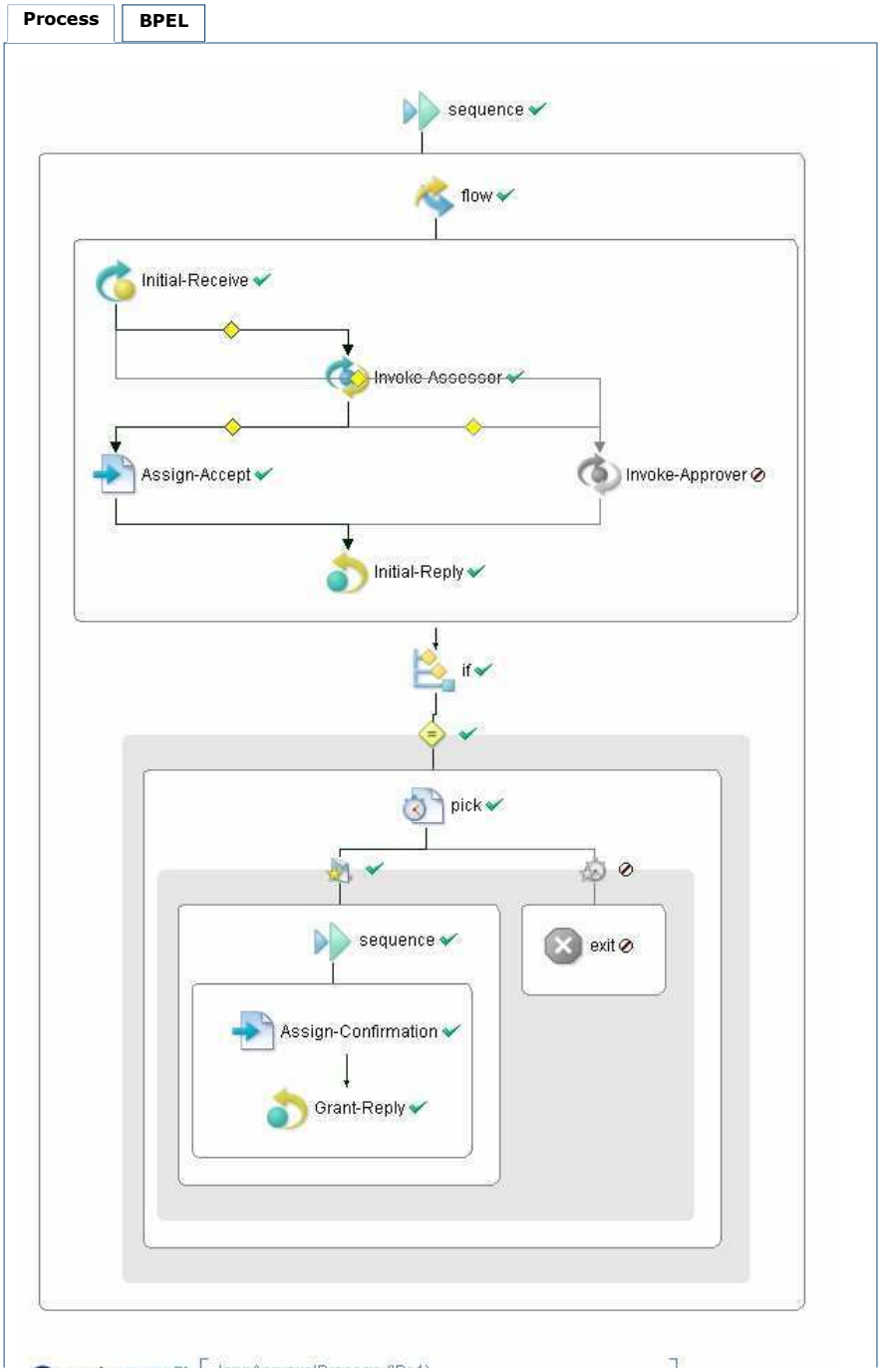

FIGURE 6.26 – Page d'administration de activeBPEL — Affichage de l'exécution du `loanApproval` sous test suite à la l'exécution du cas de test du scénario 1

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.

Quatrième partie

Conclusion et annexes

Chapitre 7

Conclusion et perspectives

Sommaire

CEtte thèse m'a permis d'explorer le domaine des services Web et de leur composition, et d'appliquer les techniques de test et les méthodes formelles à l'orchestration de services. L'objectif de cette thèse était double : d'une part, de définir une modélisation formelle (temporelle) de la composition de services, et d'autre part, de proposer une approche de test fonctionnel des services composés. Dans la suite, nous résumerons les travaux effectués et les résultats obtenus, et donnerons quelques directions envisageables pour la poursuite de nos travaux.

7.1 Synthèse des travaux et résultats

Définition d'un modèle temporisé WS-TEFSM Nous avons étendu le modèle d'automate temporisé [146] pour les besoins de la modélisation de la composition de services Web. Ainsi, nous avons défini un modèle de machine étendue à états finis temporisée, appelée WS-TEFSM, qui est enrichie par un ensemble d'horloges et de priorités associées aux transitions. Chaque état de cette machine est associé à un ensemble d'invariants permettant de contrôler l'écoulement du temps. Les priorités de transitions ont été introduites afin de pouvoir décrire la gestion des exceptions et la terminaison (forcée) des activités d'une orchestration de services. Nous avons défini formellement ce modèle, sa sémantique ainsi que ses fondements mathématiques.

Modélisation temporelle de la composition de services BPEL est un langage standard d'orchestration de services Web et d'exécution des processus métier. C'est un langage exécutable basé sur XML. De plus, sa sémantique est imprécise et informelle. Pour cela, nous avons proposé, dans le Chapitre 3, une modélisation formelle des descriptions BPEL en termes de machine WS-TEFSM

pour pouvoir appliquer notre approche formelle de test des services composés. Cette modélisation est assez complète étant donné qu'elle prend en compte la majorité des concepts de BPEL tels que la corrélation des messages, la gestion des exceptions, le traitement et l'échange des données complexes, la terminaison des activités ainsi que les activités temporelles de BPEL. Cette modélisation a servi dans la définition de notre méthode de transformation d'une description BPEL en une spécification formelle en IF — à base d'automates temporisés communicants.

Proposition d'une approche de test de la composition de services Dans le Chapitre 5, nous avons proposé une approche de test de la composition de services. Ce test de la composition est un test de conformité (fonctionnel) avec une approche de type boîte grise puisque l'on connaît les interactions entre le service composé et ses différents partenaires. Notre approche consiste à tester si l'implantation d'un service composé est conforme à sa spécification de référence, i.e. la description BPEL d'une composition. Elle est composée de quatre phases : (i) modélisation temporelle de la composition, génération des tests abstraits (temporisés), (iii) concrétisation des tests abstraits, (iv) exécution des tests et émission de verdicts. Pour ce test de conformité, nous avons utilisé une relation d'inclusion des traces temporisées que nous avons étendue et adaptée à notre approche de test boîte grise. Ainsi, les séquences de test ont été enrichies par des actions d'activation/désactivation d'horloges pour faciliter la concrétisation des tests, en particulier des actions d'attente. Nous avons aussi décrit une méthode de concrétisation des tests qui consiste à dériver des tests temporisés concrets comportant les données du test, les actions des services testeurs et les temps d'attente.

Génération automatique de tests temporisés Nous avons proposé aussi, dans le Chapitre 5, une méthode de génération automatique de tests temporisés qui est guidée par un ensemble d'objectifs de test décrivant les exigences fonctionnelles du test de conformité. Cette méthode se base sur la stratégie d'exploration partielle de l'espace d'états Hit-Or-Jump [6] que nous avons adapté aux automates temporisés communicants du langage IF. Nous avons implanté cette méthode dans l'outil TestGen-IF qui à partir d'une spécification formelle IF de la composition de services et d'un ensemble d'objectifs construit à la volée un cas de test temporisé abstrait (i.e. séquence d'actions observables et d'intervalles de temps).

Mise en œuvre de l'approche de test Nous avons mis en œuvre une plateforme de test de services composés permettant la transformation de BPEL en IF, la génération des tests, l'édition des tests concrets et leur exécution. Cette plateforme intègre nos deux outils développés : BPEL2IF l'outil de transformation de BPEL en IF, et TestGen-IF l'outil de génération de tests ainsi que le Framework d'édition et d'exécution de tests BPELUnit. Dans cette plateforme, nous considérons une architecture de test distribuée puisque chaque service partenaire peut être simulé par un service testeur (i.e. Track du Framewok BPELUnit).

En résumé, notre approche de test proposée constitue une méthode de test complète, allant de la modélisation formelle de la composition à l'exécution des tests, incluant la génération automatique des tests et leur concrétisation. Le modèle WS-TEFSM proposé est bien adapté à la modélisation de la composition de services. Notre approche est utilisable et répond à notre objectif principal qui est le test fonctionnel des services composés. Cette approche peut être complémentaire à d'autres approches de test telles que le test structurel ou le test symbolique de la composition de services.

7.2 Perspectives

Dans cette section, nous allons présenter quelques perspectives et pistes qui pourraient améliorer et/ou compléter notre approche, et plus généralement, le test de la composition de services.

7.2.1 Perspectives relatives à notre approche de test

Automatisation de la concrétisation des tests abstraits Nous comptons automatiser la concrétisation des tests en développant un outil dédié à la dérivation des tests pour le Framework BPELUnit. Cette automatisation sera divisée en trois parties :

- Génération automatique des données du test pour BPELUnit sous forme d'un arbre XML en combinant les données contenues dans les tests abstraits et la structure des données décrites par des schémas XML dans les descriptions WSDL ;
- Calcul des délais d'attente à partir des actions observables et des intervalles de temps ;
- Dérivation automatique des actions des testeurs par la transformation des actions observables du test abstrait en actions de BPELUnit.

Amélioration de l'implantation des outils développés Concernant l'outil de transformation BPEL2IF, il est possible d'effectuer la transformation de BPEL en IF avec des techniques de transformation de modèles MDA. Dans la version actuelle, cette transformation est réalisée à l'aide des règles écrites en langage XSL/XSLT. L'outil de génération TestGen-IF pourra être étendu par un éditeur intégré d'objectifs de test et l'implantation de nouvelles stratégies d'exploration (e.g. DFS, BDFS). En plus, il serait utile, et même indispensable, d'intégrer ces deux outils ainsi que le prochain outil de concrétisation des tests dans un environnement de développement intégré tel que Eclipse [1] ou NetBeans [2].

