UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

ECOLE DOCTORALE SCIENCES, TECHNOLOGIES ET SANTÉ

THÈSE de DOCTORAT

présentée par

Sylvain RAMPACEK

pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne

Spécialité : Informatique

Sémantique, interactions et langages de description des services web complexes

présentée et soutenue publiquement le 10 novembre 2006

Composition du jury :

M. B. Jean FUGÈRE	Professeur au Collège Militaire du Canada	Examinateur
M. Serge HADDAD	Professeur à l'Université Paris Dauphine, LAMSADE	Directeur
Mme Hanna KLAUDEL	Professeure à l'Université d'Evry, IBISC	Rapporteur
M. Fabrice KORDON	Professeur à l'Université Paris 6, LIP6	Rapporteur
M. Michaël KRAJECKI	Professeur à l'Université de Reims, CReSTIC	Président
M. Patrice MOREAUX	Professeur à l'Université de Savoie, LISTIC	Directeur

LABORATOIRE CReSTIC – DÉPARTEMENT SysCom

« Il existe 10 types de personnes :
celles qui savent compter en binaire et les autres... »

Remerciements

Je tiens à remercier, tout d'abord, mes directeurs de thèse, Serge Haddad et Patrice Moreaux, pour l'encadrement de mon travail et leurs apports tant au niveau des connaissances qu'au niveau humain, mais également pour leurs encouragements, ainsi que leur soutien tout au long de la thèse.

Je remercie tout particulièrement Mme Hanna Klaudel et M. Fabrice Kordon d'avoir accepté d'être rapporteurs et ainsi d'avoir accepté de lire et de commenter le manuscrit ; mais également M. Michaël Krajecki d'être président de mon jury, et enfin M. B. Jean Fugère d'être examinateur.

Un grand merci également aux collègues de bureau, Cyril, Thibault, Arnaud, Christophe et les membres de l'équipe LICA qui se reconnaîtront, pour leur accueil et la bonne ambiance. Ainsi qu'un merci aux secrétaires du département pour leur disponibilité et leur aide dans les tâches administratives. Sans oublier également de remercier les membres et doctorants du LAMSADE qui m'ont toujours accueilli chaleureusement lors de mes journées de travail sur Paris.

Je tiens à remercier tout particulièrement, mon frère Stéphane, ma mère et mon père pour leur soutien dans toutes les situations tout au long de ce travail, et sans oublier Laëtitia, ma copine, qui ont tous dû me supporter durant toutes les périodes et étapes de cette thèse.

Enfin, je remercie toutes les personnes qui m'ont aidé tout au long de la thèse et encouragé à aller dans cette direction, ainsi que mes amis de longue date : Pascal, Anne et Gilles.

Table des matières

Table des figures

Introduction

Introduction

1. Internet et évolution des architectures des systèmes d'information

Le réseau des réseaux (Internet) a pris, durant ces dix dernières années, une importance considérable dans le domaine informatique. Cette évolution, rendue possible par les performances toujours plus élevées des matériels (réseaux et ordinateurs), est à la fois la conséquence et le moteur de modifications profondes dans l'architecture même des systèmes d'information des organisations, depuis la petite entreprise jusqu'aux grands groupes internationaux. Cette place prépondérante d'Internet est également à mettre en relation avec le développement, depuis le début des années 90, d'une économie mondialisée.

La tendance est vers l'utilisation des architectures orientées services[1]. Cette utilisation permet alors de conserver le code existant (*legacy code*) et d'exposer ces programmes à travers Internet par de nouvelles technologies favorisant l'interopérabilité.

Apparition des technologies liées à XML Ces modifications profondes se traduisent par l'apparition de technologies de codage ou de transport d'informations, indépendantes de la source et du destinataire. Le but étant de permettre les communications à travers le réseau, sans se soucier de l'hétérogénéité des clients, des programmes et des données à traiter. Ainsi est apparu le langage de description XML [YCB+04], un standard du W3C (*World Wide Web Consortium*). Ce langage permet de décrire les données sous forme d'un langage balisé. Une description, dans le même format, des types de données utilisées, peut y être ajoutée : celle-ci est alors appelée une *grammaire*. L'utilisation du langage XML a permis de nombreuses créations de technologies, de langages, et de protocoles basés eux-mêmes sur ce langage.

Des langages de programmation liés à ces nouvelles technologies sont apparus, tels que C# [Sma03] ou Java [DF02]. Ils permettent de répondre aux nouveaux problèmes de codage de données, mais également de répartitions des calculs. Pour cela, l'approche adoptée repose sur une API standard très fournie, permettant de réaliser ces tâches.

Plateformes de développement En parallèle à ces langages, des plateformes de gestion d'applications liées à Internet sont apparues. Ainsi, les serveurs d'applications J2EE ou encore .Net [MBG03] permettent d'exploiter ces nouvelles technologies liées à Internet, en donnant aux programmeurs des outils allant du développement au déploiement d'applications, mais également des outils de débogage à travers le réseau. Ces nouvelles plateformes vont donc bien au delà des technologies précédentes à bus (CORBA, [GGM99]) ou à invocation de procédures à distance (RPC, [Whi75]).

[1] voir par exemple `http://xmlfr.org/actualites/decid/051109-0001`

Middleware En parallèle à ces plateformes de développement, sont arrivés les intergiciels (*middleware*) apportant des solutions aux problèmes liés à la répartition des données et des traitements. Dans ces intergiciels, des composants sont prêts à l'emploi pour la gestion de la concurrence, des contrats, des communications sécurisées, des services de nommage, et de la réutilisation des processus existants (généralement par le biais de l'encapsulation). Ils sont basés sur les échanges de messages asynchrones, contrairement au principe du requête/réponse, permettant la gestion des événements (MOM - *Message Oriented Middleware*).

Les problèmes d'interopérabilités sont nombreux. En effet, il n'est pas rare d'avoir recours à plusieurs intergiciels dans une même infrastructure logicielle, chacun répondant à des besoins particuliers (choix historiques ou besoins différents). Ainsi, des intergiciels comme PolyORB permettent la communication entre plusieurs intergiciels. Par exemple, une étude de cas a été réalisée et a permis de montrer l'interopréabilité entre des intergiciels de type MOM et de type ORB (*Object Request Broker*) [HKPQ02].

Réutilisation de l'existant Ces nouvelles technologies ont comme impératif la réutilisation de l'existant, c'est-à-dire la réutilisation des logiciels patrimoniaux (*legacy code*) et des données existantes. En effet, les industriels ne peuvent se permettre de perdre des données accumulées durant les années d'informatisation. Mais les besoins et les outils ont changé : maintenant, les ordinateurs sont tous reliés aux différents réseaux, y compris ceux des fournisseurs et des clients. La conservation des données est primordiale, mais également la conservation des logiciels. Le développement, les tests et la stabilisation de solutions logicielles ont pris plusieurs années, et il est impossible de repartir de zéro. Seules les solutions permettant l'adaptation par différents moyens (encapsulation, convertisseur, adaptateur, etc...) sont viables. D'où le principal succès des technologies présentées ci-dessus, outre le fait qu'elles répondaient à la demande au niveau des services nouveaux pour le développement de nouvelles applications.

Vers les services web Les besoins sont différents entre l'accès à la base de données à partir de l'Intranet de l'entreprise, et l'accès aux mêmes données (ou un sous-ensemble de ces données) par les fournisseurs et les clients de cette même entreprise. En effet, la solution à base de bus d'objets comme CORBA n'est viable que dans le cas d'un Intranet, et ce, à cause des déploiements à réaliser sur chaque poste client du bus, et de la latence réduite acceptable par de telles applications. Pour accéder aux données à travers Internet, la solution des services web, que nous allons présenter, est donc une des plus appropriée aujourd'hui.

2. Les services web : une étape vers les architectures orientées services

Présentation des services web Les *services web* consistent à exposer sur un réseau (et donc Internet), une ou plusieurs applications répondant à certains impératifs technologiques [SBA+02, Cha02]. Ces services peuvent proposer des fonctions très simples (du type requête/réponse) ou un ensemble complet d'outils, permettant d'aller jusqu'à la composition des services pour proposer une application complète.

Le client de ce service, qui peut être un humain ou une application, est situé sur la même machine ou sur une machine distante. Il devra, lui aussi, respecter certains impératifs permettant de répondre au problème principal d'interopérabilité. Cette interopérabilité concerne non seulement le codage des données, mais également les plateformes : le client n'est pas obligé de connaître la plateforme utilisée

par le fournisseur de services pour communiquer avec celui-ci. Les plateformes doivent être totalement indépendantes des technologies clés citées ici.

La communication entre le client et le service passe par une phase de découverte et de localisation du service, à l'aide du protocole et des annuaires UDDI [TBLCDE+02]. Cet annuaire contient un ensemble de fichiers de descriptions de services web, utilisant le langage WSDL[CCMW01], basé sur XML. Le client interroge l'annuaire à l'aide de mots clés pour obtenir un ensemble de descriptions WSDL. Ces descriptions WSDL contiennent toutes les informations nécessaires à l'invocation du service (URL de localisation, description des fonctions et des types de données).

Les communications entre les différents acteurs (service, client, annuaire) utilisent le protocole SOAP [Mit02], également basé sur XML. Elles utilisent les protocoles réseaux tel que HTTP pour faire parvenir les messages, permettant ainsi de s'affranchir des différents problèmes de pare-feux liés aux architectures réseaux.

Propriétés et langages Les architectures orientées services reposent sur l'utilisation d'un ensemble de services web. Les propriétés de telles architectures sont :

- la dynamicité, liée au fait que les services web reposent sur des communications entre un client et un service. Si la localisation d'un service est souvent stable (pour permettre de le localiser sur le long terme), il n'en est pas de même pour le client, car celui-ci est souvent connecté à travers Internet par un fournisseur d'accès ne lui attribuant pas une adresse IP fixe par exemple.
- l'interopérabilité, présente à plusieurs niveaux. Tout d'abord, entre les services eux-mêmes : en effet, rien n'oblige à utiliser la même plateforme entre les différents services formant une application à part entière. Ensuite, au niveau des clients et des services : l'architecture logicielle et matérielle peuvent être totalement différentes, l'essentiel est la mise à disposition des protocoles des services web sur ces architectures, indépendamment du langage et des logiciels utilisés.
- l'existence de langages spécifiques de descriptions comportementales. Ce type d'application, étant dynamique et ne reposant pas sur un serveur centralisé, requiert une description du comportement de l'application complète, indiquant comment les communications vont se réaliser entre les différents services. Ce type de langage hérite des travaux réalisés dans la gestion des *workflows*, permettant d'indiquer les différents traitements effectués sur un ensemble de données pour un acteur donné.

Parmi les langages de descriptions comportementales, XLANG [Tha01] et BPEL4WS [ACD+03] seront les deux utilisés dans nos travaux présentés par la suite. Le premier pour des raisons historiques, et le deuxième car il est devenu un standard de fait par son utilisation industrielle et va bientôt devenir un standard Oasis avec sa version 2.0.

3. Problématique

Contexte de la thèse L'axe principal de mon travail de thèse se situe dans la continuité du travail réalisé au laboratoire LAMSADE. Le départ de la réflexion est présenté dans la première partie de la thèse de Tarak Melliti [Mel04]. Les deux dernières années de sa thèse ont été menées en commun avec mon travail de stage de DEA [Ram03] et le début de ma propre thèse. Cette collaboration a mené à un travail théorique commun sur la sémantique formelle des services web dans le cadre temps discret [HMMR04b] et une première réflexion sur la sémantique formelle en temps dense [HMMR04a]. Le contexte du travail présenté ici se situe à la frontière de deux domaines :

- le domaine du *génie logiciel* par le développement de modules génériques dans le cadre d'une plateforme « services web » ;
- le domaine de la *vérification formelle* : par la conception et l'implémentation d'algorithmes de vérification.

Nécessité d'une sémantique opérationnelle L'utilisation d'architectures basées sur les services web, présente le problème de la cohérence des services. En effet, comment peut-on être sûr qu'une application, basée sur des services web distants, et interconnectés par des réseaux dont on ne connaît presque rien, aboutira par la bonne composition de services web à une application correcte ? Pour cela, il est nécessaire d'avoir une sémantique opérationnelle précise des langages de description comportementale de ces applications. D'où l'intérêt de notre approche et de l'apport d'une sémantique opérationnelle pour le langage BPEL par exemple.

Cette sémantique opérationnelle permet alors d'appliquer des méthodes formelles pour vérifier certaines propriétés. Pour ce faire, il est nécessaire d'utiliser des modèles temporisés. En effet, l'aspect temporel de la modélisation des systèmes (et donc des services) à vérifier est indispensable ici : l'interaction entre le client et le service, possède de nombreuses gardes au niveau du temps pour éviter des blocages, conséquences d'un problème de connexion par exemple. Les modèles formels généralement utilisés sont les algèbres de processus [BPS01], les réseaux de Petri [Dia01] ou encore les automates [AD94]. Ils permettent ainsi de décrire, par un modèle formel, le comportement des différents processus, et ensuite d'utiliser divers algorithmes permettant l'extraction de propriétés.

Une approche fondée sur les automates temporisés L'approche adoptée dans cette thèse consiste à traduire la sémantique opérationnelle du langage de description comportementale, dans un modèle de haut niveau tel que les TIOTS ou les automates temporisés. Ceci est similaire aux travaux des auteurs de [DKK05], qui traitent d'un exemple de sémantique à base d'une algèbre de processus, dont le comportement est traduit en réseau de Petri de haut niveau.

Ici, l'approche fondée sur les automates temporisés, est pertinente pour plusieurs raisons. Tout d'abord, le cadre des automates temporisés est naturel pour ce type de systèmes. Les transitions sont étiquetées par les actions que le système peut exécuter : principalement envoi ou réception de messages. De même, les gardes temporelles représentent correctement la mesure des écoulement de temps sur le système.

Ensuite, c'est un modèle permettant d'énoncer puis de vérifier certaines propriétés. Pour cela, nous avons défini une relation d'interaction. Enfin, la mise en évidence de cette ambiguïté découle de cette relation d'interaction, basée sur les ensembles états et les transitions de ces automates.

Des solutions logicielles adaptables Le domaine des services web est un domaine très mouvant et utilise les nouvelles technologies. Ces nouvelles technologies évoluant très rapidement, il devient nécessaire d'utiliser des méthodes génériques et d'automatiser le plus grand nombre de tâches. L'automatisation des tâches n'est pas un concept nouveau, les premiers langages de programmation ou les langages de *scripts shell* en sont des exemples. Mais ici, il faut également tenir compte de l'évolution des langages, des spécifications, etc... exigeant l'adaptabilité la plus rapide possible, d'où une automatisation de certaines tâches.

Ainsi, l'utilisation de méthodes *méta-programmation* ou basées sur la *généricité* est un élément clé des étapes de conception et développement. Par exemple, malgré l'évolution des technologies ou langages, des points communs apparaissent entre eux, formant le fil conducteur de la généricité et de l'adaptabilité de nos solutions.

4. Apports de la thèse

Sémantique formelle Le premier apport concerne la définition d'une sémantique formelle, en temps discret (TIOTS), et en temps dense (automates temporisés), applicable aux langages de description comportementale de services web. Dans le cas présent, cette sémantique est définie pour le langage XLANG, historiquement, puis adaptée, en utilisant la généricité de notre approche, au langage BPEL. La sémantique comportementale est issue de la description du processus (représentant le service web) par une algèbre de processus temporisés.

Détection de l'ambiguïté et génération d'un client adapté À partir de cette approche formelle, nous définissons une relation d'interaction, vérifiant un ensemble de propriétés sur les modèles. Ensuite, nous proposons une définition d'un service incorrect, basée sur la notion d'ambiguïté de service, découlant des vérifications réalisées par la relation d'interaction. Cette détection d'ambiguïté de service permet de définir si l'interaction, consistant aux échanges de messages visibles entre un service et son client, peut aboutir sur une incohérence ou un blocage, relevant de la présence d'une ambiguïté.

Par l'adaptation des méthodes précédentes à un algorithme de génération, il est possible, à partir du modèle obtenu pour le service, de générer un client. Ce client n'est généré que dans le cas de non présence d'ambiguïté du service. Nous proposons une solution en temps discret et en temps dense.

Vérification d'une chorégraphie Nous avons également étendu notre approche de détection d'ambiguïté, dans le cas temps discret, à la détection d'ambiguïté dans l'interaction des différents partenaires d'une chorégraphie (c'est-à-dire une composition de services web). Cela permet ainsi de valider l'interaction d'une application reposant sur plusieurs services web dont chaque partenaire connaît les partenaires avec lesquels il doit interagir.

Implémentation et mise en œuvre des résultats Enfin, nous avons personnellement implémenté les algorithmes et méthodes présentés ; et ce, en utilisant des méthodes et outils permettant la généricité de l'approche en vue d'adapter nos recherches à d'autres langages. Ce cas a par ailleurs été exploité dans l'adaptation de notre sémantique du langage XLANG vers le langage BPEL. Le but de l'implémentation de ces algorithmes est d'obtenir un modèle du client, à partir de la seule description d'un service web répondant à certains critères (décrit par un langage tel que BPEL).

Ce modèle client est ensuite transmis à un autre de nos programmes en cours de réalisation, dont le but est de suivre les indications fournies dans le modèle, pour réaliser une invocation du service guidée par l'utilisateur ou une application. Cette approche permet alors d'obtenir un client générique. Le but final est la perspective d'une intégration dans le cadre d'une plateforme « services web » développée par une équipe du LAMSADE.

5. Plan de la thèse

Chapitre 1 - Historique Dans une première partie, un état de l'art est réalisé. Le chapitre 1 présente un rapide historique des différentes technologies, reprenant et complétant les différents points abordés dans cette introduction, pour permettre d'aborder la présentation et les questions amenant aux services web.

Chapitre 2 - Présentation des services web Le chapitre 2 présente plus en détails les services web, reprenant l'évolution des méthodes de programmation ayant donné naissance à ce concept, mais également en présentant la définition d'un service web par le W3C. Une rapide présentation des différents langages et technologies de services web est réalisée. Et enfin, le développement d'applications entières basées sur les services web (SOA) est présenté ainsi que les langages de description associés que nous utilisons par la suite.

Chapitre 3 - Sémantique temporisée Le chapitre 3 réalise un état de l'art sur les différentes méthodes permettant la formalisation des services web. Les systèmes de transitions dans le cadre d'une sémantique temporisée sont présentés, ainsi que les automates temporisés, suivi d'une présentation de certains langages formels, clés de voûte de l'application des méthodes de vérifications. Enfin, les différentes approches connues permettant de vérifier certaines propriétés sur les services web sont évoquées.

Chapitre 4 - Sémantique en temps discret La partie 2 se focalise sur les résultats théoriques apportés par la thèse ainsi que sur leurs mises en œuvres, en réponse aux problèmes énoncés précédemment. Ainsi, le chapitre 4 aborde nos apports réalisés dans la formalisation des services web en présentant notre sémantique en temps discret. Il définit, dans un premier temps, la notion d'interaction puis présente notre méthode de vérification de cette interaction, basée sur la détection d'une possible ambiguïté entre l'interaction d'un potentiel client et d'un service donné. Le contenu de ce chapitre poursuit un premier travail concernant la réflexion de la formalisation et de l'écriture d'une sémantique d'un langage de description comportementale pour services web, réalisé dans le cadre de mon DEA [Ram03], mais également dans le cadre de la thèse [MH03]. Enfin, ce travail a été présenté lors d'une conférence [HMMR04b].

Chapitre 5 - Vérification d'une chorégraphie Le chapitre 5 étend notre vérification d'interaction à un ensemble de services web évoluant dans le cadre d'une chorégraphie. Pour cela, une définition formelle d'une chorégraphie et de ses partenaires est donnée. Notre sémantique y est adaptée pour la formalisation d'un partenaire, laissant apparaître des échanges de messages non modélisés dans l'approche précédente, car correspondant à des échanges de messages non observés entre le client et le service. Une définition d'agrégation de partenaires est présentée, permettant de définir le comportement observable de plusieurs partenaires, perçu comme un unique partenaire. Enfin, les algorithmes adaptés à la vérification de la chorégraphie sont présentés. Ces travaux ont été acceptés pour être présentés dans [MBSR06].

Chapitre 6 - Sémantique en temps dense Le chapitre 6 étend la sémantique en temps discret au temps dense pour s'affranchir du problème de l'explosion du nombre d'états liée à une telle modélisation. Ainsi, les adaptations de la sémantique de nos opérations et les nouvelles règles de notre algèbre y sont présentées. Les différentes questions concernant l'implémentation de nos algorithmes sont abordées, concernant principalement la gestion des automates temporisés et de leurs horloges, mais également la génération du modèle client. Ces travaux ont fait l'objet de deux publications [HMMR04a] (en se focalisant principalement sur la sémantique) et [HMR06] (en se focalisant principalement sur la génération du modèle de la partie client).

Chapitre 7 - Implémentation et mise en œuvre Enfin, le chapitre 7 présente la mise en oeuvre personnelle des algorithmes présentés dans les précédents chapitres. Ainsi, les technologies utilisées et les choix adoptés y seront présentés. Le but est d'obtenir un client générique permettant l'invocation d'un service web disponible à travers un réseau et dont nous n'avons pas la maîtrise. Ce service web devra répondre à certains critères concernant son interaction (pour éviter, par exemple, l'interblocage de l'exécution) ou encore l'obligation d'utiliser un langage de description comportementale décrivant le processus métier qu'il représente, c'est-à-dire les articulations entre toutes les fonctionnalités de celui-ci (par exemple BPEL). Ce travail s'inscrit dans le cadre d'une réalisation de plateforme de services web.

Première partie

État de l'art

Chapitre 1

Historique

Les services web et les technologies associées sont très présents dans l'informatique d'aujourd'hui. Pour cela, de nombreuses circonstances ont été nécessaires, en parallèle à l'évolution des ordinateurs, et des réseaux, sans oublier les habitudes des utilisateurs. Ce chapitre propose un historique de cette évolution des différentes technologies amenant à la problématique des services web. Cela ne représente qu'un choix restreint et d'autres théories peuvent s'appliquer.

1.1 Démocratisation de l'Internet

L'un des facteurs majeurs, si ce n'est le principal facteur, de l'avènement des services web sur les ordinateurs de tous les jours est la démocratisation des réseaux et d'Internet en particulier. En effet, alors qu'avant 1995-1998, un ordinateur était principalement destiné à fonctionner de façon autonome. Aujourd'hui, un ordinateur sans accès au réseau des réseaux se trouve limité au niveau de ses fonctionnalités et de l'accès aux informations. Ce facteur est, de plus, présent au niveau des entreprises, mais également des particuliers, d'où une évolution et un changement totalement radical de l'utilisation de l'ordinateur.

1.1.1 Parc d'ordinateurs hétérogène

L'interconnexion des ordinateurs entre eux par l'intermédiaire d'un réseau (et par extension d'Internet) pose le problème de la gestion du parc hétérogène. En effet, l'hétérogénéité se traduit :

- par la multitude des systèmes d'exploitations ;
- par les différentes versions majeures et mineures plus ou moins compatibles d'un même système et de ses services ;
- par les différentes architectures possibles.

L'interconnexion d'un tel parc présente de nombreux artifices permettant une communication plus ou moins aisée. C'est pour cela que les différentes technologies, abordées par la suite, tentent d'uniformiser les communications en utilisant des langages de communication standardisés et indépendants de la plateforme source et/ou cible.

Le parc d'ordinateurs est très hétérogène au niveau de l'architecture et des systèmes, comme nous venons de le voir, mais également au niveau de la puissance de calcul. Aujourd'hui, la puissance peut être très faible, comme dans le monde de l'embarqué (petite puissance de calcul, peu de mémoire, peu d'énergie, tel que nos PDA dont l'horloge du processeur est cadencée entre 100 et 300 Mhz). Mais cette puissance peut être énorme, comme pour certains serveurs ou encore des calculateurs permettant

de réaliser des simulations météorologiques ou nucléaires (voir par exemple le calculateur NovaScale 5160, comptant 8704 processeurs Itanium2 1.6 GHz, au Commissariat à l'Énergie Atomique (CEA)). Ces différences doivent cependant permettre à un ensemble d'ordinateurs de pouvoir communiquer à travers un réseau sans restreindre l'utilisation de l'un ou de l'autre.

1.1.2 Sous-utilisation des ressources et répartitions des calculs

L'évolution des capacités de calcul des ordinateurs est très liée au nombre et à la complexité des semi-conducteurs utilisés pour la fabrication de leurs processeurs. La LOI DE MOORE indique « *que la complexité des semi-conducteurs proposés en entrée de gamme doublait tous les ans depuis 1959* » date de leur invention.

La plupart des ordinateurs sont cependant sous-utilisés. L'exemple le plus flagrant est l'ordinateur de bureau qui reste allumé la nuit : à part quelques transferts réseaux liés aux scripts de sauvegarde et aux mises à jours logiciels, il n'a aucune information à traiter pendant ce laps de temps.

C'est ainsi que sont nés des projets scientifiques comme *SETI@home*[1] lancé en 2000, et permettant à tout un chacun de télécharger un petit logiciel récupérant des données provenant d'un radiotélescope pour les traiter durant les temps d'inactivités de l'ordinateur. Un autre exemple est le projet *Décrypton*[2], lancé par l'AFM (Association Française contre les Myopathies) et IBM suite au Téléthon 2001, dont le but est de réaliser la première cartographie du protéome humain (c'est-à-dire la modélisation de l'ensemble des protéines produites par le génome humain) qui a mobilisé 75000 ordinateurs permettant de réaliser en 2 ans ce qu'un ordinateur, à lui tout seul, aurait mis 1170 années !

Tous ces calculs ne sont possibles que par l'entente mutuelle traduite par les communications entre des serveurs et des ordinateurs hétérogènes à travers un réseau non stable (Internet). Ces communications utilisent alors des protocoles (pour la communication) et des langages (pour le codage des données) se rapprochant des technologies proches des services web que nous présenterons dans la suite de cette thèse.

Il est à noter que ces approches de gestion de l'interopérabilité ne sont pas récentes et ont toujours existé depuis l'apparition des premiers réseaux (ARPANET –*Advanced Research Projects Agency Network*–, projet lancé en 1967, première démonstration en 1972). Mais ces dernières années, cela est rendu encore plus présent par l'utilisation d'un plus grand nombre d'ordinateurs et l'automatisation obligatoire des différentes tâches de communications à travers un réseau. Cette automatisation se réalise sans même savoir précisément les destinataires des données et par quels moyens elles vont être transmises.

1.2 Apparition de Java

Parmi les technologies nées pour apporter des solutions aux problèmes présentés précédemment, le langage Java joue un grand rôle. Java est une technologie composée d'un langage de programmation orienté objet et d'un environnement d'exécution, développé par Sun. La première version fut présentée au SunWorld en 1995. Plusieurs versions ont suivi et aujourd'hui, la dernière version stable est Java 1.5 (ou J2SE 5.0) apportant de nombreuses modifications telle que la gestion des *generics*, un vrai type d'énumération, etc... Enfin, la version en développement est Java 6, alias Mustang, encore à l'état de *beta*-test.

[1] `http://setiathome.berkeley.edu/`
[2] `http://www.decrypthon.fr/`

Depuis 2002 et la sortie de la version Java 1.4, l'évolution de la plateforme et du langage est dirigée par le JCP (Java Community Process), en publiant des spécifications JSR (Java Specifications Requests).

De nombreux ouvrages sont basés sur cette plateforme et ce langage, nous ne citerons que ce livre de référence [DF02], regroupant un manuel de la plateforme et du langage, accompagné de la référence complète de l'API de la JDK (*Java Development Kit*). En tout, 5 éditions de ce livre existent au moment d'écrire ces lignes, couvrant chacune, une version spécifique de la plateforme.

Nous allons présenter les principaux atouts de la plateforme Java, à savoir les aspects portabilité et multi-plateforme, liés étroitement à l'utilisation d'un concept de JVM (Java Virtual Machine) et enfin, d'une API très riche et documentée.