Test des propriétés non fonctionnelles des services Web composés Dans notre travail, nous avons considéré le test fonctionnel de la composition de services. En d'autres termes, nous avons défini une approche de test qui consiste à vérifier la conformité de l'implantation d'un service composé à sa spécification par rapport aux exigences fonctionnelles et/ou temporelles — décrites par des objectifs de test. Il est peut être envisageable d'étendre notre approche aux propriétés *non fonctionnelles* telles que les propriétés de sûreté ou de sécurité. Le test de telles propriétés, pourra assurer, par exemple, le bon comportement des services partenaires participant dans une composition. Pour cela, nous pouvons nous inspirer des méthodes et des techniques de test des propriétés non fonctionnelles qui sont appliquées à d'autres domaines que les services Web pour pouvoir adapter (éventuellement) notre approche au test non fonctionnel des services Web composés. Il faudrait pour cela revoir la définition des objectifs de test.

1. `http://www.eclipse.org/`
2. http ://www.netbeans.org/

Adaptation de notre approche au test de la chorégraphie de services La chorégraphie est un type de composition qui décrit une collaboration entre services pour accomplir un certain objectif. C'est une coordination décentralisée où chaque participant est responsable d'une partie du workflow, et connaît sa propre part dans le flux de messages échangés. WS-CDL [13] est l'un des langages de description de chorégraphie de service le plus utilisé. Nous pouvons explorer la possibilité d'adapter notre approche de test au test de la chorégraphie. Il serait intéressant d'inspecter les travaux de modélisation, de validation et de test des chorégraphies décrites en WS-CDL comme ceux de [186, 187, 188]. Étant donné que nous avons transformé une description BPEL en une spécification IF, nous pouvons procéder de la même manière pour transformer une descriptions WS-CDL en une spécification formelle IF et ensuite appliquer notre approche de test.

7.2.2 Perspectives générales

Test symbolique de la composition de services Notre approche de test repose sur le déploiement du modèle IF (i.e. système d'automates temporisés communicants) par énumération des états du système. Ce déploiement est réalisé à l'aide du simulateur IF qui effectue une exploration exhaustive de l'espace d'états du système. Nous pouvons donc être confronté au problème d'explosion du nombre d'états malgré l'utilisation de la stratégie d'exploration partielle Hit-Or-Jump [6] et l'abstraction des types de données utilisées dans les descriptions BPEL (remplacement des types non bornés par des énumérations ou intervalles). Pour pallier ce problème, nous pouvons envisager de procéder à une approche symbolique pour la formalisation de la sémantique de BPEL en termes de systèmes de transition symboliques (STS) [189]. Nous pouvons donc nous inspirer des travaux réalisés dans le cadre du test symbolique (e.g. [190, 191]) pour le test de la composition de services. Il serait intéressant de vérifier si on peut adapter notre méthode de génération de test, et plus précisément, la stratégie Hit-Or-Jump, aux systèmes STS. L'approche symbolique pourrait être composée des phases suivantes : (i) décrire, respectivement, la composition et les objectifs de test par un STS, (ii) construire le produit des deux STSs (composition et objectifs de test), (iii) générer un arbre d'exécution symbolique du STS composé, (iv) parcourir ce dernier à la volée en adaptant la stratégie Hit-Or-Jump pour dériver un cas de test symbolique, (v) concrétisation des tests avec des données effectives, (vi) déploiement et exécutions des tests. Cette approche symbolique aura un avantage considérable en évitant le problème d'explosion du nombre d'états et en réduisant l'espace d'états.

Amélioration des traitements des données complexes Dans notre approche de test, pour éviter le problème d'explosion du nombre d'états, nous avons effectué une abstraction de données en réduisant les types non bornés à des énumérations ou intervalles. En outre, nous avons parfois, simplifié les conditions utilisées dans les descriptions BPEL en les substituant par des variables booléennes. Afin de prendre en compte les valeurs concrètes ainsi que les données complexes (e.g. schémas XML) échangées ou traitées par une composition BPEL, dans le cas du test symbolique, nous pouvons décrire ces types complexes par des arbres XML, respectivement, les conditions de BPEL par des contraintes d'arbres. Ainsi, il faudrait avoir un solveur de contraintes capable de prendre en considération de telles contraintes. Nous pouvons décrire ces contraintes en langage OCL (Object Constraint Language), transformer les schémas XML en diagrammes de classes UML [80] que nous annotons avec ces contraintes OCL, et ensuite utiliser l'outil UML2CSP [192] ou USE [193] pour vérifier ces modèles de manière automatique et ainsi résoudre ces contraintes OCL.

Annexe A

Le service LoanApproval — variante séquentielle

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition Edited using ActiveBPEL(tm) Designer Version 3.0.0 (http://www.active-endpoints.com) -->
<bpel:process xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:lns="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="loanApprovalProcess" suppressJoinFailure="yes"
    targetNamespace="http://docs.active-endpoints.com/activebpel/sample/bpel/loan_approval/2006/09/loan_approval.bpel">
  <bpel:import importType="http://schemas.xmlsoap.org/wsdl/" location="loan_approval.wsdl"
      namespace="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"/>
  <bpel:partnerLinks>
    <bpel:partnerLink myRole="loanService" name="customer" partnerLinkType="lns:loanPartnerLinkType"/>
    <bpel:partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
    <bpel:partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor"/>
  </bpel:partnerLinks>
  <bpel:variables>
    <bpel:variable messageType="lns:creditInformationMessage" name="request"/>
    <bpel:variable messageType="lns:riskAssessmentMessage" name="risk"/>
    <bpel:variable messageType="lns:approvalMessage" name="approval"/>
    <bpel:variable messageType="lns:creditConfirmationMessage" name="acceptanceRequest"/>
    <bpel:variable messageType="lns:ConfirmationMessage" name="ConfirmationRequest"/>
  </bpel:variables>
  <bpel:correlationSets>
    <bpel:correlationSet name="loanIdentifier" properties="lns:clientLastName␣lns:clientFirstName"/>
  </bpel:correlationSets>
  <bpel:sequence>
    <bpel:flow>
      <bpel:links>
        <bpel:link name="receive-to-assess"/>
        <bpel:link name="receive-to-approval"/>
        <bpel:link name="assess-to-setMessage"/>
        <bpel:link name="assess-to-approval"/>
        <bpel:link name="setMessage-to-reply"/>
        <bpel:link name="approval-to-reply"/>
      </bpel:links>
      <bpel:receive createInstance="yes" name="Initial-Receive" operation="request" partnerLink="customer"
          portType="lns:loanServicePT" variable="request">
        <bpel:sources>
          <bpel:source linkName="receive-to-assess">
            <bpel:transitionCondition>($request.amount &lt; 10000)</bpel:transitionCondition>
          </bpel:source>
          <bpel:source linkName="receive-to-approval">
            <bpel:transitionCondition>($request.amount &gt;= 10000)</bpel:transitionCondition>
          </bpel:source>
        </bpel:sources>
```

```
39          < bpel:correlations >
40              < bpel:correlation    initiate ="yes"  set=" loanIdentifier "/>
41          </ bpel:correlations >
42        </ bpel:receive >
43        <bpel:invoke  inputVariable ="request"  name="Invoke-Assessor" operation="check"  outputVariable ="risk"
              partnerLink ="assessor"  portType="lns:riskAssessmentPT">
44          < bpel:targets >
45              < bpel:target  linkName="receive-to-assess"/>
46          </ bpel:targets >
47          <bpel:sources>
48              <bpel:source  linkName="assess-to-setMessage">
49                < bpel:transitionCondition >($risk.level = 'low')</ bpel:transitionCondition >
50              </ bpel:source>
51              <bpel:source  linkName="assess-to-approval">
52                < bpel:transitionCondition >($risk.level != 'low')</ bpel:transitionCondition >
53              </ bpel:source>
54          </ bpel:sources>
55        </ bpel:invoke>
56        <bpel:assign  name="Assign-Accept">
57          < bpel:targets >
58              < bpel:target  linkName="assess-to-setMessage"/>
59          </ bpel:targets >
60          <bpel:sources>
61              <bpel:source  linkName="setMessage-to-reply"/>
62          </ bpel:sources>
63          <bpel:copy>
64              <bpel:from>'yes'</bpel:from>
65              <bpel:to  part="accept"  variable ="approval"/>
66          </bpel:copy>
67        </ bpel:assign>
68        <bpel:invoke  inputVariable ="request"  name="Invoke-Approver" operation="approve" outputVariable="approval"
              partnerLink="approver"  portType="lns:loanApprovalPT">
69          < bpel:targets >
70              < bpel:target  linkName="receive-to-approval"/>
71              < bpel:target  linkName="assess-to-approval"/>
72          </ bpel:targets >
73          <bpel:sources>
74              <bpel:source  linkName="approval-to-reply"/>
75          </ bpel:sources>
76        </ bpel:invoke>
77        <bpel:reply  name="Initial-Reply" operation="request"  partnerLink="customer"  portType="lns:loanServicePT"
              variable ="approval">
78          < bpel:targets >
79              < bpel:target  linkName="setMessage-to-reply"/>
80              < bpel:target  linkName="approval-to-reply"/>
81          </ bpel:targets >
82        </ bpel:reply >
83      </ bpel:flow >
84      < bpel:if >
85        < bpel:condition  expressionLanguage=" urn:oasis:names:tc:wsbpel:2 .0 :sublang:xpath1 .0 ">$approval.accept =
              'yes'</bpel:condition>
86        <bpel:pick>
87          <bpel:onMessage operation="obtain"  partnerLink="customer"  portType="lns:loanServicePT"
                variable ="acceptanceRequest">
88            < bpel:correlations >
89              < bpel:correlation    initiate ="no"  set=" loanIdentifier "/>
90            </ bpel:correlations >
91            <bpel:sequence>
92              < bpel:assign  name="Assign-Confirmation">
93                <bpel:copy>
94                  <bpel:from>'Loan␣Confirmation'</bpel:from>
95                  <bpel:to  part="confirmation"  variable ="ConfirmationRequest"/>
96                </bpel:copy>
97              </ bpel:assign >
98              < bpel:reply  name="Grant-Reply" operation="obtain" partnerLink="customer"  portType="lns:loanServicePT"
                    variable ="ConfirmationRequest"/>
99            </bpel:sequence>
100         </bpel:onMessage>
```