1.2.1 Portabilité et multi-plateforme

La plateforme Java présente l'avantage d'être présente sur la majorité des systèmes (Windows, Linux, MacOS, Sun, ...), et également sous certaines formes sur les plateformes embarquées. Cela est rendu possible par l'utilisation du concept de *bytecode* (voir 1.2.1), d'une API correctement écrite (voir 1.2.1) et par l'utilisation d'une JVM (voir 1.2.2).

Concept de *bytecode*

Le concept de *bytecode* consiste à compiler le code dans un langage intermédiaire relativement proche du langage machine mais assez éloigné pour permettre d'y ajouter un interpréteur. Cet interpréteur sera la clé de voûte du portage de la plateforme Java : le code Java, une fois compilé en *bytecode* est analysé par une application convertissant les instructions du langage intermédiaire en langage machine. Le code ainsi compilé peut alors être exécuté, sans recompilation, sur une architecture différente !

Cela étend le concept du « *write once, execute everywhere* » au concept du « *compile once, execute everywhere* ».

API de base

La plateforme Java présente une API de base définissant les types de base. Seule cette partie est dépendante de l'architecture cible et n'est pas écrite en Java. C'est donc sur cette partie relativement compacte que se base la portabilité.

API très complète et documentée

Java est une plateforme de nouvelle génération (par rapport aux langages C/C++ et leurs bibliothèques). Il se devait de proposer directement à l'utilisateur tout un ensemble de classes et méthodes testées et approuvées permettant de programmer rapidement les différents types et concepts de base de l'algorithmique. C'est le pari que Sun a réalisé avec Java et sa JDK regroupant 212 classes réparties en 8 paquets dans la version 1.0 et 3562 classes regroupées dans 166 paquets dans la version 5.0.

J2EE, Services Web, RMI, etc. Accompagnant l'API de base, l'API de haut niveau a une particularité : contrairement aux couches basses présentées ci-dessus, elle est écrite entièrement en JAVA, permettant ainsi sa portabilité sur tous les systèmes à moindre frais. Elle regroupe des classes allant de

l'implémentation des types ensemblistes à la gestion de l'interface graphique (AWT puis Swing avec Java 1.2), en passant par la gestion des données XML, des appels de méthodes à distances Java RMI, ou encore la réflexion des fonctionnalités de base de la plateforme, la gestion des *threads*, l'écriture de services web, l'écriture d'applications J2EE, etc...

La JavaDoc Cette API est documentée au mieux et la version HTML de cette documentation permet de naviguer très rapidement entre les différentes hiérarchies de classes. Elle est réalisée par l'utilisation de l'outil JavaDoc, développé par Sun et fournit avec la plateforme de développement (JDK). L'outil JavaDoc repose sur une documentation du code en utilisant des balises *meta* de descriptions des paramètres d'une méthode, du type de retour de celle-ci, etc... Le code est analysé et la version HTML de la documentation est produite : le programmeur n'a plus à tenir de cahier de bord à jour, il peut directement écrire le code et la documentation associée, permettant par la suite de générer à nouveau ce cahier de bord.

1.2.2 La machine virtuelle Java

La machine virtuelle Java (JVM) est l'interpréteur de *bytecode* produit par le compilateur. C'est la seule partie de la plateforme Java à porter sur le système cible de déploiement. Sun en fournit des versions pour les architectures les plus courantes comme Windows, Linux et Solaris, mais il existe également des JVM pour les systèmes d'exploitation tel que MacOS, Windows CE, PalmOS, etc.

Cette JVM transforme le *bytecode* en instruction machine, impliquant une optimisation de celle-ci pour éviter la lourdeur d'une interprétation de code. Pour cela, elle utilise des algorithmes tel que « *Just In Time* » (*JIT*) ou encore « *HotSpot* ». Une des améliorations de JIT consiste à ne transformer en code exécutable que les morceaux de blocs de *bytecode* qui seront réellement exécutés : seules les méthodes réellement exécutées du code sont interprétées. Par contre, HotSpot analyse le code exécuté pour optimiser les éléments clés du code : par exemple, la conversion en code machine des parties de codes souvent répétées sont conservées en mémoire.

Enfin, une courte présentation de Java ne serait pas complète sans parler de la présence d'un « *garbage collector* » ou « *ramasse-miettes* » permettant au système de conserver une référence vers les zones de mémoires dont le programme utilisateur n'est plus capable d'accéder (écrasement de référence suite à une affectation, etc...) pour ensuite les libérer pendant les moments d'inactivités. Ou encore de la présence de « *Java Native Interface* » (JNI) permettant aux, programmes s'exécutant dans la machine virtuelle, d'accéder à l'interface native du système, comme certains pilotes par exemple, pour des raisons de performances ou de non compatibilités avec les interfaces fournies par Java, au prix tout de même, d'une incompatibilité, entre plateformes différentes, du code Java produit dans ce cas.

1.3 Apparition de XML

Le langage XML (*Extensible Markup Language*) est un langage balisé, issu d'une recommandation W3C (*World Wide Web Consortium*), ayant pour but d'encoder tout type de données, indépendamment du code machine de celle-ci. Il a été développé dans le but de partager facilement des données entres différents systèmes et en particulier à travers un réseau type Internet.

1.3.1 XML : un descendant de SGML

SGML (*Standard Generalized Markup Language*) est un métalangage avec lequel il est possible de définir des langages balisés. Il est lui-même descendant de la recherche sur la théorie des langages et en particulier du langage GML (*Generalized Markup Language*) d'IBM. Il est devenu par la suite un standard ISO.

XML est un sous-ensemble simplifié du langage SGML (*Standard Generalized Markup Language*) et est utilisé dans de nombreuses technologies ou protocoles utilisés très fréquemment aujourd'hui (RDF, RSS, XHTML, Atom, MathML, etc...). Tous les langages dérivés présentent le même avantage : encoder de façon structurée l'information destinée à être partagée à travers un parc d'ordinateurs et de logiciels hétérogènes.

En plus d'avoir donné naissance à XML, SGML est également à la base du langage de balisage HTML, un des éléments clés d'Internet. Par contre, son petit frère, XHTML, hérite lui de XML, d'où les similitudes entre ces différents langages.

1.3.2 Codage des données et interopérabilité

XML permet d'écrire sous forme de fichier texte, dont le codage est exprimé en en-tête, tous les types définis eux-mêmes par un autre fichier XML. Ainsi, grâce à ces deux fichiers (l'un contenant les données, l'autre de description des types de données), n'importe quel logiciel pouvant accéder à des fichiers texte peut analyser les données.

Ce type de technologie est une solution, très utilisée aujourd'hui, pour permettre l'interopérabilité des logiciels à travers Internet. Il est vrai que la syntaxe XML est très verbeuse, ce qui la rend difficilement lisible pour un humain : mais c'est le prix à payer pour une compatibilité maximale et une description au mieux des données pour leur traduction informatique.

Son succès est grandissant, et de nouvelles technologies apparaissent comme AJAX par exemple, une technologie web basée sur la JavaScript, le XML et le XHTML, permettant de travailler les données du côté client pour en envoyer une version complète au serveur sans faire de requête entre chaque petite modification.

1.3.3 Utilisation des fichiers XML

Plusieurs méthodes permettent d'exploiter les données d'un fichier XML. Des méthodes permettent d'accéder à la donnée précise en parcourant le chemin hiérarchique lié au balisage du fichier : ce sont les technologies tel que le langage XPath. Il permet d'exploiter le fichier sans devoir construire une représentation complète en mémoire du fichier. D'autres technologies consistent à construire un arbre représentant le fichier XML dans son entier, les noeuds de cet arbre étant les différentes balises XML. Cet arbre s'appelle l'arbre DOM (*Document Object Model*). Une telle utilisation est difficilement concevable pour accéder aux quantités de données stockées dans une base de données XML, mais est totalement viable pour analyser un fichier de description simple de service web, par exemple.

1.4 Synthèse

De part l'évolution de la puissance des ordinateurs et de leur mise en réseau, mais également de leur hétérogénéité, les technologies permettant l'interopérabilité des calculs et des échanges de données dans de telles circonstances ont pris une énorme importance. De plus, ces technologies ont

été appuyées (ou créées) par les grands constructeurs tel Sun, IBM, Microsoft. De même, elles ont été normalisées ou standardisées, étapes indispensables aujourd'hui (en passant par un organisme tel que le W3C). Tout ceci a poussé l'adoption par un grand nombre d'industriels et chercheurs de ces nouvelles technologies.

Nous allons nous focaliser dans la suite de cette thèse sur une partie de ces nouvelles technologies : les services web. Ces services web correspondent à la mouvance actuelle de la mise à disposition de tous des services développés, il y a peu, de façon totalement autonome ; mais également l'apparition de plateformes tel .Net de Microsoft, WebSphere d'IBM, etc... basées sur de tels concepts.

Chapitre 2

Présentation des services web

2.1 Introduction

En parallèle à l'évolution des outils basés sur Internet et à l'interopérabilité des données et logiciels, l'architecture des applications a fortement évolué. L'architecture est passée de la programmation modulaire à la programmation orientée objet, puis à l'architecture orientée composant et enfin à l'architecture orientée service.

2.1.1 L'interopérabilité

Au niveau industriel, l'évolution de la complexité des systèmes informatiques, de même que leurs coûts et leurs fonctionnalités se traduit par une évolution des architectures logiciels (et matériels), avec un impératif qui revient régulièrement : la réutilisation de l'existant. Ainsi, les industriels sont confrontés au problème de l'interopérabilité des données et des architectures à travers le temps. Il est effectivement impossible (impensable) de re-développer une application complète, dont le premier développement a pris plusieurs années, à chaque changement de machine (bien souvent accompagné par un changement de logiciels).

Cette évolution et prise de conscience de la réutilisation de l'existant est illustrée par les différentes architectures présentées par la suite : l'ajout d'une couche supplémentaire d'abstraction du logiciel métier (développé antérieurement) est très souvent utilisé. De plus, l'ajout d'une nouvelle couche procure plusieurs avantages : les applications et les données sont conservées et des modules peuvent y être ajoutés « facilement » pour permettre l'accès aux données à travers un autre poste en Intranet, ou même en Extranet voir Internet.

Ainsi, non seulement les employés de la société ont accès aux informations, mais également les commerciaux pendant leurs démarches à l'extérieur de la société et enfin les clients eux-mêmes peuvent surveiller l'évolution du stock des produits ou la préparation de leurs commandes. Toutes ces fonctionnalités nécessitent des outils pouvant communiquer entre eux, malgré leur utilisation dans une architecture système différente : des algorithmes de conversions entre les différents codages d'une même donnée sur des systèmes différents sont développés et utilisés régulièrement, d'où l'apparition de langages basés sur XML par exemple.

De même, les outils permettant de consulter les données peuvent être utilisés dans des environnements différents : machine de bureau, ordinateur portable, ordinateur de poche (Palm ou Pocket PC par exemple) ou encore téléphone portable. Les données sont non seulement codées de façon

différentes (suivant le type d'appareil), mais également non encore exploitées dans leur globalité : impossible d'avoir la base de produits dans son ensemble sur son téléphone portable, seule la partie la plus intéressante doit être utilisée dans ce cas.

2.1.2 Évolution de l'architecture de conception du/des logiciels

L'évolution de l'architecture est la conséquence de principaux facteurs tels que :
- la prise de conscience que la méthode précédente était imparfaite (souvent dûe à une évolution des besoins de l'utilisateur également) ;
- l'évolution de la capacité des machines (tant en mémoire qu'en puissance de calcul) se traduisant par l'évolution de leurs fonctionnalités ;
- mais également la complexité des systèmes informatiques et leurs coûts.

Nous allons étudier les trois évolutions majeures de l'architecture de programmation : la programmation modulaire, la programmation orientée objet et enfin la programmation orientée composant. La section suivante (section 2.2) présentera l'étape suivante : l'architecture orientée service.

Architecture basée sur la programmation modulaire

Fin des années 70, début des années 80, la programmation structurée ou modulaire était très à la mode, en même temps que l'arrivée des langages C (langage utilisé pour l'écriture des systèmes Unix), Pascal (langage développé dans un but principal d'enseignement) ou encore Ada (langage utilisé dans le domaine militaire). Le programmeur utilise des structures de données permettant de regrouper les informations ayant une relation entre elles, de plus il regroupe également le code traitant ces informations ou données dans un même module.

Ainsi, en reprenant l'ensemble des fonctions et des structures de données développées de façon autonome, les programmeurs espèrent ne pas avoir à tout re-devélopper à chaque étape du cycle de vie de l'application. Bien évidement, cela pose d'autres problèmes comme la séparation du code permettant la gestion des données et de l'interface utilisateur (affichage sur l'écran, stockage d'informations dans des fichiers de logs, etc.).

Les principales directives de la programmation modulaire sont :
- supprimer la programmation *spaghetti* reposant sur les instructions *goto* permettant de faire des sauts en avant ou en arrière dans le code. Ceci en privilégiant l'utilisation de boucles tels que *while*, *repeat ... until*, *for*, etc...
- découper le code en module cohérents (s'apparentant généralement à des fichiers) : un module traite un type de donnée abstrait comme une liste, ou un type « concret » comme les informations caractérisants une personne par exemple.
- éviter les variables globales que tout le code peut accéder à n'importe quel moment et modifier sans aucune vérification.
- utiliser des fonctions ou des procédures pour structurer le code : une fonction ou procédure pour un traitement bien précis sur un type de donnée bien précis. De plus, l'utilisation des fonctions permet de « *factoriser* » le code.
- utiliser la compilation sérarée : un module est compilable de façon autonome, et l'ensembles des fichiers compilés seront liés ensembles pour devenir un exécutable. Ainsi, en cas de modification d'un seul module, seul la compilation de ce module et la liaison finale est à exécuter à nouveau.

Même si les réseaux n'étaient pas aussi utilisés qu'aujourd'hui, le concept d'utilisation et d'invocation à distance, dans le cadre d'un système réparti était déjà présent. Les programmeurs utilisent, dans le cadre de la programmation modulaire, les mécanismes *Remote Procedure Call* (RPC, [Whi75]).

Le mécanisme RPC permet de développer une application basée sur le modèle client-serveur. Un appel à distance est effectué par le client en envoyant un message à un système distant (serveur), pour exécuter une procédure donnée en utilisant les arguments fournis. Le résultat (calculé par le serveur) est ensuite envoyé au client.

De nombreuses implémentations sont nées de cette norme, et beaucoup sont incompatibles entre elles. Mais le principe était là : le client et le serveur pouvaient utiliser un encodage des données différents, un *proxy* réalise une conversion juste après l'appel de la fonction par le client, juste avant l'envoi effectif au serveur ou lors de la réception du message de réponse.

Architecture basée sur la programmation orienté objet

Parallèlement à l'arrivée de la programmation structurée, la programmation objet est arrivée début des années 80. La programmation objet reprenait le concept du regroupement des données et des fonctions (appelées méthodes en terminologie objet), pour obtenir un seul module : une *classe*, qui, une fois instanciée, devient un objet.

Deux avantages principaux à cette architecture :

- il existe une notion de regroupement du code et des données dans un seul et même objet, et ce, en forçant, si nécessaire leur non accessibilité depuis l'extérieur de cet objet : cela permet, par exemple, d'empêcher la modification d'une variable par du code n'appartenant pas à l'objet. Ce principe renforce le principe de séparation modulaire, de compilation séparée et de portée des variables dans les procédures et fonctions issue de la programmation modulaire.
- il existe un système d'héritage permettant de raffiner le comportement d'une classe : par exemple, une personne peut être un étudiant, un professeur ou un stagiaire. Cet héritage fournit des mécanismes tel que le polymorphisme et l'encapsulation.

SmallTalk et Eiffel font partis des langages objets cités, le plus souvent, comme étant les plus fidèles du principe objet. Mais suivant les langages, des notions différentes peuvent apparaître. Par exemple, le C++ utilise le concept d'héritage multiple alors que le Java ne permet que de l'héritage simple.

Une classe fournit donc un ensemble complet de méthodes permettant de traiter des données. Le programmeur peut alors :

- instancier un nouvel objet (qui fera appel aux constructeurs de la classe pour l'initialiser),
- accéder aux méthodes publiques (ou privées à l'intérieur d'un objet) pour obtenir ou modifier les propriétés de l'objet
- utiliser un mécanisme de destruction de l'objet (dont le comportement varie fortement d'un langage objet à un autre).

Au niveau des mécanismes de répartition, on parle alors d'objets répartis. Ce sont des objets de « *base* » qui évoluent dans une architecture répartie, bien souvent à base de bus (*Object Request Broker* ou ORB dans le cadre de CORBA [GGM99] et Java RMI [Gro01] dans le cas de Java). Ce bus est une partie logiciel faisant partie d'un *intergiciel* (ou *middleware*), visant à s'affranchir des problèmes liés à la répartition, en gérant l'interopérabilité ou encore la portabilité des objets.

Pour permettre la portabilité du composant, la description de la partie dépendante de la plateforme est faite dans un langage portable, et un générateur de code produit du code spécifique pour une

plateforme cible : ce code servira de passerelle entre les clients et l'application métier (mécanisme RMI de Java par exemple).

Architecture orientée composant

L'architecture orienté objet et ses objectifs initiaux ne permettent pas d'utiliser la puissance et les nouvelles méthodes issues de l'évolution des architectures (grilles de calcul, l'utilisation du pair à pair, architecture 3-tiers ou même n-tiers) tendant toujours vers plus d'uniformisation tout en augmentant la complexité des modèles, et ce dans le but de séparer la *logique métier* et la *logique système*. Une nouvelle architecture est apparue progressivement répondant à ces besoins : l'architecture orientée composant [PB05].

Cette nouvelle architecture tend à proposer des solutions aux problèmes courant rencontrés dans les systèmes réparties qui sont :

- la possibilité d'utiliser à *large échelle* les solutions proposées, par exemple à travers Internet,
- la gestion de l'*hétérogénéité* des systèmes,
- la prise en compte des *communications asynchrones* et la gestion du *contrôle concurrent*,
- la prise en compte des *pannes partielles* concernant le logiciel, le réseau (surcharge, coupure temporaire), le système, etc.
- et enfin, un élément indispensable : la gestion *sécurité*.

La prise en compte de ces problèmes aboutit à des solutions basées sur les bus à objets vu précédemment. Elles sont généralement englobées dans un *framework*, c'est-à-dire une boîte à outils permettant de « tout » faire. Ces frameworks peuvent être à la fois visuels (par exemple, un IDE de type RAD *–développement rapide d'applications–*) ou *abstraits* gérant le déploiement à travers un réseau des différents composants, pouvant même gérer, dans certains cas, les différentes versions de ceux-ci.

Ainsi, l'*OMG* (*Object Management Group*) propose le modèle de composant *CCM* (*CORBA Component Model*) reposant sur CORBA, Sun propose un modèle équivalent basé sur le bus d'objets Java RMI : les *EJB* (*Enterprise JavaBeans Technology*), Microsoft, quand à eux, propose une nouvelle technologie : le framework *.Net*, reposant sur un modèle complètement composant complètement nouveau (abandonnant le système précédent COM+ qui n'était pas complètement objet).

Cette nouvelle architecture orientée composant prend en compte les nouvelles architectures système comme le *P2P* (*peer to peer*) permettant de considérer chaque « acteur » du système sur un pied d'égalité, contrairement au modèle client-serveur ou n-tiers ; ou encore l'architecture basée sur les grilles de calcul, permettant de répartir les calculs et donc les composants sur un grand nombre de noeuds ou processeurs.

Une autre extension possible des intergiciels est, comme dans PolyORB [HPK03], l'utilisation de personnalités permettant alors à l'intergiciel de s'adapter rapidement aux besoins d'une application en déployant l'intergiciel et les services nécessaires. En effet, il n'existe pas, ou rarement, d'intergiciel répondant parfaitement aux besoins d'une application donnée, donc disposer de plusieurs possibilités est un atout majeur dans le développement rapide d'application.

2.2 De l'architecture Orientée Objet à l'architecture Orientée Service

Les différentes architectures vues précédemment ont évolué, et ce, particulièrement car les besoins ont, encore une fois, changé et la puissance des machines également. De même, l'arrivée du réseau Internet sur toutes les machines favorise l'utilisation de protocoles de communication solutionnant les problèmes d'interopérabilités entres les différentes applications. Ainsi, avoir une base de données locale au niveau d'un poste et consultable uniquement sur ce poste n'est plus envisageable : les informations doivent être consultables depuis n'importe quel ordinateur à l'intérieur, voir à l'extérieur de la société. Toutes ces raisons ont favorisé l'évolution de l'architecture orientée objet et/ou composant en architecture orientée service.

2.2.1 Acquis de l'objet

Plusieurs aspects présents dans les architectures basées sur les objets ou les composants ont été repris dans l'architecture orientée service. Parmi ceux-ci, une notion très importante est celle de la réutilisation : le code doit être écrit au mieux et sa documentation la plus complète possible. De même, ce code, pouvant lui-même évoluer, ne doit pas, lors d'une évolution, entraîner la réécriture d'autres programmes l'utilisant.

Une *API* (*Application Programming Interface*) est donc indispensable : cette API comprend un jeu de fonctions ou méthodes sensées être stables dans le temps (au moins lors des révisions mineures) permettant ainsi de changer soit le code de la bibliothèque et alors le programme ne doit pas être impacté, soit de changer le code du programme l'utilisant, et alors la bibliothèque ne doit pas être modifiée. De même cette API doit être clairement documentée ; par exemple, le langage Java propose le mécanisme de JavaDoc permettant de documenter chaque classe ou méthode et de générer une documentation au format HTML, « indépendante » du code.

Un autre aspect repris des architectures orientées objet est le découpage de l'interface (utilisateur, mais pas forcément humain) et du traitement des données. En effet, les composants, comme nous venons de le voir, doivent être réutilisables. Il est donc indispensable d'avoir une partie dédiée à l'interfaçage des données et une autre partie dédiée au traitement de l'information, ces deux parties devant être absolument distinctes et communiquant par le biais d'une API stable, prévue au plus tôt dans la spécification. La partie interfaçage des données regroupant les aspects :

- affichage et saisie des informations correspondant à l'interface homme-machine,
- mais également une interface au sens serveur réseau, une API, etc... permettant à d'autres logiciels/composants d'appeler des fonctions permettant la saisie/consultation de données)

Dans les aspects liés à la répartition, on peut citer l'utilisation d'intergiciels et plus exactement la possibilité d'invoquer depuis n'importe quel client un serveur donné. C'est-à-dire qu'un client n'ayant pas la même architecture logiciel et matériel doit être capable de communiquer avec un serveur donné. Cela est encore plus vrai dans l'architecture orientée service que nous allons décrire, car les clients peuvent être des ordinateurs de bureaux, mais également des terminaux mobiles, comme des téléphones, des assistants personnels, etc. qui reposent rarement sur la même architecture.

2.2.2 Limites de l'objet

Une des principales faiblesses du concept orienté objet est le manque de communication entre les différents intergiciels. En effet, à la base, il n'existe pas de norme permettant d'unifier les différents bus d'objets par exemple. Cela, principalement pour plusieurs raisons :

– au moment du développement de ces intergiciels, ils étaient destinés à une utilisation à travers un Intranet, et très peu vers l'Extranet/Internet, vecteur principal du mixage des plateformes.
– le développement d'un intergiciel étant un enjeu industriel, chaque éditeur veut imposer son format, bien évidement, incompatible avec les autres formats.

Par des transformations adéquates de modèles, et par un développement *a posteriori*, on peut effectivement parvenir à l'écriture de module permettant de rendre interopérables les intergiciels. Mais ce concept n'est arrivé que plus tard.

De même, il n'existe pas d'annuaire global regroupant les ressources déployées à travers les intergiciels. Cela découle effectivement du manque d'interopérabilité entre ceux-ci, mais même en local, tous ne possèdent pas de mécanisme de recherche suivant certaines propriétés (mots-clés, etc...).

Enfin, tous les administrateurs systèmes et réseaux considèrent l'installation de ces intergiciels comme long et fastidieux : facteur réel, dû principalement au nombreux modules à configurer et à déployer sur des parcs de machines hétérogènes. L'architecture orientée service, évolution de l'architecture orientée composant, essaye donc de minimiser ces étapes d'installations.

2.2.3 Architecture orientée services

Le passage à l'architecture orientée services (*SOA - Service Oriented Architecture*) se fait sur le long terme. La conservation de l'existant et la migration petit à petit est donc indispensable. La méthode la plus adaptée est l'encapsulation de l'existant (existant reposant généralement sur Corba, J2EE, DCOM, etc.) en respectant les nouveaux standards des publications. Ainsi, les applications métiers développées jusque là continuent de fonctionner, et les données et leurs traitements sont disponibles sous une forme publiée à travers un réseau et un protocole de communication standardisé.

Principes de SOA

Le principe de base de SOA est de publier (sur Internet, un Extranet ou encore un Intranet) un ensemble d'offres (existant ou non) réalisant un service bien précis (par exemple un service pour crypter des données, un autre pour authentifier une personne, etc...). Une application dans ce cadre appelle ces différents services quand elle en a besoin et peut être elle-même déployée comme un nouveau service (remplissant alors un plus grand rôle dans l'architecture métier).

Pour rendre interopérable les différentes plateformes basées sur l'architecture orientée service, il faut principalement :

– utiliser une API standard et normalisée de communication, aussi bien au niveau protocole de communication que de la mise à disposition des informations sur le réseau.
– coder les données de manières compatibles. Bien souvent les nouvelles architectures utilisent aujourd'hui le méta-langage XML pour coder leurs informations : il existe de nombreux analyseurs XML, disponibles pour un grand nombre de langages et de plateformes, permettant ainsi le choix de cette technologie sans risquer de se retrouver incompatible.

L'utilisation de la modélisation

La mise en place d'une architecture orientée service s'accompagne généralement d'un développement utilisant la modélisation. Les avantages de décrire son application par un langage type UML sont nombreux :

- *obtention d'un modèle indépendant de la plateforme* : le modèle indépendant de la plateforme (*PIM - Plateforme Independant Model*) représente la logique métier, les données et les exigences fonctionnelles de chaque application.
- *génération automatique de code à partir du modèle* : à partir du modèle indépendant, des outils peuvent générer du code dépendant d'une architecture cible (plus ou moins raffiné suivant le niveau de raffinement du modèle).
- *simplification de la migration de l'application* : la migration de l'application d'une plateforme à une autre permet de conserver la logique contenue dans le modèle ; tout n'est donc plus à réécrire.

On parle alors de MDA (*Model Driven Architecture*, [Ger02]) qui définit également des méta-modèles permettant la transformation de modèle. Le MOF (*MetaObject Facility*), standard OMG (*Object Management Group*), définit un ensemble de règles permettant de faire la correspondance entre les différentes structures utilisées par les méta-modèles.

En appliquant ces différents concepts à la modélisation du *workflow* (modélisation de l'enchaînement des différentes étapes de traitements de l'information), on utilise une implémentation concrète du SOA : les services web que nous allons détailler par la suite.

2.3 Les services web : une instance de SOA

Le SOA repose sur différents concepts que nous venons de présenter. La principale implémentation de ces concepts repose sur les services web [MG04]. La suite de cette section présente les services web dit « basiques » : leur définition, les protocoles utilisés pour mettre en oeuvre une telle architecture, etc. Ensuite, l'aspect composition des services web sera abordé. Enfin, deux langages de description de services web complexes seront présentés.

2.3.1 Concrétiser les éléments génériques du SOA

Le W3C (*World Wide Web Consortium*[1]), créé en 1994, est l'un des principaux organismes international de normalisation des technologies et protocoles basées sur Internet, permettant d'assurer le développement et la compatibilité de la toile dans le futur.

Nous adoptons donc ici la définition d'un *service web* du W3C [CFN+02].

Définition des services web

Définition 2.3.1 Un service web est un logiciel identifié par une URI (Uniform Resource Identifier, [BLFIM98]), dont l'interface est publique et les liaisons sont définies et décrites en utilisant XML. L'environnement du service offre le moyen à d'autres logiciels de découvrir celui-ci. Ce service interagit avec les autres services web en respectant cette définition, donc en utilisant des messages basés sur XML acheminés par des protocoles Internet.

Cette définition ne présuppose ni l'utilisation du protocole *SOAP* (*Simple Object Access Protocol*, [Mit02]) pour assurer le transport et la communication entre les entités, ni une description du service web par le langage de description *WSDL* (*Web Services Description Language*, [CCMW01]).

[1] `http://www.w3.org/`

Mais l'architecture de référence d'un service web suppose que ce service possède un niveau d'abstraction dans lequel il est décrit par le langage de description WSDL et qu'il emploie le protocole de communication SOAP.

Description du modèle fonctionnel des services web

Les services web s'articulent autour du protocole d'échange de données SOAP (lui même basé sur XML) [KM03]. Ce protocole se situe dans une couche supérieure des protocoles de niveaux applicatifs de l'Internet comme HTTP, FTP, SMTP, etc. Ces derniers encapsulent donc les messages XML issues du protocole SOAP dans leurs propres messages.

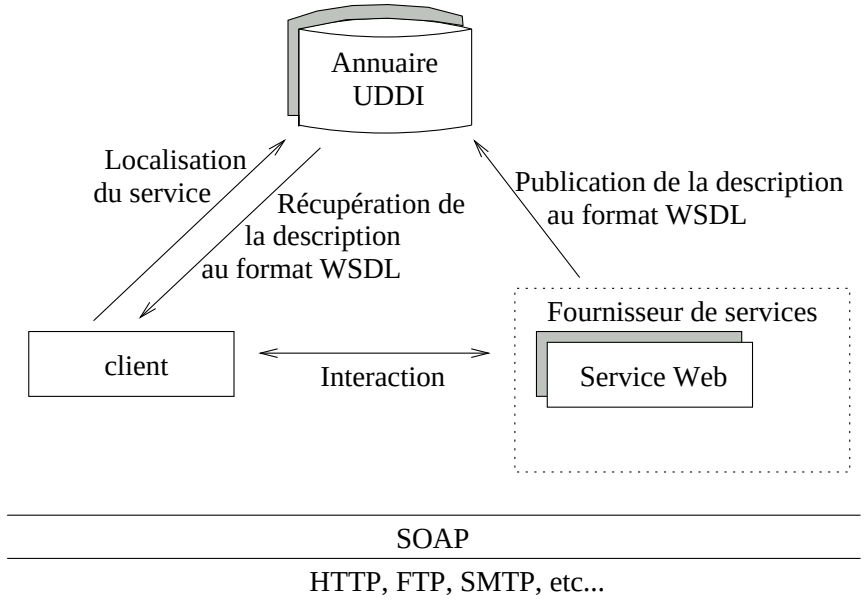


FIG. 2.1 – Modèle fonctionnel de l'architecture de publication et d'invocation d'un service web.

Comme il est possible de le voir sur le modèle fonctionnel de la figure 2.1, le *fournisseur de services* est un serveur exécutant des applications ou composants assimilables à des services web. Leur interface est alors décrite en WSDL.

Pour se faire connaître, le fournisseur de services publie la description WSDL des services qu'il fournit sur un ou plusieurs *annuaires de services UDDI* (*Universal Description, Discovery and Integration*, [TBLCDE+02]). Cet annuaire peut être local à une application, à un réseau d'entreprise ou à l'échelle d'Internet.

Ainsi, lorsque le *demandeur de services* (c'est-à-dire le client) veut savoir quels sont les serveurs fournissant un service correspondant à une certaine description, il fait appel à un annuaire UDDI dont il a la connaissance. Ce dernier lui renvoie alors le ou les fichiers WSDL des services web correspondant à la demande.

Le client fait alors son choix, s'adresse au fournisseur et invoque le service web, dont il n'avait pas nécessairement connaissance auparavant.

2.3.2 Les langages et protocoles utilisées par les services web

La description fonctionnelle précédente montre l'utilisation de nombreux langages et protocoles durant le déploiement et l'invocation du service web. Ce sont ces principaux langages et protocoles que cette section va décrire.

Le protocole SOAP

Les communications entre les différentes entités impliquées dans le dialogue avec le service web se font par l'intermédiaire du protocole SOAP (*Simple Object Access Protocol*). Ce protocole est normalisé par le W3C [Mit02, GHM+02a, GHM+02b]. La description qui va suivre est basée sur la version 1.2 de ce protocole.

Le protocole SOAP est une surcouche de la couche application du modèle OSI des réseaux. Le protocole applicatif le plus utilisé pour transmettre les messages SOAP est HTTP, mais il est également possible d'utiliser les protocoles SMTP ou FTP : la norme n'impose pas de choix.

Le choix de l'utilisation de protocoles applicatifs, comme par exemple HTTP, est lié aux problèmes d'interconnexion connus des réseaux : le but des services web étant d'être réutilisables, après publication, à travers tout le réseau Internet, il faut alors donner les moyens de passer les protections tel que les pare-feux (*firewalls*). Ces derniers autorisent généralement sans aucune restriction le trafic sur les ports liés aux protocoles tel que HTTP, permettant ainsi le passage sans problème à travers les différents réseaux des messages générés par l'utilisation du protocole SOAP.

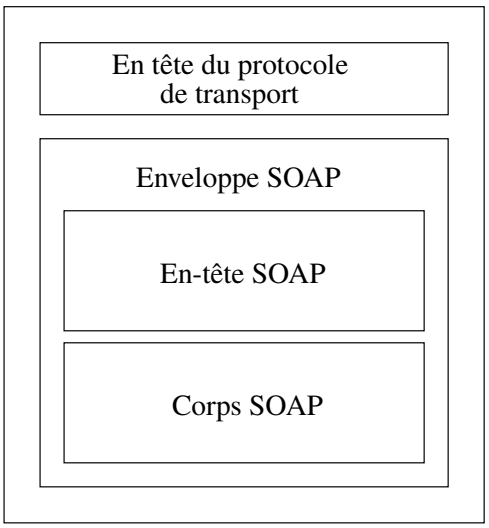


FIG. 2.2 – Schéma d'un message SOAP

Dans un soucis d'interopérabilité, et donc dans la continuité des différents langages et protocoles issus des services web, les messages échangés lors de l'utilisation du protocole SOAP sont basés sur le méta-langage XML. Ces messages comportent plusieurs parties (voir la figure 2.2) que nous allons décrire et sont donc encapsulés dans des protocoles de niveau applicatif.

L'en-tête du protocole de transport Cette partie dépend du protocole de transport utilisé. Par exemple, si le protocole HTTP est utilisé, cette en-tête contient :
- la version de HTTP utilisée,
- la date de génération de la page (qui est ici le message SOAP lui-même),
- le type d'encodage du contenu (ici, l'encodage est généralement le type *text/xml*), etc.

Les messages SOAP (l'enveloppe) La partie principale d'un message SOAP est l'*enveloppe* (symbolisée par la balise `envelope`). Cette enveloppe (*SOAP Envelope*) est elle même subdivisée en deux sous-parties : la partie en-tête et la partie corps du message. Elle permet également de spécifier des environnements de noms XML utilisés dans la suite du message.

L'en-tête du message SOAP L'en-tête SOAP (*SOAP Header*) est optionnelle et extensible.

Les balises XML permettant de symboliser cette partie sont `<env:Header>` et `</env:Header>`. Ces balises peuvent être complétées par des attributs permettant de définir le domaine de noms du service web.

En fait, l'en-tête permet principalement d'ajouter des informations sur le comportement que doivent avoir les différents *noeuds* intermédiaires, lors du traitement du message.

Un *noeud* étant un intermédiaire SOAP, incluant le récepteur et l'émetteur SOAP, désignable depuis un message SOAP. Son rôle est de traiter l'en-tête (et d'effectuer les actions qui y sont décrites) et ensuite de transférer le résultat (le message SOAP modifié) à un autre intermédiaire (qui peut être le récepteur final).

Par extension, la description du comportement des différents noeuds permet également de réaliser une "composition de services", car le message peut être routé entre différents noeuds, chacun étant capable de réaliser une action précise et décrite dans le bloc d'en-tête.

Les principaux attributs des éléments formant le bloc sont :
- `env:role` : permet d'indiquer à quel noeud la fonction décrite est destinée. Et donc par extension, cela permet le routage d'un message.
- `env:mustUnderstand` : c'est une valeur booléenne, elle permet de préciser que le traitement devient obligatoire pour un noeud intermédiaire. Par exemple, pour un calcul très long, il peut être utile d'envoyer un e-mail à chaque étape.
- `env:relay` : cet attribut permet de relayer un message à un autre noeud si le premier noeud n'est pas capable de le traiter.

Le corps du message SOAP Les données spécifiques à l'application se trouvent dans le corps du message SOAP (*SOAP Body*). Tout ce qui est présent dans cette section est défini lors de la conception de l'application. Enfin, les balises symbolisant cette partie sont `<env:Body>` et `</env:Body>`.

Les données doivent donc être sérialisées selon l'encodage choisi. L'utilisation du XML 1.0 à l'intérieur des blocs XML `<element>` permet d'envoyer absolument tous les types de documents comme par exemple des feuilles de styles, des documents XML, des images au format binaire, etc...

En plus des données, cette partie peut transporter un type spécial : les messages d'erreurs (*SOAP Fault*).

Comme précédemment on peut ajouter un attribut dans la balise `<env:Fault>` permettant d'indiquer un type d'encodage des données, mais également d'autres attributs permettant la gestion de l'erreur, comme `Code`, `Reason`, `Node`, `Role` et `Detail`.

Le protocole UDDI

UDDI (*Universal Description, Discovery and Integration*, [TBLCDE+02]) est plus qu'un simple protocole : il fournit un protocole, une API et une plateforme permettant aux utilisateurs de services web, depuis n'importe quel système, de localiser dynamiquement à travers Internet les services web qu'ils désirent utiliser. Ceci passe par le biais d'annuaires qui peuvent être maintenus dans plusieurs optiques :

- par une seule et même personne pour son usage interne ;
- par un groupe d'utilisateurs regroupant des services web répondant à certains critères ;
- par un organisme comme base de données mondiale de services web.

UDDI, dans une optique d'interopérabilité, est basé sur les standards tel que HTTP, XML, XML Schema, SOAP. Ainsi, la version 3 de ce protocole, normalisée par le *consortium Oasis*[2], considère UDDI comme un méta-service de localisation de services web.

Les différents rôles d'UDDI UDDI fournit trois services de bases :

- **Publish** : ce service gère comment le fournisseur de service web s'enregistre lui-même ainsi que ses services (en utilisant UDDI).
- **Find** : ce service gère comment un client peut localiser le service web désiré (cela peut passer par des invocations de services web pour une utilisation automatique par un programme ou par une consultation d'annuaire en utilisant des mots clés).
- **Bind** : ce service gère également comment un client peut se connecter et utiliser le service web une fois celui-ci localisé.

Les entreprises qui fournissent ce service et hébergent un annuaire global UDDI sont appelées des *opérateurs UDDI*. Ils sont responsables de la synchronisation de l'information des annuaires : cette synchronisation d'annuaires s'appelle la *réplication*.

Il existe de nombreux annuaires UDDI dont les principaux sont :

- celui de Microsoft : `http://uddi.microsoft.com` ;
- celui d'IBM : `http://uddi.ibm.com`.

Un des enjeux de UDDI est d'éviter le monopole d'une entreprise qui, par un annuaire quelconque, donnerait comme réponse systématiquement ses services web plutôt que celle des autres : ce genre de pratique étant fortement limité car les annuaires UDDI se doivent d'avoir un contenu identique aux autres annuaires. Ainsi, il permet de donner des solutions industriellement viables pour la localisation de services web.

Les structures de données UDDI UDDI est un modèle d'information composé de structures de données persistantes appelées entités. Ces entités doivent être décrites en XML et sont stockées dans les différents noeuds UDDI.

Les différentes informations sont divisées en trois catégories :

- **pages blanches (*White Pages*)** : elles regroupent les informations sur les noms des entreprises publiant leurs services web, les moyens de les contacter, etc.
- **pages jaunes (*Yellow Pages*)** : elles regroupent les informations à propos de la classification des entreprises.

[2] `http://www.oasis-open.org/`

– **pages vertes (*Green Pages*)** : elles contiennent des informations techniques comme les services offerts par une entreprise, leur spécification, etc.

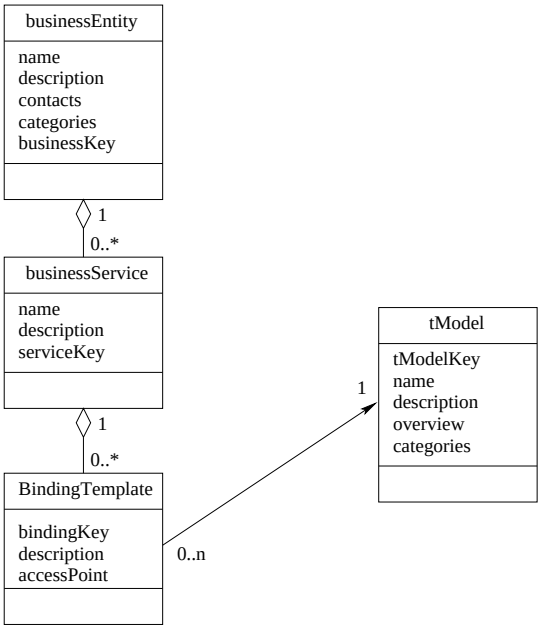


FIG. 2.3 – Les structures de données de UDDI.

Le modèle d'information UDDI (version 2) est composé de quatre types d'entités (voir figure 2.3) :

– **businessEntity :** cette entité décrit l'entreprise (également appelée fournisseur) qui permet d'accéder aux services web qu'elle publie : cette entité permet de regrouper toutes les informations concernant le nom, les contacts de l'entreprise. Chaque enregistrement est associé à une clé unique appelée *UUID* (*Universal Unique Identifier*). Ce sont les éléments accessibles par l'annuaire *pages blanches*.
– **businessService :** cette entité représente une classification des services. Elle est contenue dans l'entité *businessEntity* décrite précédemment, et elle contient une clé unique identifiant un service particulier. Ce sont les éléments accessibles par l'annuaire *pages jaunes*.
– **bindingTemplate :** cette entité est utilisée pour les détails techniques des services web. Elle contient des informations sur le point d'accès du service (l'adresse du service). Ce sont les éléments accessibles par l'annuaire *pages vertes*.
– **tModel :** cette entité permet de stocker les informations spécifiques aux services, comme le comportement, les conventions de typages et les types eux-mêmes utilisés par les services. Elle regroupe donc les informations contenues dans les fichiers WSDL.

Le langage WSDL

Le langage de description WSDL (*Web Services Description Language*, [CCMW01]) a été créé dans le but de fournir une description unifiée des services web. Il se présente comme un standard actuel dans ce domaine, et il est, de plus, normalisé par le W3C. Il est issu d'une fusion des langages de descriptions NASSL (*Network Acessibility Service Specification Language - IBM*) et SCL (*Service*

Conversation Langage - Microsoft). Son objectif principal est de séparer la description abstraite du service de son implémentation même.

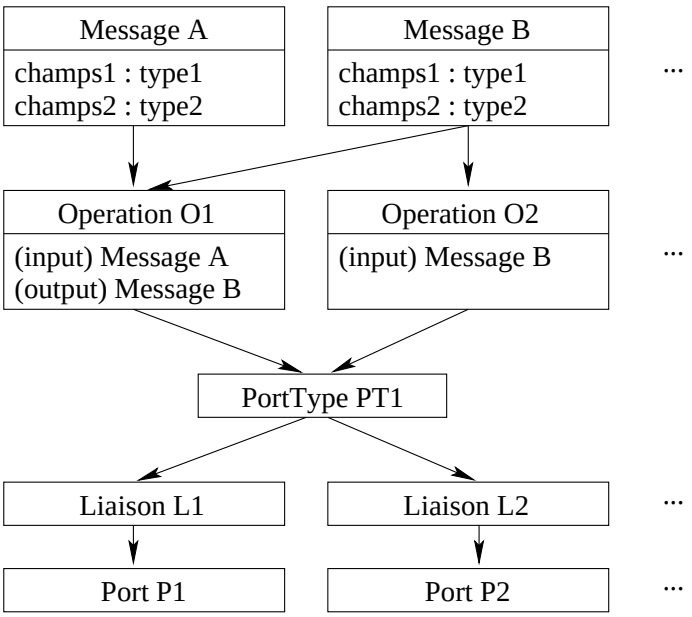


FIG. 2.4 – L'architecture d'un fichier WSDL.

Le langage de description WSDL se comporte donc comme un langage permettant de décrire l'interface visible (ou publiée) du service web. Il décrit, à l'aide du langage de balises XML, les différents éléments du service (voir figure 2.4) :

- les messages (intervenant lors des échanges) et leurs types associés (pour gérer l'interopérabilité entre les différents intervenants de l'échange)
- les opérations (composées de un ou plusieurs messages)
- les liaisons et les ports de communication (permettant de lier les opérations à un protocole de transport sous-jacent)
- la description du service lui-même.

Les messages. Ils sont composés de plusieurs parties. Ces parties correspondent, par exemple, dans le paradigm objet aux différents champs d'une structure. Ils sont décrits en XML et un type leur est associé. Les types de base XML peuvent être utilisés (entier, chaîne de caractères, etc.), mais également des types complexes définis dans un fichier WSDL (fichier identique à celui de la description du service ou externe).

Les opérations. Cette partie associe les messages aux opérations (une opération peut être vue comme une méthode). Les différents messages étant les différents paramètres de cette méthode. Un mot clé permet de distinguer le mode :

- *input* : mode entrée ou paramètre de donnée
- *output* : mode sortie ou paramètre de résultat

Plusieurs opérations sont associées pour former un *PortType*, utilisé par la suite.

Les liaisons. Elles permettent d'associer les opérations (ou un ensemble d'opérations identifiés par un *PortType*) au protocole de transport de niveau inférieur. Ainsi, dans le cas d'un service web utilisant le protocole de communication SOAP :

- pour chaque opération, on définit le type d'échange utilisé (mode *document* ou mode *XML-RPC*).
- pour chaque message (composant les opérations), on décrit l'espace de nom associé au type de message à transporter.

Le service. Enfin, le service lui-même est décrit dans la partie *service* de cette description WSDL. Le mot clé *port* permet d'associer des adresses Internet (URL) à chaque *PortType* : ces adresses seront utilisées par le client pour l'invocation du service. Il est également possible d'ajouter des informations permettant de trouver une documentation pour le service. C'est également cette partie qui peut être étendue par un autre langage que WSDL.

2.3.3 Langages de services web complexes

Les limites de WSDL

Indispensable pour connaître les informations essentielles à l'interaction entre un client et un service, la description WSDL montre vite ses limites. En effet, elle représente une description de l'interface d'un service, et modélise une interaction sans mémoire.

Uniquement un langage d'interface La seule chose que connaît un utilisateur d'un service est donc cette description WSDL. On y trouve toutes les informations décrivant ses différentes opérations : nom des opérations, les messages à envoyer, les message à réceptionner et les types des différents messages. Mais, aucune description n'est faite concernant le comportement même du service, qui permettrait de répondre aux questions :

- doit-on appeler l'opération "*authentification*" avant l'opération "*effectuer un virement*" ?
- peut-on utiliser indifféremment l'opération "*annuler*" et l'opération "*terminer*" ?

Bien évidement, on peut trouver une solution à ce problème en adoptant un système de nommage évitant les ambiguïtés, mais ceci ne fonctionne que dans le cadre d'une utilisation par un programmeur de cette description.

Donc, pour permettre la réutilisation des services dans le cade d'un processus métier composable, des informations supplémentaires sont indispensables ! Encore plus dans le cadre d'une composition automatique (ou semi-automatique) de services.

Interaction sans mémoire De même, une fois le problème précédent énoncé, la gestion de l'état côté serveur n'est pas présente : en effet, à aucun moment, le service n'explique s'il conserve les informations issues d'une précédente opération, ou s'il a besoin de faire une comparaison entre deux valeurs. On se retrouve donc bien dans un modèle d'interaction sans mémoire.

Cette gestion de l'état du côté service peut être utile à plusieurs reprises : trouver le prix minimum d'une réservation lors de l'appel à deux services différents par un service que l'on qualifiera de "maître", etc.

Le langage de description WSDL, présenté comme la solution ultime pour les services web dans le cadre de la réutilisation de composants à travers un réseau ou Internet, et dans un milieu hétérogène,

présente donc quelques inconvénients. Ces inconvénients ont été soulevés de nombreuses fois dans la littérature et certaines des solutions proposées vont être présentées dans la suite de ce chapitre.

Parallèlement à ces problèmes, il faut bien souligner que pour des services web basiques permettant par exemple d'obtenir la température d'une ville donnée (par son code postale), la description WSDL est tout à fait suffisante. Les problèmes n'apparaissent que dans le cas de services web dit *complexes*, c'est-à-dire utilisant des interactions entre opérations et/ou la mémoire.

Langages de programmation ou de composition

Les langages de programmation ou de composition de services web permettent de palier le manque d'information lié à l'utilisation du langage de description WSDL. Les services web utilisant ces langages sont alors appelés *services web complexes*.

Définition d'un service web complexe L'utilisation (et par extension la réutilisation) de plusieurs services web par un même service web permet de définir des services web modélisant un processus métier dans sa globalité plutôt qu'une de ses opérations. Par la suite, nous appellerons ces processus métiers (représentés par un ou des services web) des services web dits "*complexes*". Ces services web complexes sont à contraster avec les services web dits "*basiques*" vu précédemment (section 2.3.1).

Les langages de descriptions (et donc de programmation ou de composition) de ces services web complexes font apparaître plusieurs niveaux de descriptions : *niveau du processus abstrait* et *niveau du processus exécutable*, mais également plusieurs méthodes d'exécutions : *orchestration* (centralisé) ou *chorégraphie*.

Orchestration ou chorégraphie ? Les différents processus, intervenant dans une composition suite à l'utilisation de services web complexes, peuvent être composés et exécutés par deux méthodes différentes : en *orchestration* ou en *chorégraphie*.

Définition 2.3.2 Orchestration : un processus principal (service web) prend le contrôle du déroulement de la composition et coordonne donc les différentes opérations des différents services web. À aucun moment les autres services servant à la composition n'ont connaissance de cette composition : ils remplissent leur rôle de service sans se soucier si un client humain ou applicatif interagit avec eux.

L'orchestration est une méthode très utilisée, et pour plusieurs raisons :

1. il est relativement simple d'écrire un processus centralisé gérant l'invocation de sous-services ;
2. dans le cas d'une réutilisation de services web basiques, seule la partie centrale est à développer.

Définition 2.3.3 Chorégraphie : la chorégraphie ne repose pas sur un service web principal. Chacun des services intervenant dans la composition sait exactement ce qu'il doit faire, quand il doit le faire et avec qui. Donc ils ont tous une connaissance plus ou moins globale du processus métier dans lequel ils se retrouvent.

Contrairement à la méthode basée sur l'orchestration, la chorégraphie demande nettement plus de développement et de test : il faut développer chaque service dans le but de la composition qu'ils formeront. Cela dit, en utilisant des méthodes de développement et des stratégies bien pensées, l'intérêt de la méthode apparaît : la non-centralisation du traitement.

L'approche la plus simple dans le passage à SOA est l'*orchestration*. En effet, le but principal de la composition est la réutilisation de services (basiques ou non) sans modifier ceux-ci (car ils peuvent

très bien être hébergés par une autre compagnie et nous n'avons alors aucun moyen de contrôle sur ceux-ci). Le processus principal doit alors pouvoir s'adapter aux différentes erreurs possibles (non disponibilité d'un service, annulation, etc.).

SOA permet également d'utiliser l'approche *chorégraphie* : la preuve en est de l'existence de nombreux standards comme WSCI (*Web Services Choreography Interface* [AAF+02]), ou encore WS-CDL (*Web Services Choreography Description Language* [KBR+05]).

Processus abstraits ou exécutables ? Les langages de programmation ou de composition représentent à la fois des processus abstraits et des processus exécutables [Jur05, JSM05].

Définition 2.3.4 Processus abstraits : Les processus abstraits spécifient des échanges de messages publiques entre les différentes parties. La description n'inclut alors pas les détails des flux des différents échanges provenant des différents partenaires et le processus ainsi décrit n'est pas exécutable. On se focalise sur la vue protocolaire permettant de réaliser une abstraction du processus métier.

La description de la partie abstraite de ces processus donne les informations concernant les échanges avec le monde extérieur : aucune information n'est donnée sur le fonctionnement interne du processus. Par exemple, si le processus sous-jacent est devant une condition : l'évaluation, et donc le choix de la branche à exécuter, est vu de l'extérieur comme un choix indéterministe.

Définition 2.3.5 Processus exécutables : les processus exécutables représentent la nature et l'ordre des différents échanges entre les différentes parties : ils définissent le processus métier lui-même. Les processus exécutables sont directement exécutable par un moteur d'orchestration.

À l'opposé des processus abstraits, les processus exécutables décrivent le fonctionnement interne du processus. Le cheminement interne et les conditions sont donc visibles dans la description de ces processus. Les deux principaux avantages de posséder la description interne du processus métier sont :

- un moteur (ou un serveur) peut directement exécuter ces processus ;
- l'évaluation des conditions peut-être connue et donc n'est plus vue comme un choix indéterministe par les autres partenaires.

Bien évidement, il faut modérer l'utilisation des deux méthodes, et pour s'interfacer correctement avec un service web, l'utilisation des descriptions de processus abstraits est suffisante dans la majorité des cas.

Interaction à mémoire Contrairement à une description WSDL, les descriptions de services web complexes permettent de gérer un état (représenté par un ensemble de variables et leurs valeurs) au niveau du serveur. Ainsi, le résultat d'une méthode dépendra de la valeur d'une variable affectée lors d'un précédent appel à une méthode du même service par le même client. Ce mécanisme, appelé *interaction à mémoire* permet de réaliser des opérations plus complexes et d'alléger les communications : l'historique peut-être stocké au niveau du serveur et ne doit plus être envoyé à chaque échange.

Les différents langages de services web complexes De nombreux langages ont été développés pour répondre à ces besoins. Parmi ceux-ci, on peut citer :

- *WSFL* : le langage *Web Services Flow Language* développé par IBM [Ley01] dans le but de rendre possible la description de la composition d'un ensemble de services web en se basant sur la composition des flots de manière hiérarchique.

– *BPML* : le langage *Business Process Modeling Language* [AA02] développé par l'organisation BPMI.org (*Business Process Management Initiative*), regroupant des acteurs comme Abode, Corel, BEA, IBM, Sun, etc... Il présente de nombreux points communs avec les langages que nous allons développer par la suite, comme la notion de processus et d'activité.
– *XLANG* : la langage *XLANG* [Tha01] a été développé par Microsoft en 2001, pour les besoins de sa plateforme de gestion de processus BizTalk. Ce langage permet de représenter les éléments clés, d'un point de vue algorithmique, des processus métier et se classe ainsi dans les langages de description comportemantale.
– *BPEL4WS* : le langage BPEL4WS [ACD+03] est développé en 2003, par un regroupement d'industriels, parmi ceux-ci, on retrouve IBM, BEA et Microsoft. Ce langage est une fusion des langages, dit de première génération : XLANG et WSFL. Il reprend donc les avantages de ces deux langages en tirant parti de l'apprentissage lié à leur mises en place. Il se focalise sur la représentation du processus métier en se rapprochant des structures algorithmiques.

La suite de cette section présentera deux de ces langages :
– *XLANG* car, comme nous l'avons précisé, il ressort de la première génération des langages de description de processus métier exécutables. C'est également le premier langage que notre plateforme a supporté.
– *BPEL4WS* car il représente l'évolution, dans le temps, des premiers langages de ce type, et plus particulièrement la fusion de WSFL et XLANG. De plus il s'avère être le langage le plus utilisé par les industriels, devenant ainsi un standard de fait, en plus d'être en phase de devenir un standard Oasis. Notre plateforme le supporte également.