```
101             <bpel:onAlarm>
102                 <bpel:for>"PT5S"</bpel:for>
103                 <bpel:exit />
104             </bpel:onAlarm>
105         </bpel:pick>
106       </bpel:if>
107     </bpel:sequence>
108 </bpel:process>
```

Annexe B

Le service LoanApproval — variante parallèle

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition Edited using ActiveBPEL(tm) Designer Version 3.0.0 ( http: // www.active-endpoints.com) -->
<bpel:process xmlns:bpel=" http: // docs. oasis -open.org/wsbpel /2.0/ process / executable "
    xmlns:bpws="http: // schemas.xmlsoap.org/ws/2003/03/ business -process/"
    xmlns:lns=" http: // docs. active -endpoints.com/ activebpel / sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema" name="loanApprovalProcess" suppressJoinFailure="yes"
    targetNamespace="http://docs.active-endpoints.com/ activebpel / sample/bpel / loan_approval /2006/09/ loan_approval . bpel">
  <bpel:import importType="http: // schemas.xmlsoap.org/wsdl/" location ="loan_approval . wsdl"
      namespace="http: // docs. active -endpoints.com/ activebpel / sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"/>
  < bpel:partnerLinks >
    <bpel:partnerLink myRole="loanService" name="customer" partnerLinkType="lns:loanPartnerLinkType"/>
    <bpel:partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver"/>
    <bpel:partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType" partnerRole=" assessor "/>
  </ bpel:partnerLinks >
  < bpel:variables >
    < bpel:variable messageType="lns:creditInformationMessage " name="request"/>
    < bpel:variable messageType="lns:riskAssessmentMessage" name="risk"/>
    < bpel:variable messageType="lns:approvalMessage" name="approval"/>
    < bpel:variable messageType="lns:creditConfirmationMessage" name="acceptanceRequest"/>
    < bpel:variable messageType="lns:ConfirmationMessage" name="ConfirmationRequest"/>
  </ bpel:variables >
  < bpel:correlationSets >
    < bpel:correlationSet name=" loanIdentifier " properties =" lns:clientLastName␣ lns:clientFirstName "/>
  </ bpel:correlationSets >
  <bpel:sequence>
    <bpel:flow>
      < bpel:links >
        < bpel:link name="receive-to-assess"/>
        < bpel:link name="receive-to-approval"/>
        < bpel:link name="assess-to-setMessage"/>
        < bpel:link name="assess-to-approval"/>
        < bpel:link name="setMessage-to-reply"/>
        < bpel:link name="approval-to-reply"/>
      </ bpel:links >
      < bpel:receive createInstance ="yes" name=" Initial -Receive" operation="request " partnerLink="customer"
          portType=" lns:loanServicePT " variable ="request ">
        <bpel:sources>
          <bpel:source linkName="receive-to-assess">
            < bpel:transitionCondition >($request.amount &lt; 10000)</bpel:transitionCondition>
          </bpel:source >
          <bpel:source linkName="receive-to-approval">
            < bpel:transitionCondition >($request.amount &gt;= 10000)</bpel:transitionCondition>
          </bpel:source >
        </bpel:sources >
```

```
39          < bpel:correlations >
40              < bpel:correlation    initiate ="yes"  set=" loanIdentifier "/>
41          </ bpel:correlations >
42        </ bpel:receive >
43        <bpel:invoke  inputVariable ="request"  name="Invoke-Assessor" operation="check"  outputVariable ="risk"
              partnerLink ="assessor"  portType="lns:riskAssessmentPT">
44          < bpel:targets >
45              < bpel:target  linkName="receive-to-assess"/>
46          </ bpel:targets >
47          <bpel:sources>
48              <bpel:source  linkName="assess-to-setMessage">
49                < bpel:transitionCondition >($risk.level = 'low')</ bpel:transitionCondition >
50              </ bpel:source>
51              <bpel:source  linkName="assess-to-approval">
52                < bpel:transitionCondition >($risk.level != 'low')</ bpel:transitionCondition >
53              </ bpel:source>
54          </ bpel:sources>
55        </ bpel:invoke>
56        <bpel:assign  name="Assign-Accept">
57          < bpel:targets >
58              < bpel:target  linkName="assess-to-setMessage"/>
59          </ bpel:targets >
60          <bpel:sources>
61              <bpel:source  linkName="setMessage-to-reply"/>
62          </ bpel:sources>
63          <bpel:copy>
64              <bpel:from>'yes'</bpel:from>
65              <bpel:to  part="accept"  variable ="approval"/>
66          </bpel:copy>
67        </ bpel:assign>
68        <bpel:invoke  inputVariable ="request"  name="Invoke-Approver" operation="approve" outputVariable="approval"
              partnerLink="approver"  portType="lns:loanApprovalPT">
69          < bpel:targets >
70              < bpel:target  linkName="receive-to-approval"/>
71              < bpel:target  linkName="assess-to-approval"/>
72          </ bpel:targets >
73          <bpel:sources>
74              <bpel:source  linkName="approval-to-reply"/>
75          </ bpel:sources>
76        </ bpel:invoke>
77        <bpel:reply  name="Initial-Reply" operation="request"  partnerLink="customer"  portType="lns:loanServicePT"
              variable ="approval">
78          < bpel:targets >
79              < bpel:target  linkName="setMessage-to-reply"/>
80              < bpel:target  linkName="approval-to-reply"/>
81          </ bpel:targets >
82        </ bpel:reply >
83      </ bpel:flow >
84      < bpel:if >
85        < bpel:condition  expressionLanguage=" urn:oasis:names:tc:wsbpel:2 .0 :sublang:xpath1 .0 ">$approval.accept =
              'yes'</bpel:condition>
86        <bpel:pick>
87          <bpel:onMessage operation="obtain"  partnerLink="customer"  portType="lns:loanServicePT"
                variable ="acceptanceRequest">
88            < bpel:correlations >
89              < bpel:correlation    initiate ="no"  set=" loanIdentifier "/>
90            </ bpel:correlations >
91            <bpel:sequence>
92              < bpel:assign  name="Assign-Confirmation">
93                <bpel:copy>
94                  <bpel:from>'Loan␣Confirmation'</bpel:from>
95                  <bpel:to  part="confirmation"  variable ="ConfirmationRequest"/>
96                </bpel:copy>
97              </ bpel:assign >
98              < bpel:reply  name="Grant-Reply" operation="obtain" partnerLink="customer"  portType="lns:loanServicePT"
                    variable ="ConfirmationRequest"/>
99            </bpel:sequence>
100         </bpel:onMessage>
```

```
101             <bpel:onAlarm>
102                 <bpel:for>"PT5S"</bpel:for>
103                 <bpel:exit />
104             </bpel:onAlarm>
105         </bpel:pick>
106       </bpel:if>
107     </bpel:sequence>
108 </bpel:process>
```