2.3.4 XLANG

XLANG [Tha01] a été développé par Microsoft dans le but de palier le manque d'information concernant la description comportementale dans les descriptions WSDL des services web (voir section 2.3.3). La plateforme d'exécution de processus métier BizTalk 2003 de Microsoft est basée sur ce langage. Enfin, XLANG est un langage de services web complexes issue de la première génération.

Remarque : l'utilisation d'un langage de services web complexes ne supprime en rien l'utilisation de la description WSDL, cette nouvelle description métier du processus complète la description de son interface (WSDL). Dans le cas de XLANG, la description métier du service se fait dans la partie réservée aux extensions du fichier WSDL.

XLANG repose sur l'utilisation de structures simples :
– les éléments *actions* correspondent aux opérations WSDL,
– les éléments de programmation structurée comme *while*, *switch*, etc...
– les éléments permettant la parallélisation des éléments de base,
– ou encore les éléments d'exécution gardée (au sens événementiel et temporel) avec les éléments *pick* et *context*.

La première instruction du processus métier, décrite à l'aide du langage XLANG, doit être un des éléments ci-dessus. La plupart de ces éléments peuvent utiliser récursivement d'autres éléments du langage (voir ci-dessous pour le détail).

Les éléments de base

L'élément *vide* : c'est l'élément de base le plus petit : aussitôt commencé, aussitôt terminé ! Il est symbolisé par la balise XML `<empty/>`.

Grammaire
```
empty ::= empty
```
Syntaxe XML
```
<empty/>
```

Remarque : les extraits de grammaire et syntaxe XML ne sont là que pour faciliter la compréhension. Ils ne représentent donc pas un exemple ni la grammaire dans sa globalité.

L'élément *action* : cet élément est directement associé au type *opération* de WSDL. Il permet d'exécuter l'opération indiquée.

Grammaire
```
operation ::= action operationRef portRef activation?
              correlationBegin? correlationSetRef*
              portDescription*
```
Syntaxe XML
```
<action operation="AskLastTradePrice" port="pRequest"
        activation="true"/>
```

Les éléments de *délais* : l'une de particularités de XLANG est de prendre en compte la gestion du temps. Voici les deux éléments de bases permettant d'attendre un écoulement de temps : *delayFor* et *delayUntil*.

– L'élément *delayFor* met en pause le processus pendant une certaine durée.

Grammaire
```
delayFor ::= delayFor QName
```
Syntaxe XML
```
<delayFor period="SupplyChain:OrderAcknowledgementTimeout"/>
```

– L'action *delayUntil* met en pause le processus jusqu'à une certaine date.

Grammaire
```
delayUntil ::= delayUntil QName
```
Syntaxe XML
```
<delayUntil clock="SupplyChain:ShippingDeadline" />
```

L'élément déclenchant une *exception* : l'élément *raise* permet le déclenchement d'une exception à tout moment dans le processus métier.

Grammaire
```
raise ::= raise Qname?
```
Syntaxe XML
```
<raise signal="creditCheck:creditDenied">
```

Les contrôles structurés

Les contrôles structurés sont similaires à ceux que l'on connaît des langages de programmation structurée. Leur corps est constitué alors d'autres éléments de contrôle séquentiel ou des éléments de base vu précédemment.

L'élément de *séquence* : il permet de séquentialiser une ou plusieurs actions. La séquence elle-même ce termine en même temps que la dernière action à effectuer.

Grammaire
```
sequence ::= sequence [action | process]*
```
Syntaxe XML
```
<xlang:sequence>
  <xlang:action operation="AskLastTradePrice" activation="true"/>
  <xlang:action operation="SendLastTradePrice"/>
</xlang:sequence>
```

L'élément de *choix* : comme en programmation structurée, l'élément *choix* (ou *switch*) permet de réaliser un branchement conditionnel : suivant une certaine condition, l'exécution dérive sur une branche (symbolisée par un processus) bien précise. Si aucune des conditions n'est évaluée à vrai, alors la branche par défaut est exécutée.

Grammaire
```
switch ::= switch branch* default?
default ::= default process
branch ::= branch case process
case ::= case QName
```

L'élément de *boucle* : l'élément de *boucle* de XLANG est la boucle *while* traditionnelle : effectuer une action tant qu'une condition est vérifiée.

Grammaire
```
while ::= while case process
```

Les contrôles évolués

En plus des éléments de base et structurés vu précédemment, XLANG présente des outils allant de la parallélisation d'actions à l'exécution gardée (gestion des exceptions, évènements et aspect temporel).

L'élément de *parallélisation* : cet élément, appelé *all*, permet d'exécuter différentes tâches en parallèle. Par exemple, dans le cas d'un service devant trouver le prix le moins cher pour un ticket d'avion, le service peut lancer les demandes de prix à différentes compagnies en même temps.

Grammaire
```
all ::= all process*
```

L'élément *pick* : cet élément ressemble fortement à l'élément structuré *switch*, à cela prêt que les conditions deviennent :

- *des évènements* : lors de l'arrivée d'un message, on exécute la branche associée à ce message.
- *des attentes* : l'exécution du processus *pick* active un temps maximal d'attente, puis le processus est mis en attente. Soit un autre évènement arrive avant l'écoulement du temps maximal (et alors il est traité normalement et l'alarme n'est pas déclenchée), soit l'alarme se déclenche et lance l'exécution d'un processus donné.

– *une exception* : si une exception est levée pendant le temps d'attente, alors on exécute un processus de traitement associé à cette exception.

Cet élément permet de gérer des évènements extérieurs au déroulement normal de l'exécution. Par contre, l'espace temps possible, permettant l'arrivée de ces évènements, est limitée par une garde temporelle. De plus, cette instruction a une portée limitée entre deux autres instructions.

Grammaire

```
pick ::= pick eventHandler+
eventHandler ::= eventHandler event process
event ::= delayFor | delayUntil | operation | catch
catch ::= catch QName
```

L'élément *context* : c'est un élément très similaire à l'élément *pick*. Le but de cet élément est d'exécuter un processus de façon gardée. C'est-à-dire qu'il permet de parer à l'éventualité :

– *d'une exécution du processus principal trop longue* : une alarme est alors déclenchée et un processus associé à cette alarme est exécuté.
– *d'une exécution du processus principal déclenchant une exception* : un traitement associé à cette exception est alors exécuté.
– *de la survenue d'un message non prévue dans l'exécution du processus principal* : dans ce cas encore, un processus associé à ce message est exécuté.

Cet élément représente les instructions traditionnelles des *trigger* ou les blocs *try/catch*. Contrairement à l'élément *pick*, sa portée n'est pas limitée entre deux instructions, mais sur un bloc d'instructions.

Grammaire

```
context ::= context localDecls? process transaction? exception?
transaction ::= transaction txnid compensation?
txnid ::= NCName
compensation ::= compensation process
exception ::= exception pick finally?
finally ::= finally process
```

2.3.5 BPEL

Historique

Après une période très riche en création de nouveaux langages de description comportementale, les acteurs majeurs tel que Sun, IBM, BEA ou encore Microsoft se sont associés pour écrire un nouveau langage, héritant des avantages des précédents. Ainsi, la fusion des langages XLANG et WSFL a donné naissance à BPEL4WS.

Historiquement, BPEL4WS (que nous appellerons par la suite BPEL) fait partie de ce que nous qualifierons comme étant les langages sémantiques de seconde génération :

– la première version date de Août 2002 [CGK+02],
– la version 1.1 de Mars 2003 [ACD+03],
– la version 2.0 encore à l'état de *draft* (première version *draft* en mai 2005, toujours à l'état de *draft* en juin 2006 [AAA+06]). Cette version aura l'avantage d'être un standard OASIS (Organization for the Advancement of Structured Information Standards).

Ce langage est aujourd'hui utilisé de façon majoritaire par les acteurs des technologies basées sur les services web :

- Microsoft l'utilise dans sa plateforme Biztalk 2004 (serveur d'exécution de processus métier) ;
- Oracle l'utilise comme langage de description et d'exécution de processus métier dans son serveur Oracle BPEL ;
- IBM l'utilise également dans sa plateforme d'exécution de services web WebSphere ;
- ActiveBpel, un moteur d'exécution de services web *Open Source*, est également basé sur l'exécution de processus métier décrit en BPEL.

Avantages et fonctionnalités

Le langage BPEL présente plusieurs avantages par rapport à ses prédécesseurs :

- tout d'abord, la spécification, ayant été travaillée par plusieurs acteurs industriels, est bien plus finie et ne présentent pas d'ambiguïté, contrairement aux langages dit de première génération. Les objectifs sont donc fixés de manière claire et précise et chaque élément est détaillé.
- les éléments syntaxiques de BPEL permettent de modéliser aussi bien des processus abstraits que des processus exécutables (voir section 2.3.3).
- tous les éléments présentent des mécanismes de compensation : très utile lors d'une interaction longue dans le temps ou pour implémenter un mécanisme de transaction.
- enfin, le processus permettant la parallélisation présente également des liens de synchronisations.

BPEL est basé sur XML et il supporte les protocoles standards des services web :

- *SOAP*, *WSDL*, *UDDI* : les standards des services web (développés dans la section 2.3.2).
- *WS-Reliable Messaging* [BBC$^+$05] : ce standard Oasis décrit un protocole qui permet aux messages d'être délivrés de façon fiable entre des applications réparties, dans le cas de pannes composants, réseaux ou systèmes.
- *WS-Addressing* [BCC$^+$04] : ce standard définit un mécanisme permettant de transporter des messages SOAP de façon bidirectionnelle, en mode synchrone ou asynchrone, indépendamment de la couche de transport utilisée. Il consiste en une évolution de WSIF (*Web Services Invocation Framework*) proposé par IBM.
- *WS-Transaction* : cet ensemble de standards définit des mécanismes interopérables permettant de réaliser des transactions entre différents domaines de services web. Il décrit un *framework* extensible de coordination (*WS-Coordination* [CCF$^+$05a]) pour les propriétés ACID des transactions (*WS-AtomicTransaction* [CCF$^+$05b]) mais également pour les transactions longues des processus métiers (*WS-BusinessActivity* [CCF$^+$05c]).

Principes et syntaxe du langage

La description BPEL d'un processus métier est composée de quatre parties :

- une première partie permet de déclarer des variables, utilisant les types importés de descriptions WSDL associées à la description BPEL. Ces variables seront utilisées dans la partie description comportementale du processus métier pour maintenir un état au niveau du service, et ainsi assurer des liaisons de données entre deux opérations.
- une deuxième partie permet la déclaration des *partnerlinks* : ces *partnerlinks* définissent une liaison entre les ensembles d'opérations des services invoqués par le service web BPEL et un nom de liaison bien précis du service BPEL. Ces liaisons sont appelées *partner* et permettent ainsi d'identifier facilement les différents services web utilisés.

– une troisième partie permet la déclaration des *fauthandlers* : ce sont des processus à déclencher en cas d'exception déclenchée pendant l'exécution du processus et non interceptée avant.
– enfin, la quatrième partie décrit le comportement du processus métier lui-même en utilisant différents opérateurs dit de programmation.

Éléments syntaxiques

Syntaxe des attributs standards (*standard-attributes*) : se sont des propriétés de balises XML utilisées dans les balises de description comportementale du service.

Syntaxe XML
```
name="ncname"?
joinCondition="bool-expr"?
suppressJoinFailure="yes|no"?
```

L'élément `name` permet de donner un nom aux instructions BPEL. Les éléments *joinCondition* et *suppressJoinFailure* sont utilisés dans les mécanismes de liens (voir la description de l'élément de parallélisation *flow* page 43 et la section 7.2.3) pour préciser s'il est possible de supprimer l'instruction en cas d'interblocage d'un processus métier exécutant du code en parallèle.

Syntaxe des éléments standards (*standard-elements*) : se sont des balises permettant de gérer la synchronisation de processus s'exécutant en parallèle (mécanisme des *links*). La présence de la balise *target* empêche l'exécution du processus la contenant tant qu'une balise *source* de même nom (*linkName*) n'a pas été rencontrée. Une condition optionnelle (*transitionCondition*) permet de réaliser une évaluation pour « *activer* » la source.

Syntaxe XML
```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

Les éléments de base

L'élément *vide* : c'est l'élément de base le plus simple : il permet d'insérer une opération vide dans un processus. Ceci signifie que le processus ne fait rien, mais ne se termine pas complètement : l'instruction suivante sera exécutée. Ce processus est très utilisé dans le cadre de la synchronisation des activités parallèles : il supporte la gestion des *links*.

Syntaxe XML
```
<empty standard-attributes>
  standard-elements
</empty>
```

L'élément de *terminaison* : cet élément force le processus métier à se terminer immédiatement.

Syntaxe XML
```
<terminate standard-attributes>
  standard-elements
</terminate>
```

L'élément d'*affectation* : cet élément permet d'affecter un nouveau contenu à une variable.

Syntaxe XML
```
<from variable="ncname" part="ncname"? query="queryString"?/>
<to variable="ncname" part="ncname"? query="queryString"?/>
```

L'élément déclenchant *une exception* : cet élément permet de lever une exception qui pourra être interceptée à différents niveaux.

Syntaxe XML
```
<throw faultName="qname" faultVariable="ncname"?
       standard-attributes>
  standard-elements
</throw>
```

L'élément de *délai* : permet d'attendre un temps donné (*for*) ou jusqu'à une certaine heure (*until*).

Syntaxe XML
```
<wait (for="duration-expr" | until="deadline-expr")
      standard-attributes>
standard-elements
</wait>
```

Les éléments basés sur les messages

Les éléments présentés dans cette partie gèrent les échanges de message avec les clients ou les services web invoqués depuis un service donné.

L'élément de *réception* : permet à un processus métier de se mettre en attente (synchrone) d'un message : typiquement, la première opération d'un serveur consiste à l'attente d'une demande de traitement envoyée par un client. Cet élément est lié aux informations *partnerLink* et *portType* de la description WSDL pour l'opération indiquée par l'attribut XML *operation*. Le contenu reçu peut être affecté à la variable indiquée par l'attribut *variable*.

Syntaxe XML
```
<receive partnerLink="ncname" portType="qname" operation="ncname"
    variable="ncname"? createInstance="yes|no"?
    standard-attributes>
  standard-elements
</receive>
```

L'élément de *réponse* : permet à un processus de répondre à un message qu'il aurait reçu par un *receive*. La combinaison d'un *receive* et d'un *reply* permet de réaliser une opération de question/réponse tel que le définit WSDL, en mode synchrone. Comme l'élément de *réception*, il est lié aux informations *partnerLink* et *portType* de la description WSDL pour l'opération indiquée par l'attribut XML *operation*. Le contenu envoyé peut provenir de la variable indiquée par l'attribut *variable*.

Syntaxe XML
```
<reply partnerLink="ncname" portType="qname" operation="ncname"
    variable="ncname"? faultName="qname"?
    standard-attributes>
  standard-elements
</reply>
```

L'élément d'*invocation* d'un service web : permet d'invoquer un service web partenaire (défini dans la partie *partners* vue précédemment). Cette invocation peut être réalisée de manière asynchrone. L'opération WSDL invoquée est indiquée par l'attribut XML *operation* et associée aux informations *partnerLink* et *portType*. Le contenu envoyé peut provenir de la variable indiquée par l'attribut *outputVariable* et le contenu reçu peut être affecté à la variable indiquée par l'attribut *inputVariable*.

L'opérateur permet l'ajout de garde (*catch* et *catchAll*) en cas d'exception, et également de mécanismes de compensation en cas d'annulation d'une transaction (*compensationHandler*).

Syntaxe XML
```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
    inputVariable="ncname"? outputVariable="ncname"?
    standard-attributes>
  standard-elements
  <catch faultName="qname" faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
  <compensationHandler>?
    activity
  </compensationHandler>
</invoke>
```

Les contrôles structurés

Comme dans le langage XLANG, les contrôles structurés sont similaires à ceux que l'on connaît des langages de programmation structurée. Leur corps est constitué d'autres éléments de contrôle séquentiel ou des éléments de base ou d'échanges de messages vu précédemment.

L'élément de *séquence* : permet d'exécuter de façon séquentielle (dans l'ordre de lecture de la séquence) une ou plusieurs instructions BPEL (mot clé *activity*). La séquence se termine elle-même lorsque le dernier processus la constituant se termine.

Syntaxe XML
```
<sequence standard-attributes>
  standard-elements
  activity+
</sequence>
```

L'élément de *choix* : permet, comme en programmation structurée, d'effectuer un branchement conditionnel : suivant l'évaluation d'une condition ou d'un critère (attribut XML *condition*), le processus de la première branche (nœud XML *case*) répondant à ce critère sera exécuté. Dans le cas

où aucune condition ne serait vraie, il est possible de définir une branche *par défaut* (nœud XML *otherwise*) qui sera alors exécutée.

Remarque : l'évaluation de la condition est interne au service et ne correspond à aucun échange entre le service et le client et/ou ses partenaires.

Syntaxe XML

```
<switch standard-attributes>
  standard-elements
  <case condition="bool-expr">+
    activity
  </case>
  <otherwise>?
    activity
  </otherwise>
</switch>
```

L'élément de *boucle* : permet de réaliser une boucle *while* traditionnelle : la boucle (composée des instructions représentées par le mot clé *activity*) est exécutée tant que la condition (attribut XML *condition*) est évaluée à *vrai*. Tout comme dans le cas du *switch*, l'évaluation se passe de manière silencieuse pour les clients et/ou partenaires du service.

Syntaxe XML

```
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

Les contrôles évolués

En plus des éléments de base et structurés vus précédemment, BPEL, comme le langage XLANG, présente des outils allant de la parallélisation d'actions à l'exécution gardée (gestion des exceptions, évènements et aspect temporel).

L'élément de *parallélisation* : permet d'exécuter en parallèle plusieurs processus (mot clé *activity*).

Ces différents processus peuvent être indépendants, et alors, le processus englobant l'exécution parallèle se termine lorsque tous ses processus sont terminés. Ou alors, les processus peuvent être dépendants les uns des autres, et l'utilisation de *liens* (*links*) permet de synchroniser où d'ajouter des dépendances temporelles entre les différents processus.

Ainsi, l'ensemble des *liens* est annoncé dans la déclaration du processus parallèle et ensuite, chaque élément des sous-processus peut utiliser les *sources* et les *cibles* (*target*) : une *cible* n'est franchissable que si une *source* ayant le même nom a déjà été franchie (ou plus exactement l'exécution du processus déclarant cette *source* est terminée).

Syntaxe XML
```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="ncname">+
  </links>
  activity+
</flow>
```

L'élément *pick* : regroupe différents processus dont l'exécution est gardée par :
- *des évènements* (nœud XML *onMessage*) : à l'arrivée d'un message, le processus de la branche associée à ce message sera exécuté.
- *un temps d'attente* (nœud XML *onAlarm*) : au bout d'un temps d'attente donné, le processus de la branche est exécuté.

Seule une branche est exécutée : le processus de la première branche dont la garde est évaluée à *vrai* s'exécute.

Cet élément permet de gérer des évènements extérieurs au déroulement normal de l'exécution. Par contre, l'espace temps possible, permettant l'arrivée de ces évènements, est limitée par une garde temporelle. De plus, cette instruction a une portée limitée entre deux autres instructions.

Syntaxe XML
```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
      operation="ncname" variable="ncname"?>+
    <correlations>?
      <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>
```

L'élément *scope* : c'est un élément très similaire à l'élément *pick*, et appelé *context* dans le langage XLANG. Il permet d'exécuter un processus de façon gardée. C'est-à-dire qu'il permet de parer à l'éventualité :
- *d'une exécution du processus principal trop longue* (nœud XML *eventHandlers*) : une alarme est alors déclenchée et un processus associé à cette alarme est exécuté.
- *d'une exécution du processus principal déclenchant une exception* (nœud XML *faultHandlers*) : un traitement associé à cette exception est alors exécuté.
- *de la survenue non prévue d'un message dans l'exécution du processus principal (évènement extérieur)* (nœud XML *eventHandlers*) : encore dans ce cas, un processus associé à ce message est exécuté.

Cet élément représente les instructions traditionnelles *trigger* ou les blocs *try/catch*. Contrairement à l'élément *pick*, sa portée n'est pas limitée entre deux instructions, mais sur un bloc d'instructions (mot clé *activity*. Il définit également un bloc *compensationHandler* permettant d'annuler les instructions réellement effectuées en cas de mécanisme d'annulation de transactions.

Syntaxe XML

```
<scope variableAccessSerializable="yes|no" standard-attributes>
  standard-elements
  <variables>?
    ...
  </variables>
  <faultHandlers>?
    ...
  </faultHandlers>
  <compensationHandler>?
    ...
  </compensationHandler>
  <eventHandlers>?
    ...
  </eventHandlers>
  activity
</scope>
```

2.4 Synthèse

En partant du constat du besoin d'une interopérabilité toujours plus grande et d'une évolution des techniques et outils à disposition des développeurs et architectes, ce chapitre retrace, comment en quelques années, l'architecture Orientée Objet a évolué vers l'architecture Orientée Service. Ce chapitre met en évidence une instance du SOA : les services web. En effet, ces services web seront le centre de nos préoccupations dans la suite de cette thèse. C'est pourquoi, ce chapitre présente également quelques unes des technologies de ce domaine et plus particulièrement comment fonctionne un service web, et la mise à disposition, en plus de la description WSDL, des descriptions comportementales tel que XLANG ou BPEL.

Chapitre 3

Sémantique temporisée

Il existe une grande quantité de modèles mathématiques permettant de décrire le comportement d'un système. Nous nous concentrerons ici que sur une partie de ces modèles, et principalement ceux rencontrés dans la suite de cette thèse. Rappelons qu'un modèle, pour être utilisable, doit avoir des fondements mathématiques et une sémantique précise. Ceci permet alors d'appliquer ou de vérifier des propriétés sur le modèle d'un système et de fournir une réponse rigoureuse quand à la validité ou non de notre système (par le biais du modèle). Enfin, le choix du modèle se fait en fonction du type de système étudié.

La première partie présente les systèmes de transitions temporisés que nous utiliserons par la suite, et plus particulièrement les systèmes de transitions temporisés à entrées et sorties, suivie des automates temporisés. Puis, dans la deuxième partie nous aborderons les langages formels et les méthodes de vérifications associées, en nous concentrant particulièrement sur les algèbres de processus temporisés et sur les différentes méthodes de vérifications. Enfin, la troisième partie présentera un état de l'art sur la formalisation des services web et comment les différents modèles et outils présentés sont appliqués aux services web.

3.1 Systèmes de transitions

3.1.1 Systèmes de transitions étiquetées (LTS)

Les systèmes de transitions étiquetées (*Labeled Transition System - LTS*, voir la figure 3.1 pour un exemple) sont un modèle permettant de représenter l'évolution d'un système par un graphe incluant des états reliés par des transitions étiquetées par des actions.

Les états. Un état du LTS représente un état du système prenant en compte les évènements de ce système que l'on souhaite modéliser, c'est-à-dire une étape (visible ou non) à instant donné du système réel.

Les transitions. Les transitions du LTS relient deux états et indiquent les actions qui provoquent les changements d'états du LTS (et donc, par extension, du système).

Ainsi, à un instant donné, l'évolution du système se traduit par le passage d'un état à un autre en franchissant une des transitions reliant ces deux états.

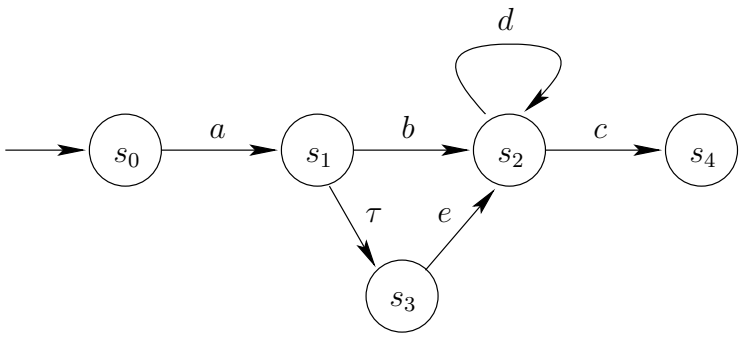


FIG. 3.1 – Un exemple de LTS.

Définition 3.1.1 (Système de transitions étiquetées (LTS)) Un LTS est un tuple $< Q, Q_0, \Sigma, T >$ tel que :

1. Q est un ensemble d'états,
2. $Q_0 \subseteq Q$ est un ensemble d'états initiaux,
3. Σ est un ensemble d'étiquettes (représentant des actions ou des évènements),
4. $T \subseteq Q \times (\Sigma \cup \{\tau\}) \times Q$ est un ensemble de transitions (τ représente une action interne).

Une *transition* $< q, a, q' > \in T$ s'écrit aussi $q \xrightarrow{a} q'$. En supposant que le système commence sur un état initial q, et si $q \xrightarrow{a} q'$ alors le système peut passer de l'état q à l'état q' sur l'évènement a.

Un état q' est dit *atteignable* de l'état q (noté $q \rightarrow^* q'$), c'est-à-dire s'il existe une suite de transitions permettant, depuis q, d'atteindre q'. Enfin l'état q est un *état atteignable* sur le système de transitions si q est atteignable à partir de l'état initial.

Un *chemin* est la suite des états et des étiquettes $q_0 a_1 q_1 \ldots a_n q_n$ permettant d'atteindre l'état q_n à partir de l'état q_0 en franchissant la transition a_1 pour atteindre l'état intermédiaire q_1 et ainsi de suite jusqu'au franchissement de la transition a_n.

3.1.2 Systèmes de transitions temporisés à entrées et sorties (TIOTS)

Avant de présenter le modèles des TIOTS, nous allons présenter les différences entre le temps discret et le temps dense, et également les différences entre le modèle synchrone et le modèle asynchrone.

Temps discret et temps dense

Deux méthodes de modélisation du temps s'opposent : le *temps discret* et le *temps dense*.

Le *temps discret* consiste à approximer l'écoulement de temps à l'aide de *tick* d'horloge : chaque *tick* représentant alors une unité de temps. Un écoulement de temps est donc représenté par un entier.

Le *temps dense* (aussi appelé temps continu), contrairement au temps discret, ne réalise plus la discrétisation du temps, et mesure un écoulement de temps par un réel.

Modèle synchrone et modèle asynchrone

On peut distinguer deux types de modèles : les *modèles synchrones* et les *modèles asynchrones*.

Le *modèle synchrone* est appliqué aux systèmes où les évènements (échange de message, etc...) se font dans un temps nul et de façon simultanées sur les différents systèmes (d'où l'aspect de synchronisation).

Le *modèle asynchrone* est appliqué aux systèmes où les évènements peuvent arriver à n'importe quel moment de l'exécution, que ce soit prévu ou non. C'est l'hypothèse la plus proche de la réalité, mais la plus difficile à mettre en place et à utiliser, notament dans le cadre des systèmes répartis.

Le modèle des TIOTS

L'ensemble des actions d'un TIOTS, symbolisés par des étiquettes Σ dans un LTS, contient différents types d'actions (voir la figure 3.2 pour un exemple) :

- *les actions symbolisant une entrée :* ces actions correspondent à une réception de message, à un bouton pressé sur le système, etc... Les étiquettes correspondant à ces actions commencent par le symbole " ?".
- *les actions symbolisant une sortie :* ces actions correspondent à un envoi de message, à un affichage, etc... Les étiquettes correspondant à ces actions commencent par le symbole " !".
- *l'action interne :* cette action symbolise une action silencieuse permettant de changer l'état du système, mais non symbolisée de façon concrète comme par l'échange d'un message. Vue de l'extérieur, rien n'est changé sur le système réel et rien n'est visible. Cette action est étiquetée τ.
- *l'action symbolisant l'écoulement de temps :* cette action permet de montrer un écoulement de temps. Cette action est étiquetée χ.
- *l'action symbolisant la terminaison :* cette action permet de préciser que le système arrive dans un état final et a terminé son exécution. Cette action est étiquetée $\surd$.