Annexe C

Flot de contrôle du service LoanApproval — variante parallèle

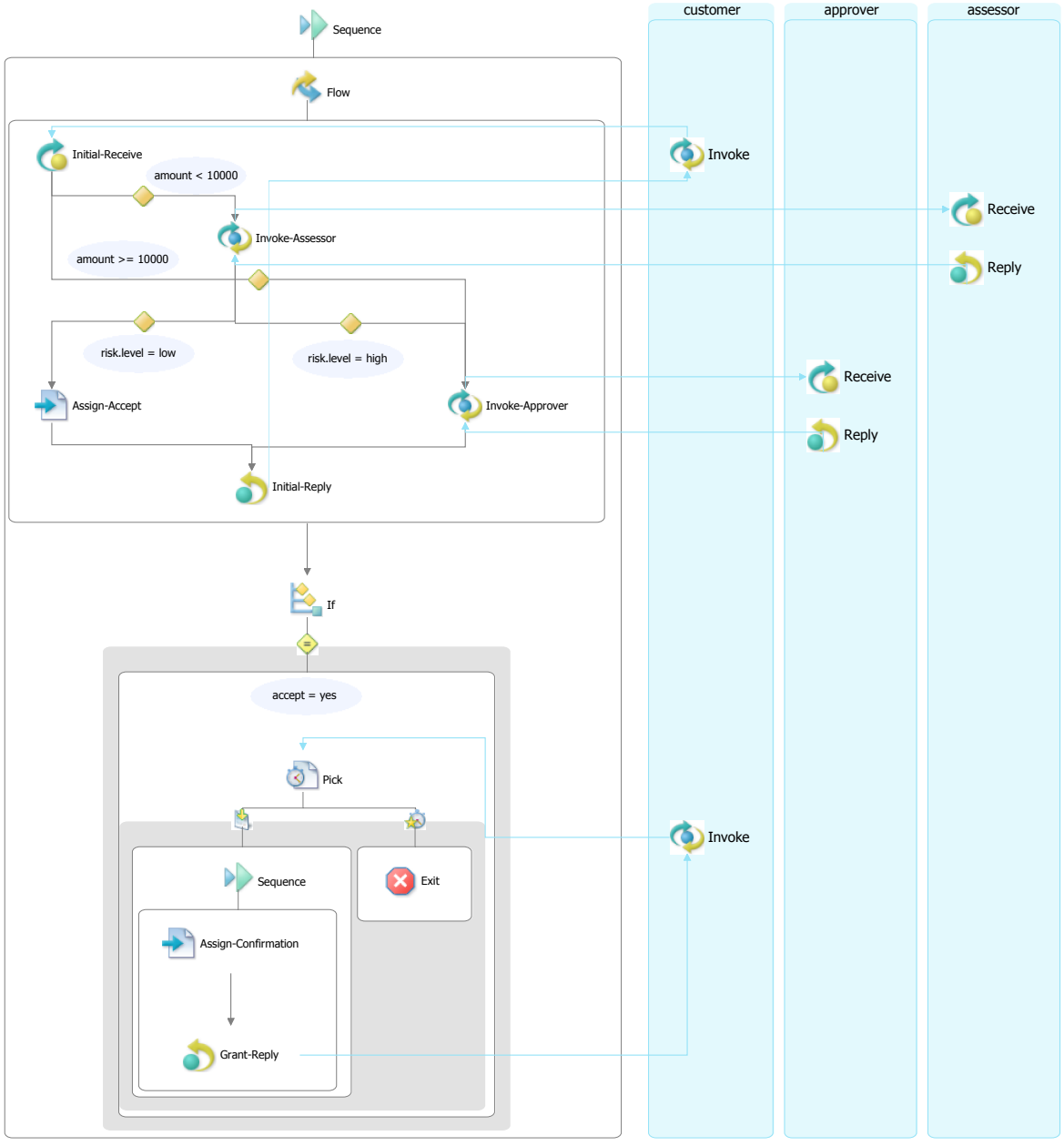

FIGURE C.1 – flot du service LoanApproval — variante parallèle

Annexe D

Descriptions WSDL du client et des partenaires du service `loanApproval`

D.1 Description WSDL du client de `loanApproval`

```
<?xml version="1.0" encoding="UTF-8" ?>
< definitions
     targetNamespace="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
    xmlns:vprop="http://docs.oasis-open.org/wsbpel/2.0/varprop"
    xmlns:tns="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl">

    <message name="creditInformationMessage">
        <part name="firstName" type="xsd:string" />
        <part name="name" type="xsd:string" />
        <part name="amount" type="xsd:integer" />
    </message>
    <message name="creditConfirmationMessage">
        <part name="firstName" type="xsd:string" />
        <part name="name" type="xsd:string" />
    </message>
    <message name="ConfirmationMessage">
        <part name="confirmation" type="xsd:string" />
    </message>
    <message name="approvalMessage">
        <part name="accept" type="xsd:string" />
    </message>
    <message name="riskAssessmentMessage">
        <part name="level" type="xsd:string" />
    </message>
    <message name="errorMessage">
        <part name="errorCode" type="xsd:integer" />
    </message>

    <portType name="loanServicePT">
        <operation name="request">
            <input message="tns:creditInformationMessage" />
            <output message="tns:approvalMessage" />
            <fault name="unableToHandleRequest"
                message="tns:errorMessage" />
        </operation>
```

```
40      <operation name="obtain">
41          <input message="tns:creditConfirmationMessage" />
42          <output message="tns:ConfirmationMessage" />
43      </operation>
44  </portType>
45
46  <portType name="riskAssessmentPT">
47      <operation name="check">
48          <input message="tns:creditInformationMessage" />
49          <output message="tns:riskAssessmentMessage" />
50          <fault name="loanProcessFault" message="tns:errorMessage" />
51      </operation>
52  </portType>
53
54  <portType name="loanApprovalPT">
55      <operation name="approve">
56          <input message="tns:creditInformationMessage" />
57          <output message="tns:approvalMessage" />
58          <fault name="loanProcessFault" message="tns:errorMessage" />
59      </operation>
60  </portType>
61
62  <plnk:partnerLinkType name="loanPartnerLinkType">
63      <plnk:role name="loanService">
64          <plnk:portType name="tns:loanServicePT" />
65      </plnk:role>
66  </plnk:partnerLinkType>
67  <plnk:partnerLinkType name="loanApprovalLinkType">
68      <plnk:role name="approver">
69          <plnk:portType name="tns:loanApprovalPT" />
70      </plnk:role>
71  </plnk:partnerLinkType>
72  <plnk:partnerLinkType name="riskAssessmentLinkType">
73      <plnk:role name="assessor">
74          <plnk:portType name="tns:riskAssessmentPT" />
75      </plnk:role>
76  </plnk:partnerLinkType>
77
78  <binding name="loanServiceBinding" type="tns:loanServicePT">
79      <soap:binding style="document"
80          transport="http://schemas.xmlsoap.org/soap/http" />
81      <operation name="request">
82          <soap:operation soapAction="" style="document" />
83          <input>
84              <soap:body use="literal"
85                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
86          </input>
87          <output>
88              <soap:body use="literal"
89                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
90          </output>
91      </operation>
92      <operation name="obtain">
93          <soap:operation soapAction="" style="document" />
94          <input>
95              <soap:body use="literal"
96                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
97          </input>
98          <output>
99              <soap:body use="literal"
100                 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
101         </output>
102     </operation>
103 </binding>
104
105 <binding name="ApproverBinding" type="tns:loanApprovalPT">
106     <soap:binding style="document"
107         transport="http://schemas.xmlsoap.org/soap/http" />
```

```
108         <operation name="approve">
109             <soap:operation soapAction="" style="document" />
110             <input>
111                 <soap:body use="literal"
112                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
113             </input>
114             <output>
115                 <soap:body use="literal"
116                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
117             </output>
118         </operation>
119     </binding>
120
121     <binding name="AssessorBinding" type="tns:riskAssessmentPT">
122         <soap:binding style="document"
123             transport="http://schemas.xmlsoap.org/soap/http" />
124         <operation name="check">
125             <soap:operation soapAction="" style="document" />
126             <input>
127                 <soap:body use="literal"
128                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
129             </input>
130             <output>
131                 <soap:body use="literal"
132                     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" />
133             </output>
134         </operation>
135     </binding>
136
137     <service name="LoanApprovalService">
138         <documentation>Loan Approval Service</documentation>
139         <port name="loanServicePort" binding="tns:loanServiceBinding">
140             <soap:address
141                 location="http://localhost:8080/active-bpel/services/customerService" />
142         </port>
143     </service>
144
145     <service name="LoanAssessor">
146         <documentation>Loan Assessor Service</documentation>
147         <port name="AssessorPort" binding="tns:AssessorBinding">
148             <soap:address
149                 location="http://localhost:7777/ws/AssessorPartner" />
150         </port>
151     </service>
152     <service name="LoanApprover">
153         <documentation>Loan Approver Service</documentation>
154         <port name="ApproverPort" binding="tns:ApproverBinding">
155             <soap:address
156                 location="http://localhost:7777/ws/ApproverPartner" />
157         </port>
158     </service>
159
160     <vprop:property name="clientFirstName" type="xsd:string" />
161     <vprop:property name="clientLastName" type="xsd:string" />
162     <vprop:propertyAlias propertyName="tns:clientFirstName"
163         messageType="tns:creditInformationMessage" part="firstName"
164         query="/firstName" />
165     <vprop:propertyAlias propertyName="tns:clientLastName"
166         messageType="tns:creditInformationMessage" part="name" query="/name" />
167     <vprop:propertyAlias propertyName="tns:clientFirstName"
168         messageType="tns:creditConfirmationMessage" part="firstName"
169         query="/firstName" />
170     <vprop:propertyAlias propertyName="tns:clientLastName"
171         messageType="tns:creditConfirmationMessage" part="name" query="/name" />
172 </definitions>
```

D.2 Description WSDL du service partenaire `loanAssessor`

```
< definitions  targetNamespace="http: // tempuri . org / services / loanassessor "
    xmlns:tns=" http: // tempuri . org / services / loanassessor "
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema"
    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"
    xmlns:plnk=" http: // schemas.xmlsoap.org/ws/2003/05/ partner -link/"
    xmlns="http: // schemas.xmlsoap.org/wsdl/">

    <message name="creditInformationMessage">
        <part  name="firstName" type=" xsd:string "  />
        <part  name="name" type="xsd:string"  />
        <part  name="amount" type="xsd:integer"  />
    </message>
    <message name="loanRequestErrorMessage">
        <part  name="errorCode" type=" xsd:integer "  />
    </message>
    <message name="riskAssessmentMessage">
        <part  name="level"  type=" xsd:string "  />
    </message>

    <portType name="riskAssessmentPT">
        <operation  name="check">
            <input  message=" tns:creditInformationMessage "  />
            <output  message="tns:riskAssessmentMessage" />
            <fault  name="loanProcessFault"
                message="tns:loanRequestErrorMessage"  />
        </operation>
    </portType>

    <plnk:partnerLinkType  name="riskAssessmentLinkType">
        <plnk:role  name="assessor">
            <plnk:portType  name="tns:riskAssessmentPT" />
        </ plnk:role >
    </ plnk:partnerLinkType>

    <binding  name="SOAPBinding" type="tns:riskAssessmentPT">
        <soap:binding  style ="document"
             transport =" http: // schemas.xmlsoap.org/soap/ http "  />
        <operation  name="check">
            <soap:operation  soapAction=""  style ="document" />
            <input>
                <soap:body use="  literal  "
                    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </input>
            <output>
                <soap:body use="  literal  "
                    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </output>
        </operation>
    </binding>

    <service  name="LoanAssessor">
        <documentation>Loan Assessor Service</documentation>
        <port  name="SOAPPort" binding="tns:SOAPBinding">
            <soap:address
                 location =" http: //  localhost:7777 / ws/ AssessorPartner "  />
        </port>
    </ service >
</  definitions >
```

D.3 Description WSDL du service partenaire `loanApprover`

```
< definitions  targetNamespace="http: // tempuri . org / services / loanapprover"
    xmlns:tns=" http: // tempuri . org / services / loanapprover"
    xmlns:xsd=" http: // www.w3.org/2001/XMLSchema"
    xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"
    xmlns:plnk=" http: // schemas.xmlsoap.org/ws/2003/05/ partner -link/"
    xmlns="http: // schemas.xmlsoap.org/wsdl/">

    <message name="creditInformationMessage">
        <part  name="firstName" type=" xsd:string "  />
        <part  name="name" type="xsd:string"  />
        <part  name="amount" type="xsd:integer"  />
    </message>
    <message name="loanRequestErrorMessage">
        <part  name="errorCode" type=" xsd:integer "  />
    </message>
    <message name="approvalMessage">
        <part  name="accept" type=" xsd:string "  />
    </message>

    <portType name="loanApprovalPT">
        <operation  name="approve">
            <input  message=" tns:creditInformationMessage "  />
            <output  message="tns:approvalMessage" />
            <fault  name="loanProcessFault"
                message="tns:loanRequestErrorMessage" />
        </ operation >
    </portType>

    <plnk:partnerLinkType  name="loanApprovalLinkType">
        < plnk:role  name="approver">
            <plnk:portType  name="tns:loanApprovalPT" />
        </ plnk:role >
    </ plnk:partnerLinkType >

    <binding  name="SOAPBinding" type="tns:loanApprovalPT">
        <soap:binding  style ="document"
             transport =" http: // schemas.xmlsoap.org/soap/ http "  />
        <operation  name="approve">
            < soap:operation  soapAction=""  style ="document" />
            <input>
                <soap:body use="  literal  "  xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </ input >
            <output>
                <soap:body use="  literal  "  xmlns:soap=" http: // schemas.xmlsoap.org/wsdl/soap/"  />
            </ output >
        </ operation >
    </binding>

    <service  name="LoanApprover">
        <documentation>Loan Approver Service</documentation>
        <port  name="SOAPPort" binding="tns:SOAPBinding">
            <soap:address  location =" http: // localhost:7777 /ws/ApproverPartner"  />
        </ port >
    </ service >
</ definitions >
```

Annexe E

Cas de test concret du scénario 1

```
<?xml version="1.0" encoding="UTF-8"?>
<tes:testSuite xmlns:tes="http://www.bpelunit.org/schema/testSuite"
    xmlns:loan="http://docs.active-endpoints.com/activebpel/sample/wsdl/loan_approval/2006/09/loan_approval.wsdl"
    xmlns:loan1="http://tempuri.org/services/loanapprover" xmlns:loan2="http://tempuri.org/services/loanassessor">
  <tes:name>suite.bpts</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <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>
  <tes:testCases>
    <tes:testCase name="test1" 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:send fault="false" delaySequence="">
            <tes:data>
              <loan:firstName>Mounir</loan:firstName>
              <loan:name>Lallali</loan:name>
            </tes:data>
          </tes:send>
          <tes:receive fault="false">
            <tes:condition>
              <tes:expression>confirmation</tes:expression>
              <tes:value>'Loan Confirmation'</tes:value>
            </tes:condition>
          </tes:receive>
        </tes:sendReceive>
      </tes:clientTrack>
      <tes:partnerTrack name="AssessorPartner">
```

```
46                  <tes:receiveSend  service="loan2:LoanAssessor"  port="SOAPPort" operation="check">
47                      <tes:send  fault=" false "  delaySequence="">
48                          <tes:data>
49                              <loan2:level>low</loan2:level>
50                          </tes:data>
51                      </tes:send>
52                      <tes:receive  fault=" false "/>
53                  </tes:receiveSend>
54              </tes:partnerTrack>
55          </tes:testCase>
56      </tes:testCases>
57  </tes:testSuite>
```

Annexe F

Cas de test concret du scénario 8

```
<?xml version="1.0" encoding="UTF-8"?>
< tes:testSuite xmlns:tes=" http: // www.bpelunit.org/schema/ testSuite "
    xmlns:loan=" http: // docs. active -endpoints.com/ activebpel /sample/wsdl/ loan_approval /2006/09/ loan_approval . wsdl"
    xmlns:loan1=" http: // tempuri . org/ services / loanapprover " xmlns:loan2=" http: // tempuri . org/ services / loanassessor ">
  <tes:name>suite.bpts</tes:name>
  <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
  <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>
  < tes:testCases >
    < tes:testCase name="test1" 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 >//loan1:accept</tes:expression>
              < tes:value >'yes'</ tes:value >
            </ tes:condition >
          </ tes:receive >
        </ tes:sendReceive >
        <tes:sendReceive service=" loan:customerService " port="customerServicePort" operation="obtain ">
          <tes:send fault=" false ">
            < tes:data >
              <loan:firstName>Fatiha</loan:firstName>
              <loan:name>ZAIDI</loan:name>
            </ tes:data >
          </ tes:send >
          < tes:receive fault=" false ">
            < tes:condition >
              < tes:expression >confirmation</tes:expression>
              < tes:value >'Loan␣Confirmation'</ tes:value >
            </ tes:condition >
          </ tes:receive >
        </ tes:sendReceive >
      </ tes:clientTrack >
      < tes:partnerTrack name="ApproverPartner">
```

```
46              <tes:receiveSend service="loan1:LoanApprover" port="SOAPPort" operation="approve">
47                  <tes:send fault="false" delaySequence="">
48                      <tes:data>
49                          <loan1:accept>yes</loan1:accept>
50                      </tes:data>
51                  </tes:send>
52                  <tes:receive fault="false"/>
53              </tes:receiveSend>
54          </tes:partnerTrack>
55          <tes:partnerTrack name="AssessorPartner">
56              <tes:receiveSend service="loan2:LoanAssessor" port="SOAPPort" operation="check">
57                  <tes:send fault="false" delaySequence="">
58                      <tes:data>
59                          <loan2:level>high</loan2:level>
60                      </tes:data>
61                  </tes:send>
62                  <tes:receive fault="false"/>
63              </tes:receiveSend>
64          </tes:partnerTrack>
65      </tes:testCase>
66      <tes:testCase name="test2" basedOn="" abstract="false" vary="false">
67          <tes:clientTrack>
68              <tes:sendReceive service="loan:customerService" port="customerServicePort" operation="obtain">
69                  <tes:send fault="false" delaySequence="2">
70                      <tes:data>
71                          <loan:firstName>Mounir</loan:firstName>
72                          <loan:name>Lallali</loan:name>
73                      </tes:data>
74                  </tes:send>
75                  <tes:receive fault="false">
76                      <tes:condition>
77                          <tes:expression>confirmation</tes:expression>
78                          <tes:value>'Loan␣Confirmation'</tes:value>
79                      </tes:condition>
80                  </tes:receive>
81              </tes:sendReceive>
82          </tes:clientTrack>
83      </tes:testCase>
84  </tes:testCases>
85  </tes:testSuite>
```

Publications personnelles

[150] Ana Cavalli, Mounir Lallali, Stephane Maag, Gerardo Morales, and Fatiha Zaidi. *in Emergent Web Intelligence, Studies in Computational Intelligence*, chapter Modeling and testing of Web based systems. Springer Verlag, 2009. 42 pages.

[151] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Timed modeling of web services composition for automatic testing. In *Proc. Third International IEEE Conference on Signal-Image Technologies and Internet-Based System SITIS '07*, pages 417–426, 16–18 Dec. 2007.

[152] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Definition of the Mapping from BPEL to WS-TEFSM. *Livrable WEBMOV-FC-D2.3/T2.4*, Novembre 2008. 40 pages.

[162] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic timed test case generation for web services composition. In *Proc. IEEE Sixth European Conference on Web Services ECOWS '08*, pages 53–62, 12–14 Nov. 2008.

[163] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Transforming BPEL into intermediate format language for web services composition testing. In *Proc. 4th International Conference on Next Generation Web Services Practices NWESP '08*, pages 191–197, 20–22 Oct. 2008.

[164] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Automation of the Mapping from BPEL to WS-TEFSM-IF. *Livrable WEBMOV-FC-D2.4a/T2.5*, Mars 2009. 32 pages.

[172] Richard Castanet Patrick Felix Mounir Lallali Fatiha Zaidi Ismail Berrada, Tien-Dung Cao and Ana Cavalli. Définition d'une méthode de génération de test pour la composition de services Web décrite en BPEL. *Livrable WEBMOV-FC-D4.1/T4.1*, Juin 2009. 44 pages.

[181] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *Proc. 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications DS-RT 2008*, pages 315–318, 27–29 Oct. 2008.

[182] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana Rosa Cavalli. Modeling and testing secure web-based systems : Application to an industrial case study. In *Proc. IEEE International Conference on Signal Image Technology and Internet Based Systems SITIS '08*, pages 128–136, Nov. 30 2008–Dec. 3 2008.

[183] Iksoon Hwang, Mounir Lallali, Ana Cavalli, and Dominique Verchere. Modeling, validation, and test generation of pcep using formal method. In *The IFIP International Conference on Formal Techniques ofr Distributed Systems FMOODS/FORTE'09*, 9-11 June 2009. 15 pages.

Bibliographie

[1] OASIS Standard. WSBPEL Ver. 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[2] Thomas Erl. *Service-oriented Architecture : Concepts, Technology, and Design*. Prentice Hall, 2005.

[3] OASIS. Organization for the Advancement of Structured Information Standards OASIS. http://www.oasis-open.org.

[4] M.-C. Gaudel. Validation et Vérification. In *Encyclopédie de l'informatique et des systèmes d'information*, pages 1136–1150. Vuibert, 2006.

[5] Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language wsdl (version 1.1), march 2001. http://www.w3.org/TR/wsdl.

[6] A. R. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump : An algorithm for embedded testing with applications to IN services. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 41–56, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[7] Thomas Erl. *Service-Oriented Architecture : A Field Guide to Integrating Xml and Web Services*. Prentice Hall, 2004.

[8] W3C. eXtensible Markup Language XML, 2008. http://www.w3.org/XML.

[9] W3C. XML Schema. http://www.w3.org/XML/Schema.

[10] W3C. HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/.

[11] W3C. Simple Object Access Protocol SOAP (Version 1.1), May 2000. http://www.w3.org/TR/soap/.

[12] UDDI. Universal Description, Discovery and Integration UDDI. http://www.uddi.org/.

[13] W3C. Web services choreography description language version ws-cdl 1.0. http://www.w3.org/TR/ws-cdl-10/.

[14] IBM. BPEL4WS (version 1.1), May 2003. http://www.ibm.com/developerworks/library/ws-bpel.

[15] active endpoints. activeBPEL. http://www.activevos.com/community-open-source.php.

[16] ORACLE. Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html.

[17] W3C. WS-Addressing. http://www.w3.org/Submission/ws-addressing/.

[18] W3C. WS-Policy. http://www.w3.org/TR/2007/REC-ws-policy-20070904/.

[19] W3C. WS-Security. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[20] OASIS. OASIS Web Services Reliable Messaging. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[21] IBM. WS-Transactions. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[22] Constantin Karapoulios Spyros Xanthakis, Pascal Regnier. *Le test des logiciels*. Hermès. 2000.

[23] CFTL. Le Comité Français des Tests Logiciels (CFTL). .