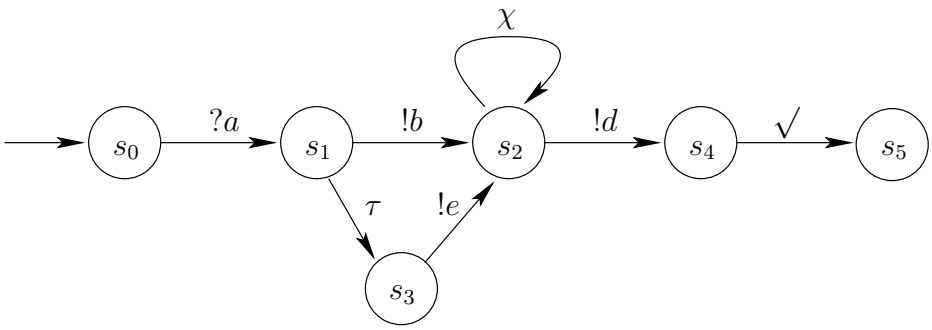


FIG. 3.2 – Un exemple de TIOTS.

Remarque : plus l'écoulement de temps est petit, plus le modèle est relativement proche d'un système réel asynchrone, mais ceci augmente alors le nombre de transitions et d'états possibles du modèle, rendant bien souvent difficile la modélisation d'un système réel complexe : il faut donc trouver un compromis ou se limiter à la modélisation des systèmes synchrones à l'aide d'un TIOTS.

Définition 3.1.2 (Systèmes de transitions temporisés à entrées et sorties (TIOTS)) Un TIOTS est un tuple $< Q, Q_0, \Sigma, T >$ tel que :

1. Q est un ensemble d'états,
2. $Q_0 \subseteq Q$ est un ensemble d'états initiaux,
3. Σ est un ensemble d'étiquettes (représentant des actions ou des évènements),
4. $T \subseteq Q \times ((\{?, !\} \times \Sigma) \cup \{\tau, \chi, \surd\}) \times Q$ est un ensemble de transitions (τ représente une action interne).

3.1.3 Automates Temporisés (AT)

Alur et Dill ont proposé (version originale dans [AD94] et version révisée dans [Alu98]) une modélisation des systèmes temporisés décrivant le temps de manière continue grâce à une ou plusieurs *horloges*. Cette modélisation s'applique alors à la plupart des systèmes temporisés (synchrones et asynchrones).

Les horloges dans les automates temporisés

Un *automate temporisé* (*timed automaton*, voir un exemple sur la figure 3.3) est un LTS auquel est associé un ensemble d'horloges. Chaque horloge est un réel qui évolue dans le temps (représentation dense ou continue du temps). Les horloges, bien que n'ayant pas toutes les mêmes valeurs à un instant donné, évoluent toutes à la même vitesse. À tout instant, une horloge indique le temps écoulé depuis sa dernière ré-initialisation.

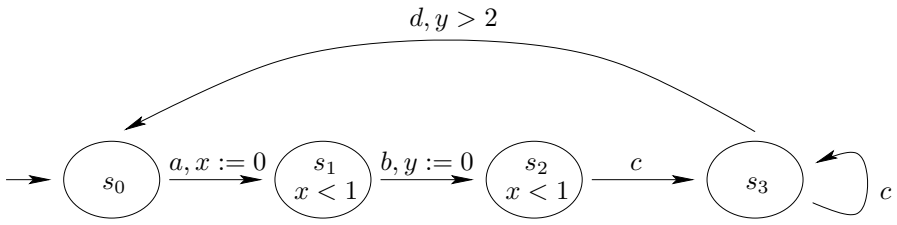


FIG. 3.3 – Un automate temporisé à 2 horloges.

Les horloges sont toutes initialisées à 0 dans l'état initial de l'automate, mais elles peuvent être réinitialisées au franchissement de chaque transition. Ces transitions peuvent également posséder des contraintes sur ces horloges qui devront être satisfaites de façon globales pour que la transition soit franchissable.

Définition 3.1.3 (Contraintes d'horloges, d'après [AD94]) Pour un ensemble X d'horloges, l'ensemble $\Phi(X)$ de contraintes d'horloges δ est défini de façon inductive par :

$$\delta := x \leq c \,|\, c \leq x \,|\, \neg\, \delta \,|\, \phi_1 \wedge \phi_2,$$

où x est une horloge de X et c un réel.

Définition 3.1.4 (Interprétation d'horloges) Une interprétation d'horloges ν d'un ensemble d'horloges X, donne une valeur réelle pour chaque horloge : c'est une fonction de X dans $\mathbb{R}^+$. On dit

qu'une interprétation d'horloge ν de X satisfait une contrainte d'horloge δ si et seulement si δ est évaluée à vrai à partir des valeurs données par ν.

- Pour $t \in \mathbb{R}$, $\nu + t$ dénote l'interprétation d'horloge qui associe à chaque horloge x la valeur $\nu(x) + t$, et l'interprétation d'horloge $t \cdot \nu$ associe à chaque horloge x la valeur $t \cdot \nu(x)$.
- Pour $Y \subseteq X$, $[Y \mapsto t]\nu$ dénote l'interprétation d'horloge pour X qui assigne t pour chaque $x \in Y$, et en accord avec les horloges ν restantes.

Les automates temporisés

Voici la définition des automates temporisés que nous adoptons [Alu98].

Définition 3.1.5 (Automate temporisé) Un automate temporisé $\mathcal{A}$ est tuple $\langle L, L_0, \Sigma, X, I, E\rangle$ tel que :

- L est un ensemble fini d'états,
- $L_0 \subseteq L$ est un ensemble d'états initiaux,
- Σ est un ensemble fini d'étiquettes,
- X est un ensemble fini d'horloges,
- I est une association entre les étiquettes et les états s de L, avec des contraintes d'horloges dans $\Phi(X)$,
- $E \subseteq L \times \Sigma^X \times \Phi(X) \times L$ est un ensemble d'arcs. Un arc $\langle s, a, \varphi, \lambda, s'\rangle$ représente une transition de l'état s à l'état s' étiquetée par le symbole a. φ est une contrainte d'horloge sur X qui spécifie quand la transition est franchissable, et l'ensemble $\lambda \subseteq X$ donne les horloges qui doivent être réinitialisées en franchissant cette transition.

Sémantique d'un automate temporisé [Alu98]

La sémantique d'un automate temporisé $\mathcal{A}$ est définie en associant un système de transitions étiquetées $S_{\mathcal{A}}$ à $\mathcal{A}$ lui-même. Un état de $S_{\mathcal{A}}$ est un couple (s, ν) tel que S est un état de $\mathcal{A}$ et ν une interprétation d'horloge de X tel que ν satisfait l'invariant $I(s)$. On note $Q_{\mathcal{A}}$ l'ensemble de tous les états de $\mathcal{A}$. Un état (s, ν) est un état initial si s est un état initial de $\mathcal{A}$ et $\nu(x) = 0$ pour toutes les horloges x.

Il existe alors deux types de transitions dans $S_{\mathcal{A}}$ correspondant à :

- un changement d'état suite au temps qui s'écoule : pour un état (s, ν) et l'incrément de temps réel $\delta \geq 0$, $(s, \nu) \xrightarrow{\delta} (s, \nu + \delta)$ si $\forall \delta$ (avec $0 \leq \delta' \leq \delta$), $\nu + \delta'$ satisfait l'invariant $I(s)$.
- un changement d'état suite à changement d'état dans A : pour un état (s, ν) et un arc $\langle s, a, \varphi, \lambda, s'\rangle$ tel que ν satisfait φ, $(s, \nu) \xrightarrow{a} (s', \nu[\lambda := 0])$.

Donc, $S_{\mathcal{A}}$ est un système de transitions étiquetées dont l'ensemble des étiquettes est $\Sigma \cup \mathbb{R}$.

Par exemple, pour l'automate temporisé de la figure 3.3 :

- l'espace d'états du LTS est $\{s_0, s_1, s_2, s_3\} \times \mathbb{R}^2$
- l'ensemble des étiquettes est $\{a, b, c, d\} \times \mathbb{R}$
- un exemple de transitions possible (appelé exécution temporisée ou mot temporisé) est : $(s_0, 0, 0) \xrightarrow{1.2}$ $(s_0, 1.2, 1.2) \xrightarrow{a} (s_1, 0, 1.2) \xrightarrow{0.7} (s_1, 0.7, 1.9) \xrightarrow{b} (s_2, 0.7, 0) \xrightarrow{0.1} (s_2, 0.8, 0.1) \xrightarrow{0.1} (s_2, 0.9, 0.2)$

Déterminisme et non déterminisme

Nous utiliserons, par la suite, les automates temporisés comme un modèle pour produire un programme exécutable. Ce programme devra alors être déterministe. Donc une question se pose : à savoir l'aspect déterministe ou non des systèmes modélisés par des automates temporisés.

Les automates temporisés représentent des systèmes déterministes ou non. Une sous-classe de ces automates sont déterminisables [AD94, AP04].

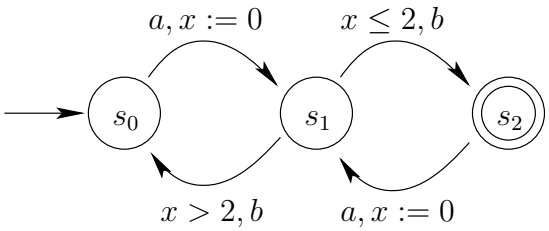


FIG. 3.4 – Exemple d'automate temporisé dont les gardes permettent de lever le non déterminisme.

Le problème se pose en effet lors du choix de franchissement de deux transitions ayant la même étiquette : ceci entraîne le non déterminisme car on ne sait pas laquelle sera réellement franchie (d'un point de vue interne au système, il est possible de le connaître, mais pas d'un point de vue externe). Il est possible d'avoir des informations supplémentaires grâce aux gardes par exemple pour lever ce non déterminisme (voir un exemple figure 3.4).

Un automate temporisé est dit *déterministe* si et seulement si :
- il possède un seul état de départ,
- et $\forall s \in L$, $\forall a \in \Sigma$, pour chaque couple de la forme $\langle s, a, \varphi_1, -, - \rangle$ et $\langle s, a, \varphi_2, -, - \rangle$, les contraintes des horloges φ_1 et φ_2 sont exclusives mutuellement (c'est-à-dire la condition $\varphi_1 \wedge \varphi_2$ est non satisfaite).

Les automates à évènements d'horloges

Il existe également une sous-classe d'automates temporisés déterminisables : les *automates à évènements d'horloges* (*event-clock automata*) [AFH99].

Les horloges des automates à évènements d'horloges sont associées de façon prédéfinies aux symboles de l'alphabet d'entrée Σ. Ainsi, une horloge associée au symbole a mesure le temps écoulé depuis la dernière occurrence de ce symbole.

Remarque : il est possible d'étendre le principe à une horloge basée sur la prédiction : cette horloge permet alors de donner l'information sur l'écoulement de temps à attendre avant la prochaine occurrence du symbole.

Contrairement aux automates temporisés, un automate à évènement d'horloge ne contrôle pas la remise à zéro de ces horloges, et, pour chaque symbole d'entrée, toutes les valeurs des horloges sont déterminées seulement par le mot d'entrée (temporisé).

Les auteurs démontrent alors que l'on peut déterminiser ce type d'automate. Le principe est de traiter les transitions possédant des gardes dont le franchissement est possible en même temps pendant

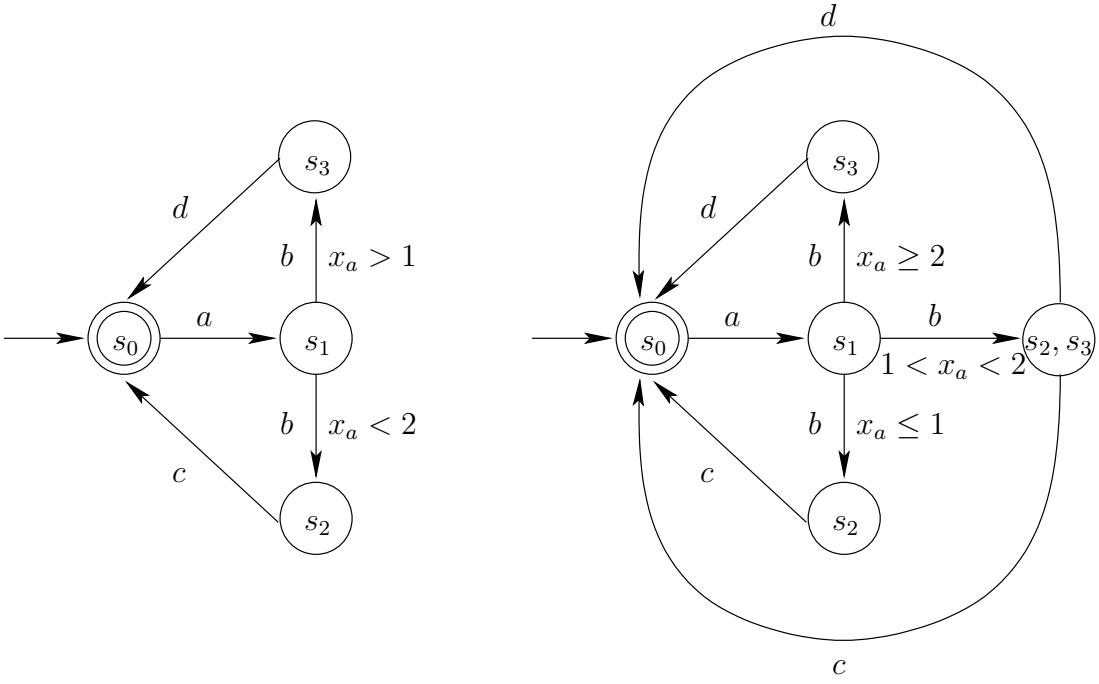


FIG. 3.5 – Exemple d'un automate à évènements d'horloges non déterministe (à gauche) et celui obtenu après déterminisation (à droite).

un certains laps de temps avec la même étiquette (source du non déterminisme) en les transformant en plusieurs transitions dont les gardes seront en exclusion mutuelles (voir exemple figure 3.5). Ainsi, pour un mot temporisé donné, un seul chemin sera possible, ce qui correspond à une exécution déterministe.

3.2 Langages formels et méthodes de vérifications

Cette section, dans un premier temps, se concentre sur deux langages formels de description de processus qui sont LOTOS et les Algèbres de Processus Temporisés de Sifakis. Ce choix est motivé par l'utilisation régulière de LOTOS dans les différentes recherches rencontrées sur la formalisation des services web, et les Algèbres de Processus Temporisés de Sifakis. De plus, la suite de notre travail est basée sur ces algèbres.

Dans un deuxième temps, nous aborderons, après avoir fait une présentation des méthodes de test, les méthodes de vérifications les plus connues (*model checking* et *bisimulation*). Ce sont des méthodes de vérifications que nous utiliserons par la suite. Ils utilisent les modèles et langages formels vu précédemment. Enfin nous présenterons les outils à notre disposition permettant d'appliquer ces différentes méthodes.

3.2.1 LOTOS

LOTOS (*Langage Of Temporal Ordering Specification*) est un langage permettant de décrire des services et des protocoles de télécommunications en spécifiant leur architecture et leur fonctionne-

ment. Ce langage étant défini de façon formelle a permis de lui donner le statut de norme (ISO 8807). De plus, il a été fortement étudié ([Gar90] présente une introduction) et utilisé dans la description des systèmes OSI (Open Systems Interconnection).

Une description LOTOS est un fichier texte ASCII regroupant un ensemble de définitions de processus et de types. Chaque définition de processus peut à son tour comporter un ensemble de définitions de processus et de types. Cela permet l'imbrication des descriptions et la structuration de la description elle-même, très utile dans le cadre de la description de systèmes répartis (systèmes asynchrones). La description du processus est spécialisée dans la description du contrôle, et les types décrivent les données.

La partie contrôle

La partie contrôle et en particulier les structures de contrôles sont très proches des *algèbres de processus* : ce sont des expressions mathématiques décrivant le processus et son évolution à l'aide d'opérateurs de contrôle. Au niveau sémantique, le comportement se traduit directement en automate d'états finis ou infinis.

LOTOS décrit des comportements qui s'exécutent en parallèle et qui correspondent par *rendez-vous*. Le rendez-vous exprime la synchronisation et la communication. Un *signal* correpond à une proposition de participation à un rendez-vous. Un signal est associé à une *porte*, une porte permettant l'envoi et la réception simultanés de différents signaux.

Enfin, la partie contrôle décrit également les *opérateurs séquentiels* et les *opérateurs parallèles* permettent de décrire le flot de contrôle de l'exécution des processus : séquences, choix, exécution en parallèle, etc.

Les structures de contrôles LOTOS ne permettent pas la gestion du temps. Des extensions de LOTOS ont été développées pour remédier à ce problème. On peut citer : RT-LOTOS (*Real Time LOTOS*), ET-LOTOS (*Time Extended LOTOS*) ou encore TI-LOTOS (*Time Interval LOTOS*).

La partie données

Les structures de données de LOTOS s'articulent autour du concept de *type* :
- les *sorts* définissent les types permettant l'« encapsulation » : le nouveau type déclaré est un héritier d'un type de base,
- les *opns* définissent les opérations possibles sur le type (en précisant les opérandes),
- les *eqns* définissent les ensembles d'équations (composé de variable d'un ou plusieurs types) et leur résultats associés.

LOTOS permet un grand nombre de définitions de type :
- utilisation d'une bibliothèque de types de base,
- possibilité pour l'utilisateur de définir ses propres notations,
- surcharge des opérateurs,
- notion d'héritage multiple,
- enrichissement des types,
- renommage des types,
- concept de généricité.

3.2.2 Algèbres de processus temporisés

Les *Algèbres de Processus Temporisés* (appelées par la suite APT) [NS94] permettent de décrire et d'analyser des systèmes de processus temporisés. Une particularité importante de cette algèbre de processus est qu'elle possède nativement une action représentant un écoulement de temps : l'élément χ. L'écoulement de temps ainsi représenté a une durée fixe et peut-être considéré comme un *tic* d'horloge : on parle alors de discrétisation du temps.

Présentation

Le modèle des systèmes temporisés utilisé dans les APT est basé sur les principes suivants :

- Un système temporisé est une composition parallèle de composants séquentiels communicants (appelés processus), qui doivent tous être capables d'exécuter une action appelée *time action* (symbolisé par χ). Les APT se placent donc dans une approche *temps discret*. De plus, toutes les actions du système sont atomiques.
- Le temps s'écoule par des exécutions synchrones d'actions temporelles. En d'autres termes, le temps ne peut s'écouler que si tous les composants du systèmes sont d'accord (peuvent le faire).
- Une exécution de séquences est une suite d'*étapes*. Une étape est déterminée par deux occurrences consécutives d'actions temporelles. Pendant une étape, les composants peuvent exécuter, indépendamment ou non, une séquence arbitrairement longue, mais finie, d'actions asynchrones (autre que l'écoulement de temps). Ensuite, ils doivent réaliser une action temporelle pour se synchroniser à nouveau (correspondant donc à l'écoulement du temps pour le système). Ce qui implique qu'un composant ne pouvant pas réaliser d'action asynchrone doit au moins pouvoir exécuter une action temporelle.

Syntaxe

Soit $\mathcal{A}$ un ensemble dénombrable de constantes appelées *actions*.

Soient l'élément χ et l'ensemble $\mathcal{A}^\alpha = \mathcal{A}\backslash\{\chi\}$. Les éléments de $\mathcal{A}^\alpha$ sont appelés des actions asynchrones et l'action χ représente un écoulement de temps. Par la suite, les symboles $\alpha_1, \alpha_2, \ldots$ représenteront les éléments de $\mathcal{A}^\alpha$ et les symboles $a_1, a_2, \ldots$ les éléments de $\mathcal{A}$.

Une règle, écrite en utilisant les APT, est de la forme (le processus P est constitué des sous processus P_i, $i \in I$) :

$$\frac{\bigwedge_{i\in I}[\neg]P_i \xrightarrow{a_j} P_i'}{P \xrightarrow{a_{j'}} P'}$$

La garde, partie supérieure de la formule, décrit la condition d'activation de la règle. Elle est constituée d'une expression booléenne décrivant les actions a_j que les différents processus P_i peuvent ou non exécuter. Si toutes ces conditions sont satisfaites, alors, le processus P peut évoluer en exécutant l'action $a_{j'}$ (comme le décrit la partie inférieure de la formule), il devient alors le processus P'.

Opérateurs de bases Voici la liste des opérateurs de base :
- δ est le processus bloqué (généralement synonyme de l'état de terminaison du processus).
- $\alpha.P \xrightarrow{\alpha} P$ est une opération de préfixage.
- $\oplus$ est un opérateur de choix non-déterministe.
- rec est un opérateur de récursivité. $rec(X).P \equiv P[(rec(X).P)/X]$.
- $\lfloor\ \rfloor$ est un opérateur de délai.
- $||$ permet le parallélisme (ou la concurrence) des tâches (voir section 3.2.2).
- ∂ est un opérateur d'encapsulation (voir section 3.2.2).

Quelques exemples

Voici quelques exemples montrant l'utilisation et la sémantique des différents opérateurs de cette algèbre.

Délai La syntaxe $\lfloor P \rfloor(Q)$ autorise le processus P à exécuter une (seule) action immédiate, permettant au processus P de poursuive ensuite. Si ce n'est pas le cas, le processus Q prend le contrôle de l'exécution après une unité de temps.

$$\frac{P \xrightarrow{\alpha} P'}{\lfloor P \rfloor(Q) \xrightarrow{\alpha} P'} \qquad \lfloor P \rfloor(Q) \xrightarrow{\chi} Q$$

Par extension, l'opérateur délai devient un opérateur *délai quantifié*. Ainsi l'expression $\lfloor P \rfloor^d(Q)$ indique que le processus P peut réaliser une (seule) action pendant d unités de temps, et ainsi prendre la main. Après ce sera obligatoirement au processus Q de s'exécuter.

$$\frac{P \xrightarrow{\alpha} P'}{\lfloor P \rfloor^d(Q) \xrightarrow{\alpha} P'} \qquad \lfloor P \rfloor^1(Q) \xrightarrow{\chi} Q$$

$$\frac{P \xrightarrow{\chi} P'}{\lfloor P \rfloor^{d+1}(Q) \xrightarrow{\chi} \lfloor P' \rfloor^d(Q)} \qquad \frac{(\nexists P')P \xrightarrow{\chi} P'}{\lfloor P \rfloor^{d+1}(Q) \xrightarrow{\chi} \lfloor P \rfloor^d(Q)}$$

Le *délai d'exécution* $\lceil P \rceil^d(Q)$ permet au processus P de s'exécuter pendant d unités de temps (plusieurs actions, contrairement aux deux opérateurs précédents), mais après, c'est au processus Q de s'exécuter.

$$\frac{P \xrightarrow{\alpha} P'}{\lceil P \rceil^d(Q) \xrightarrow{\alpha} \lceil P' \rceil^d(Q)} \qquad \frac{P \xrightarrow{\chi} P'}{\lceil P \rceil^1(Q) \xrightarrow{\chi} Q} \qquad \frac{P \xrightarrow{\chi} P'}{\lceil P \rceil^{d+1}(Q) \xrightarrow{\chi} \lceil P' \rceil^d(Q)}$$

L'alternative L'*alternative* est un élément très utilisé dans une modélisation de processus métier puisqu'il symbolise un choix lié à une évaluation interne (par exemple une condition sur la mémoire disponible, le temps d'exécution, la valeur d'une variable, etc.). Ce choix est alors non déterministe d'un point de vue extérieur au processus. L'opérateur symbolisant ce choix non déterministe est $\oplus$, dont voici quelques exemples.

$$\frac{P \xrightarrow{\alpha} P'}{P \oplus Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \oplus Q \xrightarrow{\alpha} Q'} \qquad \frac{P \xrightarrow{\chi} P' \wedge Q \xrightarrow{\chi} Q'}{P \oplus Q \xrightarrow{\chi} P' \oplus Q'}$$

Parallélisme Le parallélisme est représenté par le symbole $||$. Soient l'élément $\perp$, $\perp \notin \mathcal{A}$ et l'ensemble $\mathcal{A}_\perp = \mathcal{A} \cup \{\perp\}$. Ce symbole représente un élément absorbant. Associé à l'opérateur de synchronisation $|$ (définit plus bas), il permet de définir les propriétés algébriques suivantes :

- $a_1|a_2 = a_2|a_1$ (commutativité de $|$)
- $a_1|(a_2|a_3) = (a_1|a_2)|a_3$ (associativité de $|$)
- $a|b = \perp$ (signifiant l'impossibilité de synchroniser a et b)
- $a|\ \perp = \perp$ (propagation de l'impossibilité de la synchronisation)
- $\alpha|\chi = \perp$ (pas de synchronisation entre un écoulement de temps et une action immédiate)
- $\chi|\chi = \chi$ (une unité de temps s'écoule sur tous les processus)
- $\alpha_1|\alpha_2 \in \mathcal{A}^\alpha \cup \{\perp\}$

Les éléments synchronisables ou non par l'opérateur de synchronisation $|$ sont :

- $\chi|\chi = \chi$ (l'action *écoulement de temps* est synchronisable avec elle-même)
- $\alpha|\chi = \perp$ (impossibilité de synchroniser une action autre qu'un *écoulement de temps* avec un *écoulement de temps* lui-même).
- $\alpha_1|\alpha_2 \in \mathcal{A}^\alpha \cup \{\perp\}$ (deux actions autre qu'un *écoulement de temps* peuvent se synchroniser ou non, suivant la sémantique propre à ces actions).

En ajoutant l'opérateur parallélisme, voici quelques opérations qu'il est possible de réaliser :

$$\alpha \in \mathcal{A} \quad \frac{P \xrightarrow{\alpha} P'}{P||Q \xrightarrow{\alpha} P'||Q} \quad (1) \qquad \alpha \in \mathcal{A} \quad \frac{Q \xrightarrow{\alpha} Q'}{P||Q \xrightarrow{\alpha} P||Q'} \quad (2)$$

$$\alpha_1, \alpha_2 \in \mathcal{A} \quad \frac{P \xrightarrow{\alpha_1} P' \wedge Q \xrightarrow{\alpha_2} Q' \wedge \alpha_1|\alpha_2 \neq \perp}{P||Q \xrightarrow{\alpha_1|\alpha_2} P'||Q'} \quad (3)$$

La garde de l'équation (1) indique que le processus P doit pouvoir réaliser l'action α pour ensuite passer dans l'état P'. Donc lorsque P se retrouve en parallèle avec le processus Q, il peut réaliser l'action α et passer en P'.

L'équation (2) est similaire, mais pour l'exécution du processus Q.

L'équation (3) montre le vrai parallélisme : si les deux actions ne sont pas impossible à synchroniser ($\alpha_1|\alpha_2 \neq \perp$), il est alors possible de les exécuter toutes les deux en même temps.

Encapsulation L'opérateur d'encapsulation ∂ permet de forcer l'exécution d'une certaine action par un processus donné ou de l'empêcher de faire une action.