[24] Marlon Dumas and Reiko Heckel, editors. *Web Services and Formal Methods, WS-FM'07 Workshop, Brisbane, Australia, Sep., 2007. Proceedings*, volume 4937 of *LNCS*. Springer, 2008.

[25] Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors. *Web Services and Formal Methods, WS-FM 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, Vienna, Austria, September 2006. Springer.

[26] Shoichi Morimoto. A Survey of Formal Verification for Business Process Modeling. In *Proceedings of the 8th international conference on Computational Science, Part II (ICCS '08)*, pages 514–522, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Maurice, A. Bucciarone, and S. Gnesi. A Survey on Service Composition Approaches : From Industrial Standards to Formal Methods. Technical Report 2006-TR-15, Consiglio Nazionale delle Ricerche, 2006.

[28] Breugel Franck van and Koshkina Maria. Models and verification of BPEL, 2006. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

[29] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I : Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

[30] Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri Net Transformations for Business Processes - A Survey. *Transactions on Petri Nets and Other Models of Concurrency*, 2008.

[31] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for Web service composition. In *Proceedings of the fourteenth Australasian database conference (ADC '03)*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[32] Christian Stahl. A Petri Net Semantics for BPEL. Technical report, University, 2004. http://www2.informatik.hu-berlin.de/Institut/struktur/systemanalyse/preprint/stahl188.pdf.

[33] Chun Ouyang, Eric Verbeek, Wil M. van der Aalst, Stephan Breutel, Marlon Dumas, and Ter. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3) :162–198, July 2007.

[34] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. WofBPEL : A Tool for Automated Analysis of BPEL Processes. In *ICSOC*, pages 484–489, 2005.

[35] Karsten Schmidt. Distributed Verification with LoLA. *Fundamenta Informaticae*, 54(2-3) :253–262, 2003.

[36] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting WS-BPEL processes using flexible model generation. *Data Knowl. Eng.*, 64(1) :38–54, 2008.

[37] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In *Business Process Management*, pages 220–235, 2005.

[38] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing BPEL-based Web Service Composition Using High-level Petri Nets. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC '06)*, pages 441–444, 2006.

[39] Hui Kang, Xiuli Yang, and Sinmiao Yuan. Modeling and Verification of Web Services Composition based on CPN. In *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC '07)*, pages 613–617, 2007.

[40] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying Web Services Composition Based on Hierarchical Colored Petri Nets. In *Proceedings of the first international workshop on Interoperability of heterogeneous information systems (IHIS '05)*, pages 47–54, New York, NY, USA, 2005. ACM.

[41] YanPing Yang, QingPing Tan, Yong Xiao, JinShan Yu, and Feng Liu. Verifying Web Services Composition : A Transformation-Based Approach. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '05)*, pages 546–548, 2005.

[42] YanPing Yang, QingPing Tan, JinShan Yu, and Feng Liu. Transformation BPEL to CP-Nets for Verifying Web Services Composition. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWESP '05)*, page 137, 2005.

[43] Claude Girault and Rüdiger Valk. *Petri Nets for Systems Engineering — A Guide to Modeling, Verification, and Applications.* Springer Verlag, 2003.

[44] The CPN Group University of Aarhus Denmark. CPN Tools : Computer Tool for Coloured Petri Nets, 2009. `http://www.daimi.au.dk/CPNtools/`.

[45] Yongyan Zheng and P. Krause. Automata Semantics and Analysis of BPEL. In *Proceedings of the Inaugural IEEE-IES Digital EcoSystems and Technologies Conference (DEST '07)*, pages 147–152, 2007.

[46] Yongyan Zheng, Jiong Zhou, and Paul Krause. A Model Checking based Test Case Generation Framework for Web Services. In *Proceedings of the International Conference on Information Technology ITNG '07*, pages 715–722, 2007.

[47] Yongyan Zheng, Jiong Zhou, and Paul Krause. An Automatic Test Case Generation Framework for Web Services. *Journal of Software*, 2(3) :64–77, September 2007.

[48] ITC-IRST. NuSMV : a new symbolic model checker. `http://nusmv.irst.itc.it/`.

[49] G. J. Holzmann. The SPIN Model Checker : Primer and Reference Manual., 2003.

[50] R. Kazhamiakin, P. Pandya, and M. Pistore. Representation, Verification, and Computation of Timed Properties in Web Services Composition. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 497–504, 2006.

[51] R. Kazhamiakin, P. Pandya, and M. Pistore. Timed modelling and analysis in Web service compositions. In *Proceedings of the First International Conference on Availability, Reliability and Security (ARES '06)*, pages 840–846, 2006.

[52] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming BPEL into annotated deterministic finite state automata for service discovery. In *Proceedings of the IEEE International Conference on Web Services (ICWS '04)*, pages 316–323, 2004.

[53] B. Mahleko and A. Wombacher. Indexing Business Processes based on Annotated Finite State Automata. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 303–311, 2006.

[54] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS Web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS '05)*, volume 1, pages 293–301, July 2005.

[55] Yuhong Yan, Marie-Odile Cordier, Yannick Pencole, and Alban Grastien. Monitoring Web Service Networks in a Model-based Approach. In *Proceedings of the Third European Conference on Web Services (ECOWS '05)*, page 192, 2005.

[56] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th international conference on World Wide Web (WWW '04)*, pages 621–630, New York, NY, USA, 2004. ACM.

[57] Xiang Fu, Tevfik Bultan, and Jianwen Su. WSAT : A Tool for Formal Analysis of Web Services. In *Proceedings of the 16th Int. Conf. on Computer Aided Verification (CAV '04*, pages 510–514. Springer, 2004.

[58] W3C. XPath : XML Path Language. `http://www.w3.org/TR/xpath`.

[59] Rob Gerth. Concise Promela Reference, June 1997. `http://www.spinroot.com/spin/Man/Quick.html`.

[60] Shin NAKAJIMA. Lightweight formal analysis of Web service flows. *Progress in Informatics*, 2 :57–76, 2005.

[61] José Garcia-Fanjul, Claudio de la Riva, and Javier Tuya. Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. In *Proceedings of the Testing : Academic and Industrial Conference - Practice And Research Techniques (TAIC PART '06)*, pages 127–130, 2006.

[62] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe'06*, pages 83–94, Palermo, Italy, 2006.

[63] ISO. LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior, 1989.

[64] Radu Mateescu and Sylvain Rampacek. Formal Modeling and Discrete-Time Analysis of BPEL Web Services. *Int. J. of Simulation and Process Modelling*, 4 :183–194, 2009.

[65] INRIA. CADP : Construction and Analysis of Distributed Processes. Software Tools for Designing Reliable Protocols and Systems. `http://www.inrialpes.fr/vasy/cadp.html`.

[66] Howard Foster, Sebastián Urrutia, Jeff Magee, and Jeff Kramer. Web Service Compositions : From XML Syntax to Service Models. In *IDEAlliance XML Conference*, 2005.

[67] Mariya Koshkina. Verification of Business Processes for Web Services. Master's thesis, York University, Toronto, Ontario, October 2003.

[68] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5) :1–10, 2004.

[69] Matthias Weidlich, Gero Decker, and Mathias Weske. Efficient Analysis of BPEL 2.0 Processes Using p-Calculus. In *Proceedings of the The 2nd IEEE Asia-Pacific Service Computing Conference (APSCC '07)*, pages 266–274, 2007.

[70] Gwen Salaun, Andrea Ferrara, and Antonella Chirichiello. Negotiation Among Web Services Using LOTOS/CADP. In *ECOWS*, pages 198–212, 2004.

[71] Egon Börger, editor. *Specification and validation methods*. Oxford University Press, Inc., New York, NY, USA, 1995.

[72] R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical report, School of Computing Science, 2004.

[73] R. Farahbod, U. Glasser, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Abstract State Machines*, pages 78–94, 2004.

[74] CodePlex. AsmL : The Abstract State Machine Language. `http://www.codeplex.com/AsmL/`.

[75] Dirk Fahland. Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Informatik-Berichte 190, Humboldt-Universität zu Berlin, Sep. 2005.

[76] Dirk Fahland and Wolfgang Reisig. Asm-based semantics for bpel : The negative control flow. In *Proceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*, pages 131–152, 2005.

[77] Wolfgang Reisig. Modeling- and Analysis Techniques for Web Services and Business Processes. In Martin Steffen and Gianluigi Zavattaro, editors, *Formal Methods for Open Object-Based Distributed Systems : 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005. Proceedings*, volume 3535 of *Lecture Notes in Computer Science*, pages 243–258. Springer Verlag, May 2005.

[78] M. Butler, C. Ferreira, and M.Y. Ng. Precise Modelling of Compensating Business Transactions and its Application to BPEL. *j-jucs*, 11(5) :712–743, 2005.

[79] Yuan Yuan, Zhongjie Li, and Wei Sun. A Graph-Search Based Approach to BPEL4WS Test Generation. In *Proceedings of the International Conference on Software Engineering Advances (ICSEA '06)*, pages 14–14, Oct. 2006.

[80] Object Management Group. UML, Unified Modeling Language, version 2.1.2, 2008. `http://www.omg.org/technology/documents/formal/uml.htm`.

[81] M. Emilia Cambronero, Gregorio Diaz, J. Jose Pardo, and Valentin Valero. Using UML Diagrams to Model Real-Time Web Services. In *Proceedings of the Second International Conference on Internet and Web Applications and Services (ICIW '07)*, page 24, 2007.

[82] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing Service Composition. In *Proceedings of ASSE'07*, Mar del Plata, Argentina, August 2007.

[83] Sébastien Salva and Antoine Rollet. Automatic Web Service Testing from WSDL Descriptions. In *Proceedings of the 8th IEEE International Conference on Innovative Internet Community Systems (I2CS'08)*, Schoelcher, Martinique, 2008.

[84] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Towards automated wsdl-based testing of web services. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, pages 524–529, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] Stefan Troschütz. *Web Service Test Framework with TTCN-3*. PhD thesis, University of Göttingen, Germany, 2007.

[86] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. WSDL-based automatic test case generation for Web services testing. pages 207–212, 2005.

[87] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of Web Services for Testing Conformance to Open Specified Protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, number 3938 in Lecture Notes in Computer Science, pages 1–25. Springer, 2006.

[88] L. Frantzen, J. Tretmans, and R. d. Vries. Towards Model-Based Testing of Web Services. In A. Polini, editor, *International Workshop on Web Services - Modeling and Testing (WS-MaTe '06)*, pages 67–82, Palermo, Italy, 2006.

[89] Sébastien Salva and Issam Rabhi. Automatic Web Service Robustness Testing from WSDL descriptions. In *Proceedings of the 12th European Workshop on Dependable Computing (EWDC '09)*, Toulouse, France, 2009.

[90] Evan Martin, Suranjana Basu, and Tao Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS '06)*, 2006.

[91] Jeff Offutt and Wuzhi Xu. Generating Test Cases for Web Services using Data Perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5) :1–10, 2004.

[92] EVIWARE. soapUI Tool, 2008. `http://www.eviware.com/`.

[93] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. TAXI–A Tool for XML-Based Testing. In *Companion to the proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION '07)*, pages 53–54, 2007.

[94] Antonia Bertolino, Jinghua Gao, Eda Marchetti, and Andrea Polini. Automatic Test Data Generation for XML Schema-based Partition Testing. In *Proceedings of the Second International Workshop on Automation of Software Test (AST '07)*, page 4, 2007.

[95] ETSI. Testing and Test Control Notation Version 3 : TTCN-3. http://www.ttcn-3.org/.

[96] W3C. Document Object Model DOM. http://www.w3.org/DOM/.

[97] A. Bertolino and A. Polini. The Audition Framework for Testing Web Services Interoperability. pages 134–142, 2005.

[98] PushToTest. TestMaker. http://www.pushtotest.com/.

[99] WS-Unit. The Web Service Testing Tool. https://wsunit.dev.java.net/.

[100] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Ioannis Parissis. Data Flow-Based Validation of Web Services Compositions : Perspectives and Examples. pages 298–325, 2008.

[101] Chien-Hung Liu, Shu-Ling Chen, and Xue-Yuan Li. A WS-BPEL Based Structural Testing Approach for Web Service Compositions. In *Proceedings of the 2008 IEEE International Symposium on Service-Oriented System Engineering (SOSE '08)*, pages 135–141, 2008.

[102] Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. Bpel4ws unit testing : Test case generation using a concurrent path analysis approach. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*, pages 75–84, 2006.

[103] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori. Business-process-driven gray-box SOA testing. *IBM Syst. J.*, 47(3) :457–472, 2008.

[104] Philip Mayer and Daniel Lübke. Towards a BPEL unit testing framework. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications (TAV-WEB '06)*, pages 33–42, New York, NY, USA, 2006. ACM.

[105] Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. BPEL4WS unit testing : framework and implementation. In *Proceedings of the IEEE International Conference on Web Services (ICWS '05)*, pages 103–110, 2005.

[106] Philip Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. PhD thesis, Leibniz University, Hanover, Germany, September 2006.

[107] Philip Mayer. BPELUnit - The Open Source Unit Testing Framework for BPEL. http://www.se.uni-hannover.de/forschung/soa/bpelunit/.

[108] Parasoft. SOAtest. http://www.parasoft.com/jsp/solutions/soa_solution.jsp.

[109] ORACLE. Oracle BPEL Test Framework. http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/testsuite.htm.

[110] Hai Huang, Wei-Tek Tsai, Raymond Paul, and Yinong Chen. Automated Model Checking and Testing for Composite Web Services. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 300–307, 2005.

[111] Object Management Group. BPMN, Business Process Modeling Notation, 2009. http://www.bpmn.org/.

[112] Kathrin Kaschner and Niels Lohmann. Automatic Test Case Generation for Services. In Monika Solanki, Barry Norton, and Stephan Reiff-Marganiec, editors, *3rd Young Researchers Workshop on Service Oriented Computing (YR-SOC 2008)*, June 2008.

[113] G. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 3 :103–120, 1996.

[114] Jiehan Zhou, Juha-Pekka Koivisto, and Eila Niemelä. A Survey on Semantic Web Services and a Case Study. In *CSCWD*, pages 763–769, 2006.

[115] Jorge Cardoso. Approaches to developing semantic web services. *International Journal of Computer Science (IJCS)*, 1(1) :8–21, 2006.

[116] W3C. Web service description language with semantics (wsdl-s). `http://www.w3.org/Submission/WSDL-S/`.

[117] W3C. Web ontology web language for services (owl-s). `http://www.w3.org/Submission/OWL-S/`.

[118] W3C. Web services modeling ontology (wsmo). `http://www.wsmo.org/`.

[119] Mithun Sheshagiri. Automatic Composition and Invocation of Semantic Web Services. Master's thesis, UMBC, August 2004.

[120] Ruoyan Zhang, Ismailcem Budak Arpinar, and Boanerges Aleman-Meza. Automatic Composition of Semantic Web Services. In *International Conference on Web Services (ICWS '03)*, pages 38–41, 2003.

[121] Schahram Dustdar and Wolfgang Schreiner. A survey on Web Services Composition. *Int. J. Web Grid Serv.*, 1(1) :1–30, 2005.

[122] Shufang Lee, Xiaoying Bai, and Yinong Chen. Automatic Mutation Testing and Simulation on OWL-S Specified Web Services. In *Proceedings of the 41st Annual Simulation Symposium (ANSS '08)*, pages 149–156, 2008.

[123] Yongbo Wang, Xiaoying Bai, Juanzi Li, and Ruobo Huang. Ontology-Based Test Case Generation for Testing Web Services. In *Proceedings of the Eighth International Symposium on Autonomous Decentralized Systems (ISADS '07)*, pages 43–50, 2007.

[124] Ying Yu, Ning Huang, and Quizhong Luo. OWL-S Based Interaction Testing of Web Service-Based System. In *Proceedings of the Third International Conference on Next Generation Web Services Practices (NWESP '07)*, pages 31–34, 2007.

[125] Amit M. Paradkar, Avik Sinha, Clay Williams, Robert D. Johnson, Susan Outterson, Charles Shriver, and Carol Liang. Automated Functional Conformance Test Generation for Semantic Web Services. In *ICWS*, pages 110–117, 2007.

[126] Amit M. Paradkar, Avik Sinha, Clay Williams, Robert D. Johnson, Susan Outterson, Charles Shriver, and Carol Liang. Automated Functional Conformance Test Generation for Semantic Web Services. *IEEE International Conference on Web Services (ICWS '07)*, 0 :110–117, 2007.

[127] Ying Yu, Ning Huang, and Quizhong Luo. OWL-S Based Interaction Testing of Web Service-Based System. *Proceedings of the IEEE International Conference on Next Generation Web Services Practices (NWeSP '07)*, 0 :31–34, 2007.

[128] Andreas Both and Wolf Zimmermann. Automatic Protocol Conformance Checking of Recursive and Parallel Component-Based Systems. In Michel R. V. Chaudron, Clemens A. Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering, 11th International Symposium, CBSE 2008, Karlsruhe, Germany, October 14-17, 2008. Proceedings*, volume 5282 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2008.

[129] Yi Qian, Yuming Xu, Zheng Wang, Geguang Pu, Huibiao Zhu, and Chao Cai. Tool Support for BPEL Verification in ActiveBPEL Engine. In *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC '07)*, pages 90–100, 2007.

[130] Geguang Pu, Xiangpeng Zhao, Shuling Wang, and Zongyan Qiu. Towards the Semantics and Verification of BPEL4WS. *Electr. Notes Theor. Comput. Sci*, 151(2) :33–52, 2006.

[131] Honghua Cao, Shi Ying, and Dehui Du. Towards Model-based Verification of BPEL with Model Checking. In *CIT*, page 190, 2006.

[132] Raman Kazhamiakin. WS-Verify Tool. `http://dit.unitn.it/~raman/ws-verify.html`.

[133] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. VieDAME - flexible and robust BPEL processes through monitoring and adaptation. In *ICSE Companion '08 : Companion of the 30th international conference on Software engineering*, pages 917–918. ACM, 2008.

[134] Luciano Baresi, Sam Guinea, Raman Kazhamiakin, and Marco Pistore. An Integrated Approach for the Run-Time Monitoring of BPEL Orchestrations. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet (ServiceWave '08)*, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[135] Carlo Ghezzi and Sam Guinea. Run-Time Monitoring in Service-Oriented Architectures. In *Test and Analysis of Web Services*, pages 237–264. 2007.

[136] Marco Pistore and Paolo Traverso. Assumption-Based Composition and Monitoring of Web Services. In *Test and Analysis of Web Services*, pages 307–335. 2007.

[137] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of the International Conference on Web Services (ICWS '06)*, pages 63–71, 2006.

[138] Yuhong Yan, Y. Pencole, M.-O. Cordier, and A. Grastien. Monitoring Web Service Networks in a Model-based Approach. In *Proceedings of the Third IEEE European Conference on Web Services (ECOWS '05)*, page 12, 2005.

[139] H. Gros-Desormeaux, H. Fouchal, and P. Hunel. A Distributed Approach for Testing Timed Systems. In *Advanced International Conference on Telecommunications/International Conference on Internet and Web Applications and Services (AICT-ICIW '06)*, pages 94–94, 2006.

[140] Ismail Berrada, Richard Castanet, and Patrick Félix. Testing Communicating Systems : a Model, a Methodology and a Tool. In *17th IFIP International Conference on Testing of Communicating Systems*, Lecture Notes in Computer Science, Montreal, Canada, 2005. Elsevier.

[141] I. Berrada, R. Castanet, and P. Félix. A formal approach for Real-Time Test Generation. In *Proc. of Workshop on testing real-time and embedded systems (satellite event of FM)*, September 2003.

[142] Henrik Bohnenkamp and Alex Belinfante. Timed Testing with TorX. In *FM 2005 : Formal Methods*, Lecture Notes in Computer Science 3582, pages 173–188, 2005.

[143] Laura Brandan Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *Formal Approaches to Software Testing*, volume LNCS 3395 of *Lecture Notes in Computer Science 3395*, pages 64–78. Springer, 2005.

[144] L. Brandán Briones and M. Röhl. Test derivation from timed automata. In M. Broy, B. Jonsson, J. P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 201–231. Springer Verlag, Berlin, 2005.