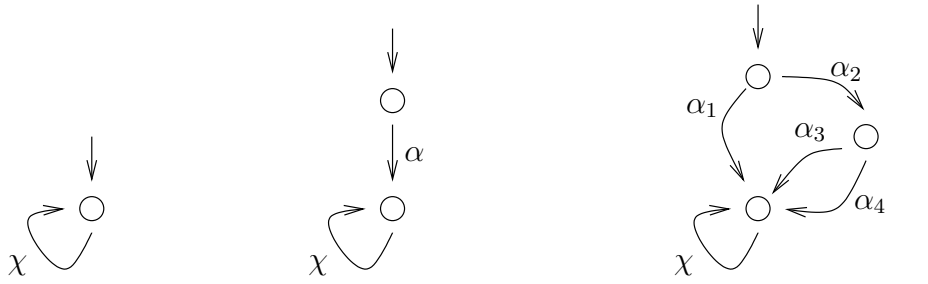
Par exemple :

$$\frac{P \xrightarrow{\alpha} Q \quad \alpha \notin H}{\partial_H(P) \xrightarrow{\alpha} \partial_H(Q)} \qquad \frac{(\forall \alpha \in \mathcal{A} \backslash H) P \xrightarrow{\alpha}}{\partial_H(P) \xrightarrow{\chi} \delta}$$

Sémantique des APT

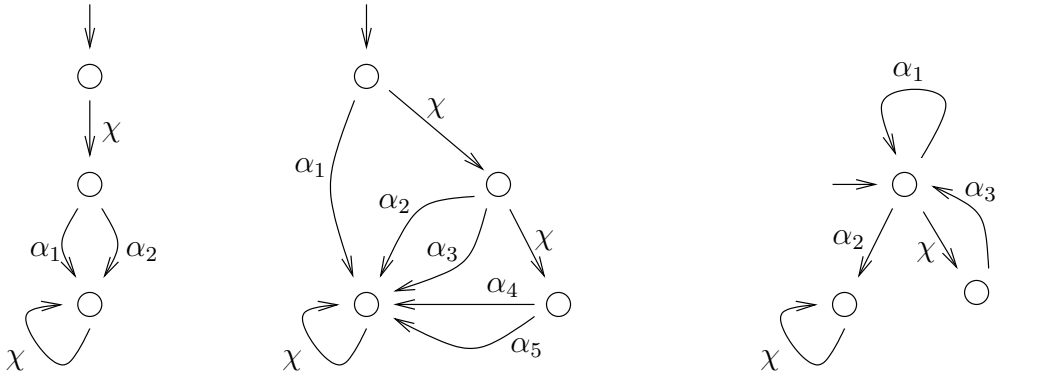
Tout comme la sémantique d'un automate temporisé est un LTS (expression de l'ensemble des exécutions temporisées de celui-ci, voir section 3.1.3), de même que la sémantique d'un réseau de Petri [Dia01] est, sous certaines conditions, le graphe de marquages accessibles (expression de l'ensemble des marquages possibles pour un réseau de Petri donné), la sémantique opérationnelle de l'APT est un LTS.

En effet, chaque règle donne l'évolution possible en terme d'actions d'un processus. Cet ensemble d'actions peut alors être mis en relation avec les transitions d'un état du LTS (les étiquettes de ces transitions seront les différentes actions possibles d'un processus).



(a) δ (b) $\alpha\delta$ (c) $\alpha_1\delta \oplus \alpha_2(\alpha_3\delta \oplus \alpha_4\delta)$

FIG. 3.6 – Exemples de transformations (séquentialités et alternatives).



(a) $\lfloor\delta\rfloor(\alpha_1\delta \oplus \alpha_2\delta)$ (b) $\lfloor\alpha_1\delta\rfloor\lfloor\alpha_2\delta \oplus \alpha_3\delta\rfloor(\alpha_4\delta \oplus \alpha_5\delta)$ (c) $recX.\lfloor\alpha_1 X \oplus \alpha_2\delta\rfloor(\alpha_3 X)$

FIG. 3.7 – Exemples de transformations (délai et récursivité).

Quelques exemples Sur la figure 3.6, l'élément bloquant δ peut être représenté par un état simple sur lequel il est possible d'exécuter un écoulement de temps (et rien d'autre).

De même pour la séquentialité de deux processus (figure 3.6.b). L'expression $\alpha\delta$ peut être symbolisée par un système à transitions étiquetées à deux états : le premier état possède une transition étiqueté α vers le deuxième état et le deuxième état est le système δ vu précédemment.

La figure 3.6.c montre comment le processus $\alpha_1\delta \oplus \alpha_2(\alpha_3\delta \oplus \alpha_4\delta)$ est traduite en LTS.

Enfin, les exemples de la figure 3.7 permettent de montrer la sémantique des opérateurs de *délais* et de *récursions*.

3.2.3 Méthodes de test

Cette section présente les méthodes de test, en comparaison aux méthodes de vérifications que nous aborderons dans la section 3.2.4.

Les différents types d'erreurs

Le test de logiciel ou de matériel est basé sur la détection d'erreurs non prévues dans la spécification, en envoyant des séquences de tests directement sur l'implémentation ou le système réel (et non son modèle). Il est ainsi possible de détecter les erreurs suivantes :

- *erreur de sortie* : le message reçu n'était pas attendu ou prévu dans la spécification,
- *erreur de transfert* : l'entrée n'est pas acceptée, contrairement à la spécification,
- *erreur de dépassement de délai* : attente plus longue que ce que prévoyait les délais limites de la spécification,
- *erreur d'entrée temporelle* : une entrée, bien que prévue à un certain moment, est arrivée hors délai.
- *erreur de sortie temporelle* : une sortie, bien que prévue à un certain moment, est arrivée hors délai.

Les méthodes de tests

Pour détecter ces erreurs, de nombreuses méthodes existent dans la littérature. Tout d'abord, les test peuvent être regroupés en deux catégories :

- les tests en *boîte blanche* : ces tests sont appliqués sur des systèmes dont l'implémentation est connue et accessible. Par exemple, les boucles d'un code source peuvent être testées (au niveau des invariants, des initialisations, des conditions d'arrêt, etc.).
- les tests en *boîte noire* : ces tests sont appliqués sur des systèmes dont l'implémentation n'est pas connue. Le seul moyen de vérifier qu'une opération est effectuée correctement est donc de regarder les informations retournées (par un affichage par exemple). L'état des composants internes du système ne peut être connu.

Voici quelques exemples de tests en boîte noire :

- *test de performance* : ces tests permettent de vérifier un certain nombre de paramètres liés aux performances du système, comme le débit, le temps de réponse, etc. Le système est placé en conditions réelles d'utilisation et un simulateur de connexion (par exemple) s'y connecte.
- *test de robustesse* : ces tests permettent de vérifier les réactions d'un système dans des conditions d'utilisations extrêmes. Ainsi, les tests permettent, par exemple, de vérifier le bon passage en mode dégradé si nécessaire.
- *test d'interopérabilité* : ces tests permettent de vérifier les échanges et communications d'un système avec son environnement. Ceci principalement en testant les types de codages, l'ordre des messages échangés, etc.
- *test de l'usager* : ces tests sont effectués au niveau de l'utilisateur qui va manipuler le système et vérifier s'il répond à ses attentes. On parle alors de *beta-tests* (très couramment utilisé dans l'environnement logiciel).
- *test de conformité* : ces tests vérifient si le comportement de l'implémentation est conforme à la spécification. Cela se passe en deux étapes. La première étape consiste à rechercher des propriétés significatives de la spécification permettant d'obtenir des *séquences de test*. La deuxième étape consiste à appliquer au système réel ces séquences de test pour obtenir un verdict : *pass* permet de dire que le comportement observable prévu dans la spécification a été rencontré sur le système réel après l'application d'une séquence de test, le verdict *inconcluant* ne permet pas de dire si oui ou non l'application est conforme à la spécification, et le verdict *fail* indique une erreur dans l'implémentation par rapport à la spécification.

Un des inconvénient du test est qu'il est difficilement exhaustif. En effet, comment être certain d'avoir exploré tous les cas ? Les méthodes de test sont cependant très utilisées, car ils permettent de tester le bon fonctionnement d'une grande partie du système ou du logiciel à moindre coût.

3.2.4 Méthodes de vérifications

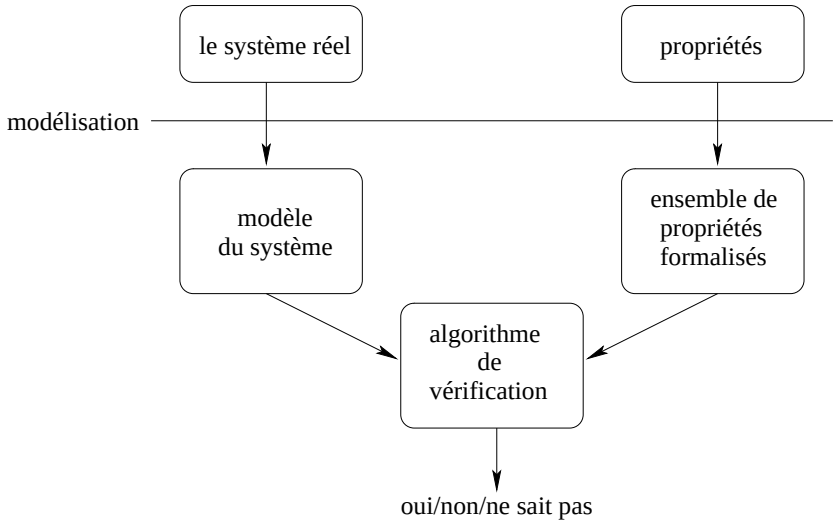


FIG. 3.8 – Principe de la vérification d'un système

Le principe de la vérification de système (ou de logiciel) est représenté figure 3.8. En fait, le système réel est modélisé, via un modèle et un niveau approprié suivant la propriété à vérifier. Ensuite, on extrait de la spécification une propriété à vérifier qui sera également modélisée (dans un modèle « compatible » avec le premier). Ensuite, le logiciel de vérification controlera, par un algorithme de vérification, si la propriété modélisée (ou l'ensemble de propriétés) est vérifiée ou non.

Le fait de modéliser le système pose le problème du degré de précision du modèle. Si le grain est trop fin, la complexité de la vérification est telle qu'il est difficile d'avoir une réponse dans un temps raisonnable, et si le grain est trop gros, il sera difficile (voir impossible) de détecter certains problèmes ou certains états précis du système. De même, la modélisation du système réel peut comporter des approximations voire même des erreurs, et alors, les propriétés étant vérifiées sur la modélisation du système réel, elles ne reflètent plus la réalité. Il faut donc utiliser des méthodes strictes et adaptées permettant de palier au mieux ces problèmes.

La suite de cette section va se concentrer sur deux méthodes de vérifications : le *model checking* et la *bisimulation*.

Le *model checking*

Le principe du *model checking* [CES83] repose tout d'abord sur la modélisation du système (par un automate par exemple). Ensuite, les propriétés à vérifier sur le système peuvent être exprimées dans une logique temporelle ou non (par exemple, LTL – *Linear Temporal Logic* [Eme90] –, CTL – *Computation Tree Logic* [EH85] – ou encore PLTL – *Propositional Linear Logic* [Pnu79] –). Bien

évidement, l'utilisation de telle logique pour exprimer les propriétés à vérifier permet une vérification informatique plus aisée.

Les modèles et les propriétés sont alors données en entrée à un algorithme de model checking qui vérifiera si oui ou non la propriété est vérifiée. Les algorithmes sont très différents suivant le langage utilisé pour l'expression des propriétés. Ainsi, dans le cas de CTL, on cherche à vérifier des propriétés sur des états de l'automate, et dans le cas de PLTL, la vérification se porte plus particulièrement sur la vérification de formules de chemin, c'est-à-dire concernant les exécutions du systèmes.

La mise en oeuvre de ces algorithmes peut se faire sous différents aspects que nous allons décrire.

La méthode dite *explicite complète* Cette méthode consiste à modéliser le système, sur lequel les algorithmes de vérifications vont être appliqués, par un seul et même système de transitions. Ce système de transitions peut devenir rapidement relativement énorme et devra en plus être entièrement stocké en mémoire.

La méthode dite *explicite partielle* Cette méthode est très similaire à la précédente, mais dans ce cas, le système de transitions n'est plus nécessairement complètement stocké en mémoire mais peut être généré à la volée pendant la vérification du code.

La méthode dite *implicite, condensé ou symbolique* Les méthodes de *model checking symbolique* (une introduction se trouve dans le livre [SBB+99]) permettent d'éviter ces cas d'explosion du nombre d'états. La représentation symbolique permet de condenser certaines représentations de sous-systèmes. Prenons l'exemple d'un programme comportant deux entiers (dont l'interval de définition est 0 à 255). L'automate de ce programme comporte plusieurs milliers d'états rien que pour ces deux entiers (les états sont de la forme $\langle q, v, v' \rangle$, où q est un état de contrôle, et v et v' sont les octets correspondants aux valeurs des deux entiers). La représentation symbolique consiste, par exemple, à représenter l'ensemble de ces états (liés à la représentation des deux entiers) par l'expression $\langle \star, \star, \star \rangle$. Ainsi, la notation symbolique $\langle q_2, 3, \star \rangle$ représente l'ensemble des triplets $\langle q, v, v' \rangle$ tel que $q = q_2$, $v = 3$ et v' quelconque. De même, cette méthode peut être étendue à un ensemble infini (par exemple l'ensemble $\mathbb{N}$ des entiers naturels). Il faut ensuite définir différentes opérations permettant de travailler sur ces états, comme par exemple calculer le complément de l'ensemble, comparer des ensembles deux à deux, etc.

Ces états symboliques peuvent être représentés à l'aide des BDD (*Diagrammes de Décision Binaires* [Bry86]), qui sont une représentation concise pour une représentation d'une donnée binaire. En effet, une donnée binaire peut être représenté par une formule de logique de premier ordre. Cette formule est ensuite dépliée en un *arbre de choix*, dont certains éléments sont en double ou n'apportent pas d'information supplémentaire (deux arcs étiquetés différemment sortent d'une feuille, mais arrivant sur le même sous-arbre par exemple). Ces éléments sont superflus donc supprimés et on obtient la représentation BDD. Quatre mots clés ressortent des BDD :

- *efficacité* : les opérations de bases (intersection, complément, comparaison, etc.) sont très rapides et la structure de donnée est très compacte,
- *simplicité* : les algorithmes et les structures de données sont relativement faciles à comprendre,
- *bonne adaptation* : il s'appliquent très bien aux données se modélisant par un automate,
- *généralité* : ils permettent de représenter de nombreux systèmes finis, et ils ne sont pas liés à un modèle précis d'automates.

La méthode dite *structurée* La méthode de *model checking* dite *structurée* consiste à modéliser le système par un ensemble de (petits) automates $\mathcal{A}_1, \ldots, \mathcal{A}_n$ représentant chacun un composant ou un comportement bien précis de ce système. Le produit synchronisé $\mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ permet alors d'obtenir l'automate $\mathcal{A}$, représentation de tous les états du système. Or ce produit peut générer un nombre potentiellement exponentiel d'états. Ce qui rend complexe, voire impossible de vérifier des propriétés sur certains systèmes ainsi modélisés.

La *simulation* et la *bisimulation*

La *simulation* et la *bisimulation* permettent de vérifier le comportement du système (dans sa globalité ou non) par comparaison à un comportement de référence. Ce comportement de référence peut-être un ensemble de propriétés issues de la spécification ou un système déjà vérifié.

La bisimulation est une relation portant sur l'ensemble des états du système. Deux systèmes sont dits bisimilaires s'il existe une relation entre les états des systèmes telle que deux états en relation se comportent exactement de la même manière. Vus de l'extérieur, les deux systèmes sont complètement indiscernables.

Définition 3.2.1 (Relation de bisimulation) Soit $S = (Q, s_0, \Sigma, \rightarrow)$ et $S' = (Q', s'_0, \Sigma, \rightarrow)$ deux systèmes de transitions. $\mathcal{R} \subseteq S \times S'$ est une relation de bisimulation entre S et S' si et seulement si $\mathcal{R}$ est totale et $\forall s \in S$, $s' \in S'$, $s\mathcal{R}s'$, on a :

$$\forall (s \xrightarrow{a} s_1) \Rightarrow \exists (s' \xrightarrow{a} s'_1) \wedge s_1 \mathcal{R} s'_1 \tag{3.1}$$

$$\forall (s' \xrightarrow{a} s'_1) \Rightarrow \exists (s \xrightarrow{a} s_1) \wedge s_1 \mathcal{R} s'_1 \tag{3.2}$$

S et S' sont en bisimulation (noté $S \approx S'$) si et seulement si il existe une relation de bisimulation $\mathcal{R}$ entre S et S' avec $s_0 \mathcal{R} s'_0$.

Remarque : Si seul l'équation 3.1 est vérifiée, alors on parle de **simulation** S' simule (le comportement) de S.

La définition de la relation de bisimulation peut être étendue à une équivalence entre les étiquettes a et $\tau^* a \tau^*$ (avec τ une action inobservable), ce qui permet de dire que les deux systèmes sont en **bisimulation faible** (à opposer à la *bisimulation* définie précédemment –définition 3.2.1–, appelée **bisimulation forte**).

3.2.5 Outils de vérification

Cette section présente les outils les plus connus implémentant des algorithmes de vérification, comme le *model checking*, présentés ci-dessus. Ces algorithmes utilisent les modèles présentés précédemment.

SPIN

SPIN est un outil permettant la simulation et la vérification d'algorithmes répartis, développé principalement aux Bell Labs et disponible sur Internet[1] [Hol03].

Pour étudier un système ou un algorithme avec l'outil SPIN, il faut d'abord le modéliser en utilisant le langage Promela. Ce langage ressemble au langage C auquel sont ajoutées des primitives

[1] `http://spinroot.com/spin/whatispin.html`

permettant la communication. Chaque processus du système est ainsi décrit, sans oublier les interactions entre ceux-ci. Les possibilités de communications sont les canaux de communications *fifo*, les prises de rendez-vous et les variables communes.

Il est possible d'exécuter SPIN dans deux modes différents. Le premier permet de simuler le système et de voir comment celui-ci s'exécute étape par étape. Le deuxième mode permet de vérifier certaines propriétés exprimées en PLTL sur le système, par un parcours exhaustif de l'ensemble des états atteignables du système.

SPIN est issu de l'étude des modèles d'automates communiquant via des canaux bornés et se limite aux système ayant un nombre fini d'états. Il a l'avantage de présenter de nombreuses optimisations concernant la réduction de l'espace d'états, comme la compression des états, l'utilisation d'ordres partiels, la vérification à la volée et l'utilisation de techniques de hachage.

IF

IF [BGM01] est un langage et une boîte à outils fournissant un environnement de validation ouvert pour les systèmes distribués, développé au sein du laboratoire VERIMAG[2]. IF est très utilisé dans les domaines de la vérification des protocoles de télécommunication et des systèmes embarqués ou temps-réels.

L'environnement de IF supporte différentes techniques de validations en utilisant plusieurs outils, allant de la simulation à la vérification de propriétés de logique, en ajoutant des cas de test et de la génération de code exécutable. Ces outils étant relativement complexes sont donc proposés comme outil indépendant et s'intégrant dans la plateforme IF. Pour cela, ils utilisent plusieurs niveaux d'intégration permettant l'ouverture à d'autres outils, mais également de travailler à différents niveaux : niveau de la modélisation, ou gestion de l'explosion du nombre d'états, etc... IF est donc une plateforme d'interconnexion entre ces différents outils.

Les différents outils fournis par IF sont :
- CADP regroupant un ensemble d'outils dédiés à la vérifications de spécification LOTOS (voir plus loin pour une description plus détaillée) ;
- Kronos, un logiciel de *model-checking* pour la vérification symbolique de formules Timed-CTL sur des automates temporisés (voir plus loin pour une description plus détaillée) ;
- TGV, un générateur de séquence de tests développé au-dessus des outils de CADP ;
- différents outils d'un autre niveau permettant de convertir les différents formats de fichiers entre eux.

Par exemple, la spécification du système à modéliser peut être écrite en SDL, UML, LOTOS ou encore Promela. Un premier outil permet de transformer cette spécification dans un langage intermédiaire propre à IF. Ensuite, l'ensemble d'outils permet par exemple de transformer ce langage intermédiaire en LOTOS, Promela ou autres. Et ainsi pouvoir bénéficier des apports des outils tel que CADP, Kronos, TGV, etc.

Kronos

Kronos [Yov97] est un ensemble d'outils développé à Grenoble, au sein du laboratoire VERIMAG[3]. Kronos permet l'analyse d'automates temporisés et est disponible sur Internet.

[2] `http://www-verimag.imag.fr/~async/IF/index.shtml.en`
[3] `http://www-verimag.imag.fr/TEMPORISE/kronos/`

Kronos est un logiciel de *model checking* utilisant la logique *Timed CTL*. Il vérifie des propriétés exprimées par une formule *Timed CTL* sur un automate temporisé, donnée sous forme textuelle. Il est également capable, si le système est décrit par plusieurs automates, de calculer le produit synchronisé correspondant.

Il est capable de vérifier des propriétés d'atteignabilité, mais également des propriétés de vivacité. En outre, c'est un outil en ligne de commande facilement invoquable par d'autres programmes.

L'outil *minim*, permet de générer un graphe de régions minimal à partir d'un automate temporisé au format Kronos. Il est alors possible de vérifier d'autres propriétés liées à ces modèles, en repoussant la limite d'explosion du nombre d'états.

UPPAAL

UPPAAL est issu d'une collaboration entre l'équipe BRICS (Aalbord University, au Danemark) et le Département of Computing Systems (Uppsala University, en Suède). L'outil est disponible, sous conditions, sur Internet[4].

UPPAAL [LPY97] et [BLL+98] est un ensemble d'outils permettant de modéliser, simuler et vérifier les systèmes temps réel. Il est particulièrement destiné au systèmes qui peuvent être modélisés comme un ensemble de processus non-déterministes avec des contrôles de structures finies et d'horloges à valeurs réelles, chaque processus pouvant communiquer par des actions de synchronisations (par exemple, les contrôleurs temps-réels, les protocoles de communications, ...).

Il est donc principalement basé sur le modèle des automates temporisés. Il a l'avantage de posséder une version texte et graphique, cette dernière permettant même la simulation pas à pas de systèmes dont le nombre d'états est trop grand pour appliquer les algorithmes de vérifications classiques.

UPPAAL utilise des méthodes très similaires à celles de Kronos. Par contre, il fourni une interface graphique, qui en plus de lancer les algorithmes de vérifications, permet également de simuler pas à pas des systèmes modélisés.

CADP

CADP (*CAESAR/ALDEBARAN Development Package*) [JHA+96] est un ensemble d'outils pour la validation de protocoles. Il fournit des outils de simulation et de vérification formelle. CADP est développé par l'équipe VASY de l'INRIA Rhone-Alpes et le laboratoire de VERIMAG. Il est disponible sur Internet[5].

Les principales fonctionnalités de CADP sont la compilation, la simulation, la vérification formelle et la génération de tests pour les descriptions de systèmes utilisant l'algèbre de processus LOTOS (voir section 3.2.1). Il transforme ces spécifications en code C, en passant par une étape intermédiaire de modélisation par des réseaux de Petri symboliques si nécessaire. Le code C dispose ensuite d'une API permettant son invocation dans le cadre d'une simulation, d'utilisation d'un outil de *model checking* ou même du développement d'un nouvel outil.

[4] `http://www.uppaal.com/`
[5] `http://www.inrialpes.fr/vasy/cadp.html`

3.3 La formalisation des services web

Cette section présente une partie des travaux réalisés sur la formalisation des services web. Et plus exactement en se focalisant principalement sur les approches orientées composition ou interaction entre plusieurs services web. Cette formalisation passe par une modélisation du service permettant ensuite la vérification de propriétés écrites dans une logique temporelle ou un autre formalisme. Ces travaux utilisent les éléments de modélisation, les langages formels et les méthodes de vérifications présentés ci-dessus.

3.3.1 Utilisation des systèmes de transitions

L'utilisation des systèmes de transitions pour modéliser directement le service web est une approche dite de bas niveau, et posant généralement un problème d'adaptation et de changement relativement rapidement en cas d'évolution de la technologie sous-jacente utilisée. Les approches, dites de plus haut niveau que nous aborderons par la suite, se ramènent généralement à un moment ou à un autre à une transformation de leur modèle en systèmes de transitions, mais de manière automatique ou semi-automatique : leur utilisation sur une technologie mouvante telle que les services web est à privilégier.

Approche systèmes de transitions

Les travaux [FBS04a, FBS04b] portent sur le développement d'un outil (nommé WSAT) permettant à l'utilisateur de vérifier des propriétés, exprimées en LTL, sur la composition de services.

La première étape de leur recherche, a été de définir une sémantique formelle. Cette sémantique consiste à l'écriture de patrons pour chaque élément BPEL. Ces patrons associent un système de transitions étiquetées (automates) à chacun de ces éléments. Ainsi, la modélisation globale d'un service BPEL consiste à mettre bout à bout les différents patrons, en utilisant des transitions internes (symbolisées par ϵ), et en modifiant au préalable les différentes étiquettes suivant le service considéré.

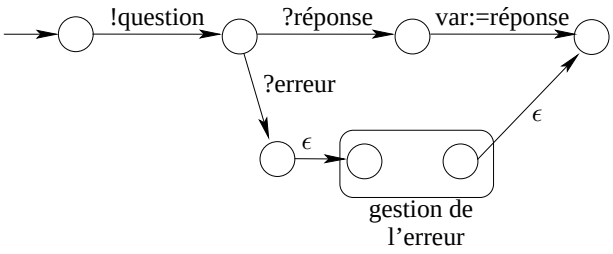


FIG. 3.9 – Exemple de traduction d'un processus $invoke$ (le processus consiste à envoyer une question et récevoir une réponse, avec possibilité d'obtenir une erreur) [FBS04a].

La traduction (voir un exemple de traduction pour le processus BPEL $invoke$ figure 3.9) montre plusieurs étapes et modélise une partie du comportement interne du service web (non observable par un client) : par exemple, l'action *invoke* de BPEL est traduite en un automate dont la première transition envoie le premier message de l'opération ; ensuite, la deuxième transition correspond à la réception de la réponse ; et, la troisième transition correspond à l'affectation à la variable devant recevoir cette réponse (cette transition correspond à comportement interne non observable de l'extérieur).

Comme une exception peut arriver pendant l'action *invoke*, ils ajoutent des transitions supplémentaires pour les gérer (une transition pour chaque faute possible à gérer).

Ce principe de construction fonctionne pour les éléments tels que *sequence*, *switch* et *while*. Mais l'élément *flow* est un peu particulier puisqu'il consiste à exécuter plusieurs processus BPEL en parallèle. La solution évoquée ici est de construire le produit cartésien de chaque sous-branche. Comme il existe la possibilité d'avoir des liens de contrôle de flux, ils associent à chaque transition une variable booléenne, et la sémantique des liens est reportée sur chaque transition d'activité BPEL par une garde. Aucun détail n'est donné sur la transformation des actions *scope* et *pick* de BPEL (faisant intervenir la notion d'écoulement de temps).

Ensuite, pour un service web composite donné, modélisé par des automates gardés comme décrit ci-dessous, ils génèrent une spécification Promela qui consiste en un ensemble de processus concurrents, un pour chaque automate. Chaque processus est associé à un canal de communication asynchrone stockant ces messages d'entrées.

Une fois cette spécification obtenue, l'outil permet de vérifier de propriétés LTL spécifiées par l'utilisateur, à l'aide de l'outil SPIN (voir section 3.2.5) sur la *conversation* obtenue par la composition. Cette conversation est obtenue par un *observateur virtuel* qui est supposé enregistrer toutes les séquences de messages envoyés par chacun des pairs impliqués dans la composition.

3.3.2 Utilisation des algèbres de processus

L'utilisation d'algèbres de processus a plusieurs avantages : tout d'abord, comme nous l'avons vu dans la section 3.2.2, elles sont très proches des structures de programmation d'un processus et permettent de modéliser facilement et directement un service web (qui est un processus). De plus, ces algèbres de processus sont transformables dans d'autres modèles (comme les LTS) permettant alors l'application de méthodes de vérification comme le *model checking*.

Approche processus à états finis

Les auteurs de [FUJ+03] ont proposé une sémantique formelle basée sur une transformation des services BPEL en FSP (*Finite State Processes* ou *processus à états finis*, une notation d'algèbres de processus) permettant ensuite de les valider en vérifiant un ensemble de propriétés sur ces expressions.

Chaque élément de BPEL est analysé puis un patron de transformation en FSP est proposé. Certains cas ne peuvent pas s'exprimer correctement en FSP : par exemple, l'élément *pick* de BPEL permet de gérer des exceptions et un temps maximal de déclenchement ; le patron FSP est alors un choix non déterministe entre les différentes branches et l'écoulement même du temps n'est alors plus représenté (mais son processus associé est alors repositionné au même titre que les exceptions).

L'architecture de vérification de modèle est basée sur deux éléments (voir figure 3.10), utilisant et/ou fournissant des éléments au vérificateur de propriétés :

– une première partie concerne la spécification : la spécification est décrite à l'aide des *message sequence charts* (*MSC*), depuis l'outil LTSA[6] (*Labelled Transition System Analyser*) est un outil de vérification pour systèmes concurents, qui permet également de visualiser un LTS correspondant à une expression FSP. L'outil permet ensuite de transformer la spécification en FSP, puis, ces FSP sont injectés dans le vérificateur de propriétés, accompagnés d'un ensemble de propriétés.

[6] `http://www.doc.ic.ac.uk/ltsa/`

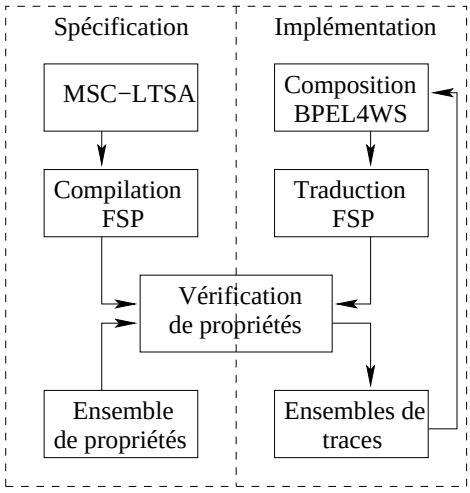


FIG. 3.10 – Architecture de la vérification [FUJ+03].

– la deuxième partie fournit au vérificateur le service BPEL transformé en FSP, et récupère en sortie un ensemble de traces.

C'est dans le dernier ensemble de traces que l'on peut alors détecter les propriétés erronées. Ces ensembles de traces ne prennent pas en compte les aspects temporels de BPEL.

Approche basée sur la notation CRESS

D'autres recherches s'articulent autour de LOTOS. Dans [Tur05], l'auteur utilise la notation CRESS (*Chisel Representation Employing Systematic Specification*) pour la formalisation des services web basés sur BPEL. Cette notation est ensuite transformée en LOTOS : cela permet alors une analyse avec des outils comme TOPO, LOLA et CADP. Encore une fois, les aspects temporels ne sont pas présents.

Mais la notation CRESS présente deux avantages : CRESS peut être transformé directement aussi bien dans un langage formel pour une analyse (LOTOS typiquement) mais aussi dans un langage permettant l'implémentation en vue du déploiement.

De plus, l'utilisation de CRESS qui a déjà été appliqué précédemment dans différents domaines est un avantage de maturité.

Approche basée sur la notation LOTOS

Un autre auteur a travaillé également sur la transformation de BPEL en LOTOS [Fer04]. Comme précédemment, LOTOS ne permettant pas de modéliser le temps, l'aspect temporel n'est pas pris en compte ; mais il est remplacé par un choix indéterministe.

Après la transformation en LOTOS, des algorithmes de vérifications habituels peuvent être exécutés : vérification par la logique temporelle, bisimulation, simulation, traces d'exécutions, etc.

Approche basée sur LOTOS et Timed CSP

Dans [SBS04], les auteurs ont fait une étude de cas pour permettre l'utilisation des outils conçus pour les algèbres de processus sur les services web. En effet, ils partent du constat suivant : les descrip-

tions BPEL sont essentiellement des description de processus, basées sur des primitives de descriptions de comportements et d'échanges de messages, que l'on retrouve dans les algèbres de processus.

Ainsi, ils annoncent quelques éléments de solutions dans les domaines de l'*orchestration* et la *chorégraphie*. Par exemple, l'utilisation de l'*orchestration*, pour le développement de services web, implique généralement l'utilisation de services web basés essentiellement sur un processus central qui envoie et reçoit des messages avec les autres partenaires. Il est alors possible de s'assurer de l'interaction correcte de ceux-ci grâce aux aspects formels des algèbres de processus.

Voici quelques solutions énoncées :

- Les données abstraites peuvent être manipulées grâce aux algèbres tels que LOTOS, ou avec certaines extensions et dans certains cas seulement, le recours au langage Promela.
- Les contraintes temporelles peuvent être représentées en Timed CSP.
- La mobilité (défini, dans ce cas, comme le fait d'échanger des données avec des clients arbitraires) peut être modélisée par le π-calcul.
- Les communications asynchrones sont généralement « simplifiées » pour arriver à un modèle synchrone. Dans les cas où cela n'est pas possible, on peut alors utiliser des model-checker comme SPIN.

3.3.3 Utilisation d'une approche par contrat ou interface

Les deux approches présentées ici sont plus d'ordre syntaxique. Elles ne vérifient pas le comportement global du processus métier, mais seulement le fait que le comportement ou la description (niveau syntaxique) de chaque fonction est identique.

Approche par contrat

L'approche par contrat utilisée dans [MM05] consiste à utiliser une adaptation à XML de CDL (*Contract Definition Language*). Un contrat ne décrit pas l'implémentation du service, mais seulement la sémantique de son exécution. Les éléments principaux du contrat sont les pré-conditions, les post-conditions et les invariants.

L'utilisation du contrat permet de palier le manque de description des propriétés non-fonctionnelles comme la dépendance, la sécurité, le temps-réel, la localisation, le prix, etc. La description de la connectivité est assurée par la description WSDL.

En parallèle, ils définissent une relation d'équivalence entre deux services. Cette notion d'équivalence est basée sur la syntaxe : si deux services, après avoir renommé les variables et les noms des opérations, sont identiques (très restrictif, car ils possèdent donc la même description), alors ces deux services sont dits équivalents. Cette équivalence est définie dans l'espoir d'une composition automatique, rendue possible en ajoutant des règles de recherches heuristiques (diminuant le nombre de possibilités), mais également dans le but d'une stratégie de composition automatique probabiliste et par apprentissage.

Ils se basent sur l'architecture des services web tel que l'énonce le W3C dans WS-Architecture. Ils indiquent clairement qu'une seule personne, ou une même institution, ne peut se lancer dans l'implémentation complète d'une architecture basée sur les services web et proposent donc une des approches possibles : une approche de composition avec un certain niveau d'automatisation.

Approche par l'utilisation des interfaces

Dans [BCH05], les auteurs ont défini une signature de services web (*web service signature*) qui consiste à associer une action a à un ensemble d'actions qui peuvent être invoquées par a (une action

étant un couple associant une méthode du service à un objet éventuel dû au retour de cette méthode).

Une action a est supportée par la signature S si $S(a)$ est définie. Une action a requiert une action a' dans S si $a' \in S(a)$. Une signature S requiert une action a' si au moins une action a requiert l'action a' dans S. Ensuite, ils définissent une signature bien formée (*well-formed-signature*) : pour chaque méthode supportée par la signature S, chaque action requise par S est supportée par S.

Grâce à cette signature ainsi définie, ils peuvent ensuite définir des propriétés permettant la compatibilité et la composition de services, et également la substitution (en travaillant sur les ensembles représentés par les signatures).

L'étape suivante consiste à définir ce qu'est une interface consistante, pour laquelle ils définissent les même propriétés que précédemment. Et par extension, ils définissent des interfaces de protocoles desquelles ils peuvent tirer les mêmes conclusions.

3.3.4 Utilisation des ontologies

Introduction

Les *ontologies* et le domaine du *web sémantique* se situent à un niveau différent de ce que nous venons de voir. En effet, les ontologies font partie des systèmes de raisonnements permettant, par exemple, de faire des déductions automatiques à partir d'une description formelle, autorisant ainsi le remplacement d'un service par un autre. Mais l'on remarque une approche similaire aux travaux précédents : la description du service web est modélisée dans un *méta-modèle*, pour ajouter des informations ou les mettre en valeur au vue d'un traitement informatisé. Ensuite, ce méta-modèle, possède une sémantique dans une autre représentation comme les systèmes de transitions, c'est-à-dire dans un modèle de niveau inférieur à ce méta-modèle : l'utilisation du modèle permet ensuite de lui appliquer directement des algorithmes de vérifications, simulations, etc. déjà connus dans la littérature.

Applications aux services web

Dans le cadre du web sémantique, il est courant d'utiliser des services décrits avec DAML-S. DAML-S [MBD+03] est un langage reposant sur XML et RDF(S), très utilisé dans le domaine de l'intelligence artificielle où tout doit être décrit le plus formellement possible pour permettre la réutilisation automatique des éléments ainsi décrits.

Les auteurs de [NM03] utilisent ce langage dans le but de décrire, simuler, composer, tester et vérifier les compositions de services web. Pour cela, ils ont défini une sémantique pour une partie des éléments de DAML-S (constructeurs de séquence, de concurrence, de choix, de test – if-then-else – et de boucle) en termes de logique du premier ordre (*situation calculus*).

À partir de cette sémantique, ils transforment une description DAML-S d'un service web en un réseau de Petri. Pour cela, ils associent à chaque axiome un réseau de Petri ; et grâce à l'ajout d'une structure de réseaux, ils définissent une composition de processus DAML-S ainsi modélisés.

Enfin, l'utilisation d'un modèle comme les réseaux de Petri, permet d'avoir des approches similaires aux travaux précédents pour l'application d'une simulation, de la validation ou même de la vérification des services.

3.4 Relation avec le travail de thèse

Au vu des différentes approches énoncées précédemment, il existe un réel attrait à la modélisation et la vérification de certaines propriétés sur les services web. L'axe de mon travail de thèse se situe

dans la continuité du travail réalisé au laboratoire LAMSADE, et dont le départ de la réflexion est présenté dans la première partie de la thèse de Tarak Melliti [Mel04]. Les deux dernières années sa thèse ont été menées en commun avec mon travail de stage de DEA [Ram03] et le début de ma propre thèse, ayant amené à un travail théorique commun sur la sémantique formelle des services web dans le cadre temps discret [HMMR04b] et une première réflexion sur la sémantique formelle en temps dense [HMMR04a]. Ce travail utilise les services web possédant une description comportementale par le langage XLANG (présenté en section 2.3.4) et propose une première réflexion amenant à la formalisation, en temps discret, des langages de ce type. Ces éléments de ces langages sont très proches des Algèbres de Processus et certains présentent même des notions de temps.

L'état de l'art L'état de l'art précédent sur les technologies des services web montre l'évolution rapide de ces différentes technologies. Les différentes avancés de ma thèse tiennent compte de cette mouvance et une approche la plus générique possible à été retenue face, par exemple, aux éléments à modéliser dans la sémantique formelle. Cette approche générique permet alors d'appliquer notre travail sur des langages ou technologies de nouvelles générations.

Chapitre 4 - Sémantique en temps discret Dans une première partie (voir chapitre 4), mon travail de thèse consiste à définir une sémantique formelle, en temps discret, pour chaque élément essentiel des langages de description comportementale. Cette sémantique formelle a été réalisée à base des Algèbres de Processus Temporisés de Sifakis [NS94]. Elle permet de décrire le fonctionnement d'un service web (composé par des séquences ou des boucles de plusieurs éléments de base du langage XLANG). Ensuite, cette description du comportement du service est modélisée par un TIOTS, proche des systèmes de transitions, correspondant à la sémantique des Algèbres de Processus Temporisés, voir section 3.2.2.

Ce TIOTS est alors un modèle de la partie serveur du service web. Grâce à l'écriture d'une relation d'interaction vérifiant la possibilité ou non de générer un modèle de client permettant l'invocation du service de façon « correcte » (c'est-à-dire que l'interaction ne présente pas d'ambiguïté, comme l'attente d'un message par les deux parties), il est alors possible de générer ce modèle représentant le client.

Chapitre 5 - Vérification d'une chorégraphie Puis, dans le chapitre 5, nous étendrons cette méthodologie temps discret à la vérification d'un ensemble de services web évoluant dans une chorégraphie. Ainsi, notre algorithme de vérification sera étendu à la vérification des différents partenaires de la chorégraphie, et un algorithme d'agrégation de partenaires y sera présenté.

Chapitre 6 - Sémantique en temps dense Le chapitre 6 adapte la sémantique présentée dans les deux chapitres précédents à la formalisation du temps dense. En effet, les modèles obtenus dans le cas temps discret peuvent présenter des explosions du nombre d'états, rendant difficile leur traitement informatique.

Chapitre 7 - Implémentation et mise en œuvre Enfin, le chapitre 7 présente les choix technologiques adoptés pour l'implémentation des différents algorithmes présentés précédemment. Le but est d'obtenir un client générique capable d'invoquer un service web dont seule la description comportementale (utilisant le langage BPEL par exemple) est connue.

Deuxième partie

Résultats théoriques et mise en œuvre

Chapitre 4

Sémantique en temps discret

Ce chapitre décrit la formalisation de langages de description comportementale et la modélisation de services web à l'aide des algèbres de processus de Sifakis. Ce travail s'inscrit dans la continuation du départ de la réflexion présenté dans le travail réalisé par Tarak Melliti et Serge Haddad [MH03]. Une partie de ce travail de recherche et une première implémentation des algorithmes (sans tenir compte de l'aspect générique de l'approche) ont été réalisées pendant mon travail de DEA [Ram03]. Enfin, une version plus aboutie a été présentée lors de la conférence ICEIS'04 [HMMR04b].

Nous aborderons l'approche formelle dans une première partie, et en particulier les différents choix concernant l'utilisation des algèbres de processus temporisés (ATP) pour la modélisation du service et l'emploi des systèmes de transition pour décrire notre sémantique. Dans une deuxième partie, nous mentionnerons les différentes catégories d'actions possibles et les conditions d'exclusions entre elles. La troisième partie sera consacrée à la description de notre algèbre de processus pour chacun des éléments de XLANG ou BPEL. Enfin la quatrième et dernière partie présentera nos algorithmes, concernant la relation d'interaction et la détection de l'ambiguïté, accompagnés d'exemples expliquant les différents cas et enfin l'algorithme de synthèse du client.

4.1 Approche formelle

Cette section explique le choix de la discrétisation du temps, suivi du choix des modèles formels à savoir les algèbres de processus temporisés de Sifakis [NS94] et les systèmes de transitions et plus particulièrement les TIOTS.

4.1.1 Discrétisation du temps

L'approche en *temps discret* consiste à discrétiser le temps, c'est-à-dire à modéliser par une action bien précise chaque unité de temps, ou *tic* d'horloge. Par extension, une durée est ainsi représentée par plusieurs *tics* d'horloge, soit plusieurs actions, symbolisant un écoulement de temps, consécutives.

Cette approche permet d'éviter de se confronter directement au problème de l'indéterminisme lié aux modèles de systèmes de transitions appliqués au temps dense (voir la partie consacrée à l'indéterminisme des automates temporisés section 3.1.3) Dans la suite de ce chapitre, nous verrons qu'un service web modélisé par notre formalisme temps discret peut :

– soit exécuter une action autre qu'un écoulement de temps (l'action s'exécute de façon « *instantanée* ») ;

– soit laisser passer le temps (pour une unité de temps, correspondant à un *tic* d'horloge).

De plus, nous utilisons comme modèle formel les algèbres de processus temporisés (voir les explications de ce choix en section 4.1.2). Ce modèle présente également l'aspect discret du temps (symbolisant les *tic* d'horloge par le symbole χ).

Nous avons ensuite travaillé sur une approche temps continu ou temps dense, dont les motivations sont expliquées au cours du chapitre 6.

4.1.2 Algèbres de processus temporisés de Sifakis

Comme nous l'avons vu dans l'état de l'art sur la formalisation des services web (section 3.3), la notion de processus des *algèbres de processus temporisés* est très similaire aux différentes opérations présentes dans les langages de descriptions comportementales de services web comme XLANG (section 2.3.4) et BPEL (section 2.3.5). En effet, les algèbres de processus temporisés et les langages de descriptions de service web ont en commun la notion de séquentialité, la notion d'exécution en parallèle et la gestion du temps (ici, temps discret pour les algèbres).

Remarque : historiquement, l'approche a été réalisée seulement pour XLANG, le langage BPEL ayant été introduit après. Nous avons adapté notre approche à BPEL par la suite, grâce à une approche initiale basée sur la généricité : ainsi, la méthode énoncée ici peut s'étendre à d'autres langages par adaptation ou redéfinition.

Une opération de XLANG ou BPEL est définie par un ensemble de règles dont la structure est la suivante (le processus P est constitué des sous processus P_i, $i \in I$, voir section 3.2.2) :

$$\frac{\bigwedge_{i \in I} [\neg] P_i \xrightarrow{a_j} P_i'}{P \xrightarrow{a_{j'}} P'}$$

Cet ensemble de règles pour une opération donnée permet de définir, suivant les actions a_j que le processus P actuel peut réaliser, l'action $a_{j'}$ qui sera finalement exécutée. Les règles sont donc en concordances avec les différentes règles d'exclusion entre les actions que nous définirons dans la partie 4.2.

Remarque : si aucune condition n'est émise sur l'exécution de l'action (c'est-à-dire si l'expression booléenne de la partie haute est à « *vrai* »), alors, la partie haute peut être omise, et la règle sera constituée seulement de la partie basse : $P \xrightarrow{a_{j'}} P'$ indique que le processus P se transforme en processus P' par l'action $a_{j'}$, quelques soient les conditions dans lesquelles ce processus se trouve.

4.1.3 Système de transitions : TIOTS

La sémantique des algèbres de processus étant un système de transitions (voir section 3.2.2), l'utilisation de ces derniers était la plus appropriée pour définir celle de nos services. De plus, de nombreuses méthodes et outils ont été développés pour travailler sur les systèmes modélisés à partir de ce formalisme.

Un service web communique avec un client par le biais d'échanges de messages : donc des envois et des réceptions de messages. Les services web possédant une description comportementale gèrent également la notion du temps (temps maximal d'exécution ou d'arrivée d'un message, etc...). Nous avons alors fait le choix d'utiliser les TIOTS (description du modèle en section 3.1.2) qui sont un

raffinement des systèmes de transitions permettant de modéliser les entrées/sorties, le temps et les actions internes.

4.2 Les actions d'un service web

Cette section décrit les différentes actions qu'un service web peut réaliser, ainsi que leurs particularités. Seules les actions qui seront modélisées dans le TIOTS vont être présentées ici, c'est-à-dire les actions « visibles » dans l'interaction service-client.

4.2.1 Les différentes actions

Les étiquettes de transitions du TIOTS correspondent aux différentes actions qu'un service peut réaliser durant son interaction avec le client :

- l'échange d'un message m, également appelé événement, est modélisé soit comme une *réception*, dans ce cas, le symbole $?m$ est utilisé, soit comme un *envoi* et dans ce cas, le symbole $!m$ est utilisé. Le caractère étoile '$*$' peut être interprété à la fois comme un envoi et/ou une réception et peut donc être remplacé par les symboles '?' ou '!'.
- il est nécessaire de modéliser certains *comportements inobservables* du service (voir section 4.2.2), le symbole utilisé est τ.
- en concordance avec notre approche *temps discret* et la définition du temps dans les APT, l'action symbolisant un *écoulement d'une unité de temps* (un « *tic* » d'horloge) est symbolisé par χ.
- un service web peut générer une *exception*, qui peut être interceptée ou non. En cas de non interception (voir les processus scope et pick), le symbole e apparaît.
- enfin, pour permettre la détection correcte de la fin d'une exécution (voir section 4.2.4), l'*action de terminaison* est utilisée ; le symbole correspondant est $\surd$.

4.2.2 Modélisation du comportement inobservable

Les *comportements inobservables* correspondent par exemple à l'évaluation d'une condition booléenne : aucun message n'est échangé entre le service et le client, mais le service évolue de façon significative suivant le résultat de l'évaluation de la condition. En effet, suivant cette évaluation, les actions exécutées après ne seront pas les mêmes (par exemple, dans le cas d'une boucle *while*, le corps de la boucle est exécuté ou non). Il est donc nécessaire de modéliser ce comportement par le symbole τ pour séparer les deux évolutions possibles.

Cette action inobservable ou silencieuse a une particularité : elle ne peut être en concurrence avec une autre action de type différent. Si un état du système a la possibilité d'exécuter une action interne, alors ce sera la seule action qu'il pourra réaliser.

Comme nous le verrons dans la relation d'interaction et la génération du client, ceci n'est pas restrictif car, comme dans la sémantique des APT et des TIOTS, les actions autres que le passage de temps s'exécutent de façon immédiate, c'est-à-dire sans laisser passer le temps. Dès que le système arrive sur un de ces états, le choix de la transition à franchir est tout de suite fait et l'exécution passe à l'état suivant, qui lui peut à nouveau exécuter d'autres actions (comme le passage du temps ou l'envoi d'un message), sauf s'il exécute à nouveau une action interne.

4.2.3 Écoulement du temps et les différentes actions

Comme la syntaxe des APT l'exige (voir section 3.2.2), toutes les actions autres que χ ne permettent pas l'écoulement du temps. C'est-à-dire qu'elles s'exécutent de façon « instantanée ».

Ceci est bien évidement une « imprécision » dans le modèle par rapport à un système réel asynchrone (cas des services web) : en effet, le temps de propagation d'un message, par exemple, n'est pas pris en compte.

La modélisation en temps discret, ici présentée, se rapproche donc des systèmes synchrones et des imprécisions sont faites quant au modèle réel des services web. Une approche plus fidèle est faite par la modélisation temps dense. Cependant, dans cette deuxième modélisation, la prise en compte des temps de communications reste un des objectifs futurs à réaliser.

4.2.4 Détection de la terminaison

Même si l'étape correspondant à la terminaison n'est pas visible au niveau des échanges de messages entre le client et le serveur, il est totalement indispensable, aussi bien pour le serveur que le client, de permettre la détection de cette terminaison.

En effet, il suffit d'imaginer ce que pourrait produire l'incohérence suivante, lors d'une transaction banquaire par exemple :

- le serveur, après avoir enregistré une transaction, attend une confirmation de celle-ci de la part du client : le serveur termine son exécution et enregistre la transaction après avoir reçu cette confirmation ;
- le client, ayant un comportement erroné par rapport à la description précédente, a la possibilité de confirmer ou de terminer son exécution : il décide alors de terminer son exécution immédiatement après la première étape d'enregistrement de la transaction.

Le serveur va donc rester dans un état d'attente de la confirmation, et, s'il est bien programmé, atteindra un temps maximal d'exécution (*time-out*) et annulera la première transaction. Dans les deux cas (attente et annulation), l'état du serveur n'est pas ce que le client espérait, donc l'interaction entre le service et le client a échoué (au niveau comportemental).

Il est donc indispensable que l'état du système (aussi bien côté client que côté serveur), présentant la possibilité d'exécuter l'action de terminaison, ne puisse pas être en concurrence avec d'autres actions : de cette façon, le serveur ne pourra pas prendre une direction de l'exécution différente de celle du client ou inversement.

4.3 Sémantique des opérations

Comme nous l'avons annoncé un peu plus haut, nous ne présenterons ici que la partie BPEL, mais historiquement, la première approche et implémentation de notre plateforme était basée sur XLANG. Les opérateurs XLANG sont très proches de ceux de BPEL et la sémantique des uns peut se déduire de celles des autres. Le tableau 4.1 montre la correspondance entre les opérateurs des deux langages (voir sections 2.3.4 et 2.3.5 pour les détails exactes des opérateurs).

Ce changement de langage montre bien l'intérêt de notre approche générique, c'est-à-dire factoriser les mécanismes fondamentaux tels que l'envoi ou la réception de messages, l'opérateur de séquence ou de boucle, etc., accepter une variation de la syntaxe et définir plus aisément la sémantique du langage étudié.

XLANG	BPEL4WS
empty	empty
sequence	sequence
action	receive, reply, (invoke)
swicth	switch
while	while
pick	pick
all	flow
context	scope

FIG. 4.1 – Comparaison des opérateurs XLANG et BPEL

Cette approche générique permet de faire évoluer indépendamment les éléments suivants :
– les *langages de descriptions de services* : passage de XLANG à BPEL par exemple. Il est même possible d'imaginer s'affranchir des services web et modéliser des systèmes présentant les mêmes caractéristiques.
– les *règles de la sémantique* : elles peuvent être ajustées, affinées, ou même modifiées complètement suivant ce que l'on désire mettre en évidence. La section 7.2.2 montre que les règles de traductions ne se trouvent pas dans le code de l'outil de modélisation développé pendant mon travail de thèse, mais dans un fichier permettant ainsi la modification à loisir sans même une recompilation de l'outil.

4.3.1 Les règles pour les éléments de base

Cette section présente l'ensemble des règles de notre sémantique pour les éléments de base (appelé processus) du langage BPEL. Pour les informations concernant la syntaxe XML, se reporter à la section 2.3.5.

Le processus time

L'élément le plus basique est le processus time : il peut seulement laisser passer le temps. Il correspond donc à l'exécution d'une infinité de « *tick* » d'horloge χ des APT :

$$\mathrm{time} \xrightarrow{\chi} \mathrm{time}$$

Le processus empty

Un autre élément de base de notre sémantique est le processus empty. C'est un processus quelque peu particulier : il ne peut que se terminer. La seule action qu'il peut réellement faire est l'action $\surd$. Il devient alors le processus 0 (ou $null$), un processus n'ayant plus la possibilité de s'exécuter (par analogie, on peut le comparer à l'état *zombie* des processus Unix).

$$\mathrm{empty} \xrightarrow{\surd} 0$$

Le processus throw

Le dernier élément de base est le processus throw qui consiste à déclencher une exception e, parmi l'ensemble des exceptions E_X associé au processus métier global. Cette exception peut ensuite être interceptée par un processus, le gérant (voir le processus scope en section 4.3.4).

$$\forall\, e \in E_X,\ \mathrm{throw}[e] \xrightarrow{e} 0$$

4.3.2 Les règles pour les éléments basés sur les messages

Les processus receive **et** reply

Le processus receive correspond à l'opération *input* de WSDL, c'est-à-dire la réception d'un message de type m.

De la même façon, le processus reply correspond à l'opération *output* de WSDL, c'est-à-dire l'envoi d'un message de type m.

Le langage WSDL permet également une réception suivie d'un envoi de message et inversement. Cette opération est en fait composée de deux sous-opérations s'exécutant séquentiellement. Il faut donc, dans ce cas, utiliser le processus sequence décrit en section 4.3.3 dont le corps consistera à exécuter les processus receive et reply.

Remarque : ici, nous avons présenté la séparation envoi/réception qui n'existe que sous BPEL, XLANG utilise qu'un seul élément pour faire les deux : action(correspondant à une opération WSDL).

De plus, nous considérons que l'action envoi ou réception peut prendre un certain temps (à cause de conditions internes au service ou de la propagation du message), donc l'action d'envoi ou de réception réelle permet l'écoulement du temps.

$$*o[m] \xrightarrow{*m} \mathrm{empty} \quad \text{avec } * \in \{?, !\}$$

$$*o[m] \xrightarrow{\chi} *o[m]$$

Le processus invoke

Le processus invoke est quelque peu particulier. En effet, il consiste à invoquer une opération d'un sous-service à partir d'un service. Il n'intervient donc pas dans les échanges entre le service et le client (voir figure 4.2). Ce processus n'a donc tout simplement pas besoin d'être modélisé dans l'interaction client/service que nous modélisons ici.

Une approche prenant en compte les sous-services sera abordée dans le chapitre 5 et nous devrons alors modéliser ce processus (section 5.3.1).

4.3.3 Les règles pour les éléments de contrôle structuré

Cette section présente les éléments de contrôle structuré. Ils correspondent aux structures traditionnelles de la programmation : *séquence*, *boucle* et *choix*.

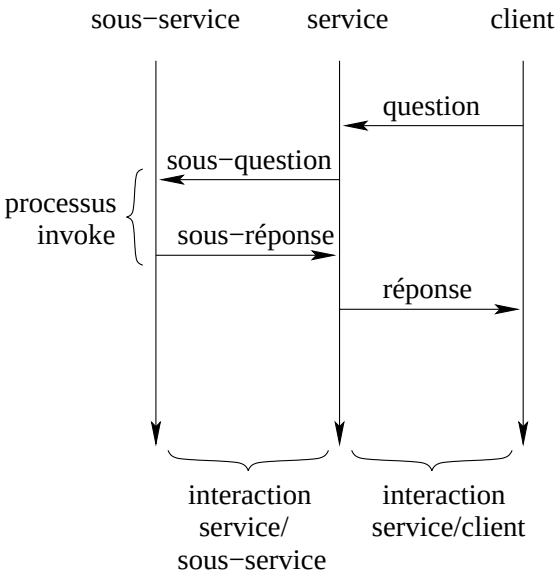


FIG. 4.2 – Diagramme des messages échangés lors de l'utilisation d'un processus invoke par le service.