[145] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-UPPAAL : Online Model-based Testing of Real-time Systems. In Paul Grünbacher, Virginie Wiels, and Kurt Stirewalt, editors, *19th IEEE International Conference on Automated Software Engineering*, 2004. Tool demo.

[146] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[147] Ahmed Khoumsi, Thierry Jeron, and Herve Marchand. Test Cases Generation for Nondeterministic Real-Time Systems. In *FATES*, pages 131–146, 2003.

[148] Sweden Uppsala University. UPPAAL, 2006. `http://www.uppaal.com/`.

[149] Ismail Berrada, Richard Castanet, and Patrick Félix. TGSE : Un outil générique pour le test. In *Colloque Francophone sur l'ingénierie des Protocoles (CFIP'05)*, pages 67–84, Bordeaux, France, 2005. Hermes.

[150] Ana Cavalli, Mounir Lallali, Stephane Maag, Gerardo Morales, and Fatiha Zaidi. *in Emergent Web Intelligence, Studies in Computational Intelligence*, chapter Modeling and testing of Web based systems. Springer Verlag, 2009. 42 pages.

[151] M. Lallali, F. Zaidi, and A. Cavalli. Timed Modeling of Web Services Composition for Automatic Testing. In *Proceedings of the Third International IEEE Conference on Signal-Image Technologies and Internet-Based System (SITIS '07)*, pages 417–426, 2007.

[152] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Definition of the Mapping from BPEL to WS-TEFSM. *Livrable WEBMOV-FC-D2.3/T2.4*, Novembre 2008. 40 pages.

[153] J. Bengtsson and Wang Yi. Timed Automata : Semantics, Algorithms and Tools. *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098, Springer-Verlag*, 2004.

[154] Kai Chen, J. Sztipanovits, and S. Abdelwahed. A Semantic Unit for Timed Automata Based Modeling Languages. In *Proceedings of RTAS'06*, pages 347–360, 2006.

[155] Moez Krichen and Stavros Tripakis. Black-Box Conformance Testing for Real-Time Systems. In *SPIN*, pages 109–126, 2004.

[156] André Arnold. *Finite transition systems*. Prentice-Hall. 1994.

[157] André Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson. 1992.

[158] Dorel Marius BOZGA. *Vérification symbolique pour les protocoles de communication*. PhD thesis, Université Joseph Fourier - Grenoble I, France, 1999.

[159] M. Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and J. Sifakis. The IF toolset. In *SFM-04*, volume 3185 of *LNCS*, pages 237–267. Springer-Verlag, June 2004.

[160] Marius Bozga and Yassine Lakhnech. IF-2.0 Common Language Operational Semantics, September 2002. `www-if.imag.fr/tutorials/semantics.ps.gz`.

[161] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF : An Intermediate Representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[162] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic Timed Test Case Generation for Web Services Composition. In *Proceedings of the IEEE Sixth European Conference on Web Services (ECOWS '08)*, pages 53–62, 2008.

[163] Mounir Lallali, Fatiha Zaidi, and Ana Cavalli. Transforming BPEL into Intermediate Format Language for Web Services Composition Testing. In *Proceedings of the 4th International Conference on Next Generation Web Services Practices (NWESP '08)*, pages 191–197, 2008.

[164] Ana Cavalli Mounir Lallali, Fatiha Zaidi and Patrick Felix. Automation of the Mapping from BPEL to WS-TEFSM-IF. *Livrable WEBMOV-FC-D2.4a/T2.5*, Mars 2009. 32 pages.

[165] Verimag/IMAG. IF Toolset, 2003. `www-if.imag.fr/`.

[166] U. Glässer, R. Gotzhein, and A. Prinz. The formal semantics of SDL-2000 : status and perspectives. *Comput. Netw.*, 42(3) :343–358, 2003.

[167] K. K. Sandhu. Specification and description language SDL. In *Proceedings of the IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*, pages 3/1–3/4, 1992.

[168] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1) :25–59, 1987.

[169] Marius Bozga and Yassine Lakhnech. IF-2.0 Language : Concrete Syntax Annotated, 2003. `http://www-if.imag.fr/tutorials/syntax.ps.gz`.

[170] Verimag. Common Description Language, June 2003. `http://www-if.imag.fr/tutorials/introduction.ps.gz`.

[171] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0 : A Validation Environment for Component-Based Real-Time Systems. In *Proceedings of The International Conference on Computer Aided Verification (CAV'02)*, pages 343–348, London, UK, 2002. Springer-Verlag.

[172] Richard Castanet Patrick Felix Mounir Lallali Fatiha Zaidi Ismail Berrada, Tien-Dung Cao and Ana Cavalli. Définition d'une méthode de génération de test pour la composition de services Web décrite en BPEL. *Livrable WEBMOV-FC-D4.1/T4.1*, Juin 2009. 44 pages.

[173] IEEE. the Institute of Electrical and Electronics Engineers, Inc. `http://www.ieee.org/web/aboutus/home/index.html`.

[174] Cross Checknet Networks. SOA Testing Tools, 2008. `http://www.crosschecknet.com/soa_testing_black_white_gray_box.php`.

[175] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing : unit testing with mock objects. pages 287–301, 2001.

[176] Oracle. Interaction Patterns, 2007. `http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/interact.htm`.

[177] Jan Tretmans. Testing Concurrent Systems : A Formal Approach. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR '99)*, pages 46–65, London, UK, 1999. Springer-Verlag.

[178] Anders Hessel, Kim Guldstrand Larsen, Marius Mikuèionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. pages 77–117. Spring-Verlag, 2008.

[179] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theor. Comput. Sci.*, 254(1–2) :225–257, 2001.

[180] Ismail Berrada, Richard Castanet, Patrick Félix, and Aziz Salah. Timed diagnostics and test Case minimization for real-time systems. In *Proceedings of the 18th IFIP International Conference on Testing of Communicating Systems (TESTCOM '06)*, Lecture Notes in Computer Science. Elsevier, 2006.

[181] Ana Rosa Cavalli, Edgardo Montes De Oca, Wissam Mallouli, and Mounir Lallali. Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints. In *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2008)*, pages 315–318, 2008.

[182] Wissam Mallouli, Mounir Lallali, Gerardo Morales, and Ana Rosa Cavalli. Modeling and Testing Secure Web-Based Systems : Application to an Industrial Case Study. In *Proceedings of the IEEE International Conference on Signal Image Technology and Internet Based Systems (SITIS '08)*, pages 128–136, 2008.

[183] Iksoon Hwang, Mounir Lallali, Ana Cavalli, and Dominique Verchere. Modeling, validation, and test generation of pcep using formal method. In *The IFIP International Conference on Formal Techniques ofr Distributed Systems FMOODS/FORTE'09*, 9-11 June 2009. 15 pages.

[184] W3C. Extensible Stylesheet Language Transformations XSLT. `http://www.w3.org/TR/xslt`.

[185] Verimag. IFx Toolbox. `http://www-omega.imag.fr/tools/IFx/IFx.php`.

[186] Enrique Martí andnez, Marí anda Emilia Cambronero, Gregorio Dí andaz, and Valentí andn Valero. Design and Verification of Web Services Compositions. In *Fourth International Conference on Internet and Web Applications and Services (ICIW '09)*, pages 395–400, May 2009.

[187] Hongli Yang, Xiangpeng Zhao, Zongyan Qiu, Geguang Pu, and Shuling Wang. A Formal Model for Web Service Choreography Description Language (WS-CDL). In *International Conference on Web Services (ICWS '06)*, pages 893–894, Sept. 2006.

[188] W.L. Yeung. Mapping WS-CDL and BPEL into CSP for Behavioural Specification and Verification of Web Services. In *4th European Conference on Web Services (ECOWS '06)*, pages 297–305, Dec. 2006.

[189] P. Poizat and J.-C. Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. *j-jucs*, 12(12) :1741–1782, 2006.

[190] Thierry Jéron. Symbolic model-based test selection. In P. Machado, A. Andrade, and A. Duran, editors, *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2008), Salvador, Bahia, Brazil*, pages 17–32, 2008.

[191] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in Lecture Notes in Computer Science, pages 40–54. Springer, 2006.

[192] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP : a tool for the formal verification of UML/OCL models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE '07)*, pages 547–548. ACM, 2007.

[193] Martin Gogolla, Fabian Büttner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1–3) :27–34, 2007.

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Abréviations et acronymes

aDFA annotated Deterministic Finite State Automata

ARP Arbre de Recherche Partielle

ASM Abstract State Machine

AsmL Abstract State Machine Language

ATC Architecture de Test Centralisée

ATD Architecture de Test Distribuée

Atp Algebra of Timed Processes

BCFG WS-BPEL Control Flow Graph

BFG BPEL Flow Graph

BPEL Business Process Execution Language

BPMN Business Process Modeling Notation

CCS calculus of communicating systems

CS Communicating Systems

CFG Control Flow Graph

CPN Colored Petri Nets

CTL Computation tree logic

DES Discrete Event System

DOM Document Object Model

EFA Extended Finite State Automaton

EFSM Extended Finite State Machine

FSP Finite State Processes

FIFO First In First Out

GFSA Guarded Finite State Automata

HPN High-level Petri Nets

IF Intermediate Format

LOTOS Language Of Temporal Order Specification

LTS Labeled Transition System

LTSA Labelled Transition System Analyser

MDA Model Driven Architecture

OCL Object Constraint Language

oWFN open Workflow Nets

Promela Process Meta Language

PSM Protocol State Machine

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

SOA Service Oriented Architecture

SST Service Sous Test

ST Service Testeur

stAC Structured Activity Compensation

STS State Transition System

TA IF Timed Automaton

TGSE Émulation, Simulation et Génération de Test

TTCN-3 Testing and Test Control Notation version 3

UDDI Universal Description, Discovery and Integration

UML Unified Modeling Language

USE A UML-based Specification Environment

WS-CDL Web Services Choreography Description Language

WS-TEFSM Timed Extended Finite State Machine(s) for Web Service

WSA Web Service Automata

WSAT Web Service Analysis Tool

WSDL Web Service Description Language

WSTTS Web Service Timed State Transition Systems

XCFG Extended Control Flow Graph

XML eXtensible Markup Language

Index