Le processus sequence **(;)**

Le processus sequence consiste à exécuter deux sous-processus P et Q séquentiellement. Il est symbolisé par ';'. L'opérateur est associatif, donc nous pouvons restreindre le nombre de sous-processus à deux.

Le principe général est le suivant : $P;Q$ laisse exécuter le processus P jusqu'à ce qu'il se termine, et passe ensuite la main au processus Q.

La première sémantique Une première solution consiste à dire que le passage du processus P à Q se fait de façon silencieuse, ce passage est alors symbolisé par l'action τ. Ce qui nous donne alors les deux règles suivantes.

$$\forall\, a \neq \surd,\ \frac{P \xrightarrow{a} P' \quad \wedge \quad \neg P \xrightarrow{\surd} P''}{P\ ;\ Q \xrightarrow{a} P'\ ;\ Q} \quad \text{et} \quad \frac{P \xrightarrow{\surd} P'}{P\ ;\ Q \xrightarrow{\tau} Q}$$

La deuxième sémantique Nous avons ensuite proposé une deuxième solution. En effet, le fait de modéliser explicitement par l'action τ le passage du processus P à Q n'apporte aucune information au vu de la sémantique imposée pour l'action τ et augmente le nombre de transitions et d'états. Nous avons donc supprimé cette action supplémentaire : lorsque le processus P se termine, il passe directement la main au processus Q, ce changement est traduit par l'exécution de la première action du processus Q.

$$\forall\, a \neq \surd,\ \frac{P \xrightarrow{a} P'}{P\ ;\ Q \xrightarrow{a} P'\ ;\ Q} \quad \text{et} \quad \forall\, a,\ \frac{P \xrightarrow{\surd} P' \quad \wedge \quad Q \xrightarrow{a} Q'}{P\ ;\ Q \xrightarrow{a} Q'}$$

C'est cette deuxième solution que nous adoptons pour la suite.

Le processus switch

Le processus $\mathrm{switch}[\{P_i\}_{i\in I}]$ consiste à exécuter, après un choix interne au service, un des processus de l'ensemble $\{P_i\}$. Au niveau du processus métier, ce choix est déterminé par une véritable condition booléenne. Mais d'un point de vue extérieur (donc la partie que nous voulons modéliser), ce choix n'est pas visible puisqu'il ne correspond à aucun échange de messages : c'est donc un *choix indéterministe*, que nous modéliserons par une transition τ pour chaque processus P_i.

$$\forall\, i \in I,\ \mathrm{switch}[\{P_i \mid i \in I\}] \xrightarrow{\tau} P_i$$

Le processus while

Le processus $\mathrm{while}[P]$ exécute en boucle le sous-processus P tant qu'une condition interne au processus métier est vérifiée. Tout comme dans le cas du processus switch, l'évaluation de cette condition n'est pas visible depuis l'extérieur et sera donc modélisée par une action τ.

Donc, deux possibilités d'exécution se présentent pour le processus while :
– soit il exécute le corps de la boucle (sur une évaluation interne à *vrai* de la condition). Il faut alors exécuter séquentiellement le sous-processus P suivi de la boucle while à nouveau.

$$\mathrm{while}[P] \xrightarrow{\tau} P\ ;\ \mathrm{while}[P]$$

– soit la condition est évaluée de façon interne à *faux*, alors, le processus while termine son exécution en devenant le processus empty.

$$\mathrm{while}[P] \xrightarrow{\tau} \mathrm{empty}$$

4.3.4 Les règles pour les éléments de contrôle évolué

Ici sont présentés les trois processus les plus complexes : flow, scope et pick.

Le processus flow

Le processus $\mathrm{flow}[\{P_i\}_{i\in I}\}]$ exécute simultanément l'ensemble des processus $\{P_i\}$.

Remarque : ce processus présente une légère différence entre les deux langages de description comportementale étudiés. XLANG ne présente pas de mécanisme permettant la *synchronisation temporelle* des différents processus, contrairement à BPEL. Ici, dans la sémantique adoptée, nous n'indiquons aucun mécanisme de synchronisation. Par contre, nous verrons dans la section 7.2.3 qu'une implémentation des mécanismes de BPEL a été réalisée.

L'exécution parallèle des différents processus est similaire au mécanisme système *fork-join*, c'est-à-dire que les processus s'exécutent « indépendamment » et se synchronisent sur leur terminaison. L'indépendance de leur exécution a deux exceptions :
– l'exécution d'un écoulement de temps se fait de manière simultanée à tous les processus (sémantique des APT) ;
– la terminaison s'exécute également de manière simultanée. Une fois cette synchronisation effectuée, le processus flow devient le processus *null* (0).

Actions individuelles : ces règles consistent à l'exécution des actions concernant les messages, de l'action interne et d'une exception.

$$\forall\, a \in E_X \cup \{\tau\},\ \frac{\exists\, j \in I,\ P_j \xrightarrow{a} P'}{\text{flow}[\{P_i \,|\, i \in I\}] \xrightarrow{a} \text{flow}[\{P_i \,|\, i \in I \backslash \{j\}\} \cup \{P'\}]}$$

$$\forall\, m \in M,\ \frac{\exists\, j \in I,\ P_j \xrightarrow{*m} P' \text{ et } \forall\, i \neq j,\ \forall\, a \in E_X \cup \{\tau\},\ \neg\, \exists k \in I,\ (P_k \xrightarrow{a})}{\text{flow}[\{P_i \,|\, i \in I\}] \xrightarrow{*m} \text{flow}[\{P_i \,|\, i \in I \backslash \{j\}\} \cup \{P'\}]}$$

Action de terminaison : cette règle gère la synchronisation des terminaisons entre l'ensemble des processus.

$$\frac{\forall\, i \in I,\ P_i \xrightarrow{\surd} P'_i}{\text{flow}[\{P_i \,|\, i \in I\}] \xrightarrow{\surd} 0}$$

Action d'écoulement du temps : cette règle gère la synchronisation de l'écoulement de temps entre l'ensemble des processus.

$$\frac{\exists J \neq \emptyset,\ J \subseteq I,\ \forall\, i \in J,\ P_i \xrightarrow{\chi} P'_i \text{ et } \forall\, i \in I \backslash J,\ P_i \xrightarrow{\surd}}{\text{flow}[\{P_i\}_{i \in I}] \xrightarrow{\chi} \text{flow}[\{P'_i\}_{i \in J} \cup \{P_i\}_{i \in I \backslash J}]}$$

Le processus scope

Soit $M_I = \{m_i \,|\, i \in I\}$ un ensemble de messages, soit $E_J = \{e_j \,|\, j \in J\}$ un ensemble d'exceptions, et soit E_X l'ensemble des exceptions ($E_J \subseteq E_X$).

Le processus $\text{scope}(P, E)$ est défini par un processus P et l'expression

$$E = [\{(m_i, P_i) \,|\, i \in I\}, (d, Q), \{(e_j, R_j) \,|\, j \in J\}]$$

Le processus scope consiste en l'exécution du processus P telle que :
- l'exécution de P doit prendre au plus d unités de temps. Si ce n'est pas le cas, alors le processus Q est exécuté ;
- si une exception $e_j \in E_J$ est déclenchée, alors le processus R_j associé est exécuté ;
- si un message $m_i \in M_I$ non intercepté par P arrive, alors le processus P_i associé est exécuté.

Exécution du processus P : premier cas, si P se termine, alors le scope se termine également.

$$\frac{P \xrightarrow{\surd}}{\text{scope}(P, E) \xrightarrow{\surd} 0}$$

Deuxième cas, si P peut exécuter une action (autre que l'écoulement de temps, l'action de terminaison, une exception ou la réception d'un message de M_I), alors le processus scope exécute cette action.

$$\forall\, a \notin \{\chi, \surd\} \cup E_X \cup M_I \quad \frac{P \xrightarrow{a} P'}{\text{scope}(P, E) \xrightarrow{a} \text{scope}(P', E)}$$

Écoulement du temps : premier cas, le processus P a encore le temps de s'exécuter (E^a désigne l'entité E dans laquelle d est remplacé par a).

$$d > 1,\ \frac{P \xrightarrow{\chi} P' \text{ et } \forall\, a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\mathrm{scope}(P, E^d) \xrightarrow{\chi} \mathrm{scope}(P, E^{d-1})}$$

Deuxième cas, le temps d'exécution imparti à P est écoulé, la procédure de *timeout* est déclenchée.

$$\frac{P \xrightarrow{\chi} P' \text{ et } \forall\, a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\mathrm{scope}(P, E^1) \xrightarrow{\chi} Q}$$

Réception d'un message $m_i \in M_I$: en recevant le message m_i, l'événement est traité et le processus scope arrêté.

$$\forall\, i \in I,\ \frac{\forall\, a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\mathrm{scope}(P, E) \xrightarrow{?m_i} P_i}$$

Traitement des exceptions : premier cas, l'exception e_j est prévue, le processus scope l'intercepte et exécute le processus R_j associé :

$$\forall\, j \in E_J,\ \frac{P \xrightarrow{e_j}}{\mathrm{scope}(P, E) \xrightarrow{\tau} R_j}$$

Deuxième cas, l'exception e n'est pas prévue, le processus scope passe dans un état de terminaison (processus *zombie*). Cette exception devrait être interceptée à un niveau plus haut (dans le cas d'un processus scope contenu dans un autre processus scope par exemple).

$$\forall\, e \notin E_J,\ \frac{P \xrightarrow{e}}{\mathrm{scope}(P, E) \xrightarrow{e} 0}$$

Le processus pick

Le processus pick est considéré comme un cas particulier du processus scope : c'est un processus scope dont le processus principal P est défini comme étant le processus time.

$$\mathrm{pick}[E] = \mathrm{scope}(\mathrm{time}, E)$$

$$\text{avec}\quad E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}$$

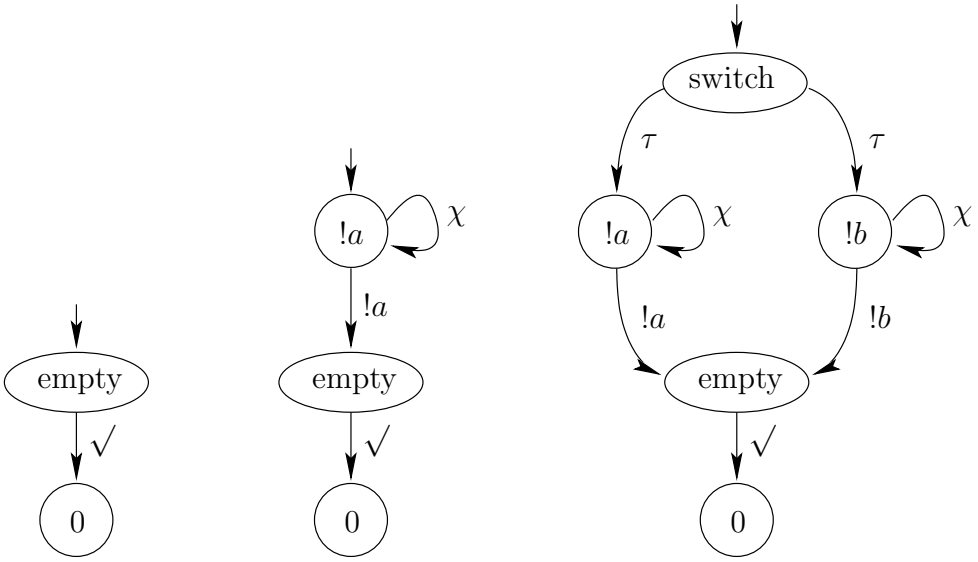
4.3.5 Génération du TIOTS

La génération du TIOTS pour un processus métier donné commence par la modélisation de son état initial. Cet état initial représente la globalité du processus métier (tel qu'il est décrit dans le fichier de description par BPEL ou XLANG). Pour cela, il faut obtenir le fichier WSDL de description du service, et le fichier de description comportementale (extension du fichier WSDL dans le cadre de XLANG et un deuxième fichier dans le cadre de BPEL). En plus de ces deux descriptions, il peut être nécessaire d'obtenir la description WSDL des sous-services invoqués. Une fois ces fichiers analysés,

il est alors possible de générer l'expression correspondant au processus métier dans sa globalité, puis de générer l'état initial.

Ensuite, l'ensemble des actions possibles pour ce processus est obtenu en analysant les règles décrites ci-dessus. Chaque action a est traduite dans le TIOTS par des transitions sortantes (de l'état initial lors de la première étape, et d'un autre état par la suite), étiquetées par l'action a et dont l'état d'arrivée correspond à l'état associé au processus dérivé en exécutant cette action.

En appliquant ce mécanisme récursivement à tous les états du TIOTS ainsi créés, on obtient le TIOTS complet représentant l'ensemble des possibilités d'exécution du processus métier (voir la figure 4.3 pour des exemples).



(a) empty (b) $!a$ (c) switch$\{!a, !b\}$

FIG. 4.3 – Exemples de processus et leur TIOTS

4.4 Relation d'interaction et synthèse de client

Maintenant que nous avons obtenu un modèle pour la partie serveur, nous allons définir notre relation d'interaction, permettant de vérifier formellement si un client peut interagir de façon adaptée avec notre service. Et, si c'est le cas, générer le modèle du client. Nous expliquerons également les exemples d'ambiguïté empêchant cette génération. Ce modèle nous servira, dans la suite, de fil conducteur pour écrire un client générique pour un service donné.

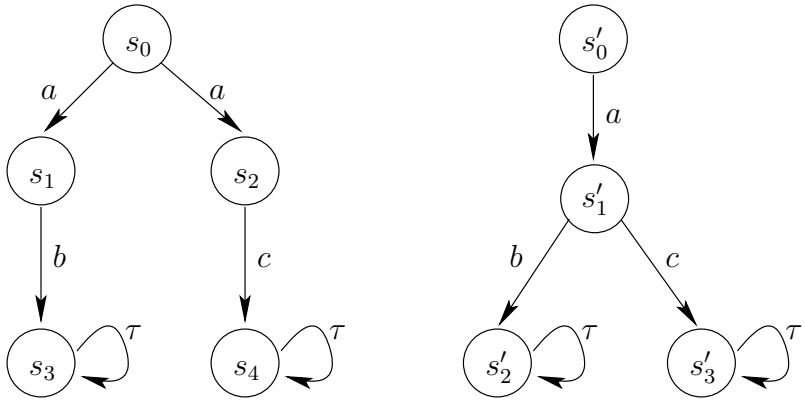
4.4.1 Relation d'interaction

L'idée principale pour la *relation d'interaction* est de proposer un algorithme permettant de vérifier la possibilité d'interagir de façon adaptée avec un service web possédant/proposant une description comportementale de son fonctionnement et disponible sur Internet (dont nous n'avons pas la maîtrise de l'implémentation, nous pouvons juste invoquer le service).

Presque une bisimulation

La bisimulation, comme nous l'avons vue dans la section 3.2.4, vérifie que deux systèmes sont en relation totale. Or ici, nous voulons une relation plus faible :

- tout d'abord, les événements internes, symbolisés par les transactions τ, ne sont présents que sur le modèle du service web côté serveur ; le client n'ayant pas d'action à faire pendant ces actions locales au service (donc sans échange de message entre le client et le service). Cela rejoindrait l'idée de la *bisimulation faible* ;
- par contre, le deuxième élément, indiquant que notre relation d'interaction est différente d'une bisimulation, est que nous acceptons le fait que le client et le service peuvent prendre deux chemins légèrement différents mais aboutissant sur des états ayant le même ensemble de messages à envoyer (voir figure 4.4).



(a) Automate S (b) Automate S'

FIG. 4.4 – Automates non bisimilaires

Définitions

Comme dans de nombreux travaux basés sur les systèmes de transitions, nous définissons une relation basée sur les transitions observables (chemins) entre les états :

- $s \stackrel{a}{\Rightarrow} s'$ si et seulement si $s \stackrel{\tau^* a \tau^*}{\longrightarrow} s'$: c'est-à-dire, si un chemin est défini par la suite d'actions $\tau^* a \tau^*$, alors, pour mettre en évidence la seule action observable a (envoi, réception de message ou écoulement de temps), l'écriture $\stackrel{a}{\Rightarrow}$ est utilisée ;
- $s \stackrel{\epsilon}{\Rightarrow} s'$ si et seulement si $s \stackrel{\tau^*}{\rightarrow} s'$: de même, si le chemin est défini par la suite d'action τ^*, l'écriture $\stackrel{\epsilon}{\Rightarrow}$ est utilisée, montrant qu'aucune action observable n'a été réalisée.

Remarque : nous supposons que les événements correspondant à des exceptions sont des événements inobservables, ceci est traduit dans les règles définies plus haut.

Présentation informelle

Nous allons décrire la *relation d'interaction*, entre un service et son client, de façon informelle. Tout d'abord, la description et les impératifs globaux qui doivent être vérifiés à tout instant de l'exécution sont les suivants :

- si une partie (client/service) est capable d'envoyer un message (symbolisé par l'action $!m$), alors l'autre partie doit être capable de recevoir ce message (action $?m$) ;
- si une partie est capable de laisser écouler le temps (symbolisé par l'action χ), alors l'autre partie doit être capable de le faire également ;
- enfin, si une partie se termine (action $\surd$), l'autre partie doit se terminer aussi.

La complexité de la relation, par rapport à la définition informelle ci-dessus, est au niveau de la réception du message. Supposons qu'une partie (le client ou le service) puisse recevoir le message m ($?m$), l'autre partie doit-elle être capable d'envoyer *ce* message ? Ce n'est pas nécessaire, puisque l'autre partie peut évoluer de façon non observable d'un état à l'autre, avec certains états capables d'envoyer ce message m, et d'autres non. Par contre, la relation d'interaction oblige qu'à un moment, après un état n'étant pas capable d'envoyer ce message m, il existe un nouvel état qui pourra à nouveau l'envoyer, permettant ainsi d'éviter un blocage d'une ou des deux parties par une attente infinie.

Plus exactement, à tout instant de l'interaction, les éléments suivants doivent être vérifiés :
- l'ensemble des messages que le client est susceptible d'envoyer est l'ensemble des messages attendus par le service ;
- tout message attendu par le client correspond à un message susceptible, au vu des messages échangés, d'être envoyé par le serveur ;
- si le temps s'écoule sur le service, il doit s'écouler sur le client, et inversement si le temps s'écoule sur le client, il doit s'écouler sur le service.

Définition formelle

Soient les notations définissant le complémentaire des actions :
- $\overline{?m} = !m$: le complément de la réception d'un message est l'envoi de ce message ;
- $\overline{!m} = ?m$: le complément de l'envoi d'un message est la réception de ce message ;
- $\forall\, a \notin \{!m\}_{m\in M} \cup \{?m\}_{m\in M}, \overline{a} = a$: le complément des autres actions est l'action elle-même (un écoulement de temps reste un écoulement de temps par exemple).

Définition 4.4.1 (Relation d'interaction client-service) Soit $T_1 = (S_1, L_1, \rightarrow_1, s_{01})$ et soit $T_2 = (S_2, L_2, \rightarrow_2, s_{02})$ deux TIOTS. T_1 et T_2 interagissent correctement si et seulement si $\exists \sim \subseteq S_1 \times S_2$ tel que :

- $s_{01} \sim s_{02}$
- $\forall\, s_1, s_2$ tels que $s_1 \sim s_2$
  - soit $a \notin \{?m \mid m \in M\}$;
    - si $\exists\, s_1 \stackrel{a}{\Longrightarrow}_1 s_1'$, alors $\exists\, s_2 \stackrel{\overline{a}}{\Longrightarrow}_2 s_2'$ avec $s_1' \sim s_2'$ et
    - si $\exists\, s_2 \stackrel{a}{\Longrightarrow}_2 s_2'$, alors $\exists\, s_1 \stackrel{\overline{a}}{\Longrightarrow}_1 s_1'$ avec $s_1' \sim s_2'$
  - soit $m \in M$; si $s_1 \stackrel{?m}{\Longrightarrow}_1 s_1'$ alors
    - $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2$, $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2^+$, $\exists\, s_2^+ \stackrel{!m}{\Longrightarrow}_2 s_2'$ avec $s_1 \sim s_2^+$ et $s_1' \sim s_2'$ tel que w est un mot
    - $\exists\, s_2 \stackrel{!m'}{\Longrightarrow}_2 s_2'$
  - soit $m \in M$; si $s_2 \stackrel{?m}{\Longrightarrow}_2 s_2'$ alors
    - $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1$, $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1^+$, $\exists\, s_1^+ \stackrel{!m}{\Longrightarrow}_1 s_1'$ avec $s_1^+ \sim s_2$ et $s_1' \sim s_2'$ tel que w est un mot
    - $\exists\, s_1 \stackrel{!m'}{\Longrightarrow}_1 s_1'$

Si deux TIOTS T_1 et T_2 existent et vérifient cette relation d'interaction, alors, l'un est le client et l'autre le serveur. Si ce n'est pas le cas, le service est déclaré *ambigu* et une explication est donnée (ensemble des messages ne correspondant pas, etc...).

4.4.2 Ambiguïté

Définition d'un service ambigu

Un service web est dit ambigu lorsqu'il est impossible de générer un client permettant d'interagir de façon adaptée avec ce service. Il faut donc que les TIOTS du couple service-client vérifient la relation d'interaction (voir définition 4.4.1).

Un exemple typique d'ambiguïté est le suivant : le serveur, à un instant donné (donc sur le même état) attend à la fois un message (donc en réception) et peut également en envoyer un. Comment le client peut-il savoir quelle est l'action que le serveur va décider d'exécuter, c'est à dire choisir :

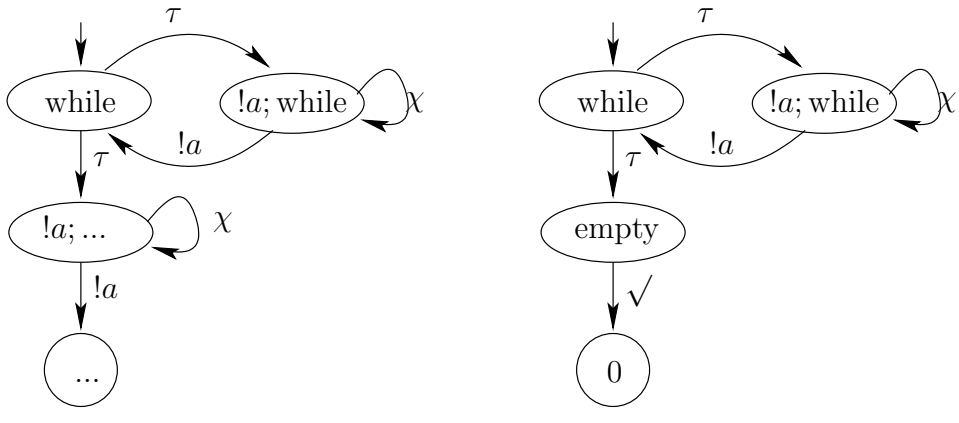
– soit de recevoir le message que le serveur va envoyer ;
– soit d'envoyer le message que le serveur attend...

Pire, imaginons que les deux parties décident d'envoyer un message : ils prennent donc un chemin différent dans leur exécution (ou TIOTS) !

L'ambiguïté porte également sur la détection ou non de la terminaison du service : ainsi, la possibilité de terminer dans un service doit être bien claire et définie pour que le client, n'étant pas informé par un message de cette terminaison, puisse déterminer sans ambiguïté l'instant de la fin du service. La vérification de la relation d'interaction est donc indispensable pour un couple serveur-client.

Exemples de services ambigus

Deux constructeurs de BPEL (ou XLANG) présentent un ou plusieurs risques d'ambiguïté dans des cas d'utilisations très simples : les processus while et switch.



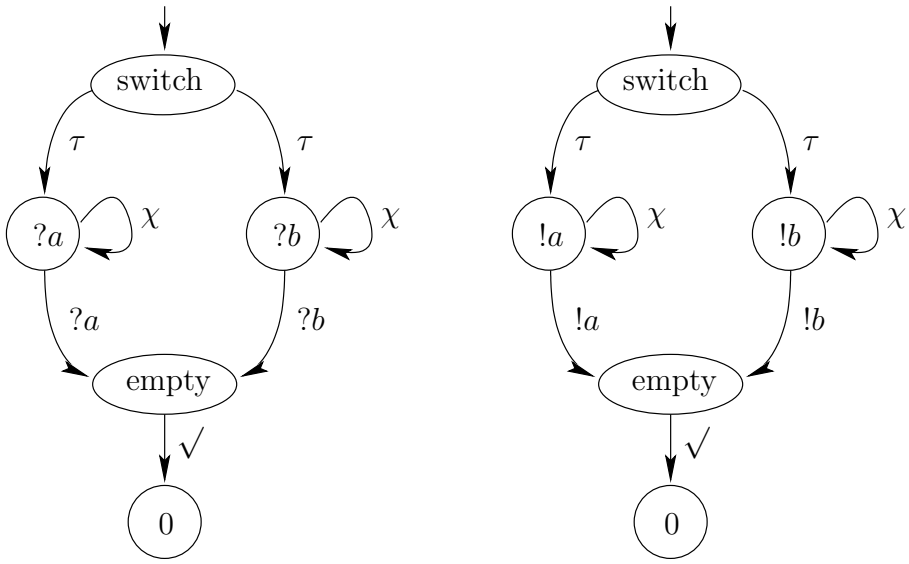
(a) while$\{!a\}; !a; ...$ (b) while$\{!a\}$

FIG. 4.5 – Exemples d'ambiguïté sur le processus while

Exemple d'ambiguïté sur le processus while **(figure 4.5).** Le premier exemple correspond au service web dont le processus métier est défini en partie par : $while\{!a\}; !a; ....$ Ce service est *peut-être*

ambigu : en effet, après la réception d'un message, le client ne sait pas dans quel état se trouve le serveur : continue-t-il toujours d'exécuter sa boucle ou en est-il déjà sorti ? La suite du processus peut effectivement permettre de lever l'ambiguïté du service, d'où l'incertitude sur la possible ambiguïté du processus.

Le deuxième exemple : $while\{!a\}$ est, quant à lui, un service toujours ambigu. En effet, le client ne connaît pas le nombre de messages qu'il va recevoir avant de devoir terminer : la terminaison étant une action silencieuse, elle ne doit pas être en concurrence avec d'autres actions ne permettant pas de la détecter. Ici, pour lever l'ambiguïté, le serveur devrait envoyer un message indiquant qu'il est sorti de la boucle.



(a) switch$\{?a, ?b\}$ (b) switch$\{!a, !b\}$

FIG. 4.6 – Exemples d'ambiguïté et de non ambiguïté sur le processus switch

Exemple d'ambiguïté sur le processus switch **(figure 4.6).** Un autre exemple d'ambiguïté correspond au processus $switch\{?a, ?b\}$: en effet, le client ne sait pas quel message le service attend de recevoir. Le choix de la branche du processus switch à exécuter se réalise au niveau du service et est interne à celui-ci (action τ) : elle ne correspond donc pas à l'échange de messages entre les deux parties. Le client doit alors envoyer un message $!a$ ou $!b$ sans avoir connaissance du réel message attendu par le serveur ! D'où l'ambiguïté.

Par contre, le processus $switch\{!a, !b\}$ n'est pas ambigu : en effet, bien que le choix de la branche à exécuter se fasse au niveau du serveur, le client peut très bien attendre les deux messages. Donc quelque soit le choix du service, le client recevra un message qui le fera poursuivre son exécution dans la même direction que le service (le choix de la branche ne se fait pas au niveau du client).

4.4.3 Algorithme de synthèse du client

Principe de l'algorithme

Le but de cet algorithme est d'obtenir, grâce au TIOTS d'un service non ambigu, un TIOTS du côté client, qui interagit correctement avec le serveur (figure 4.7).

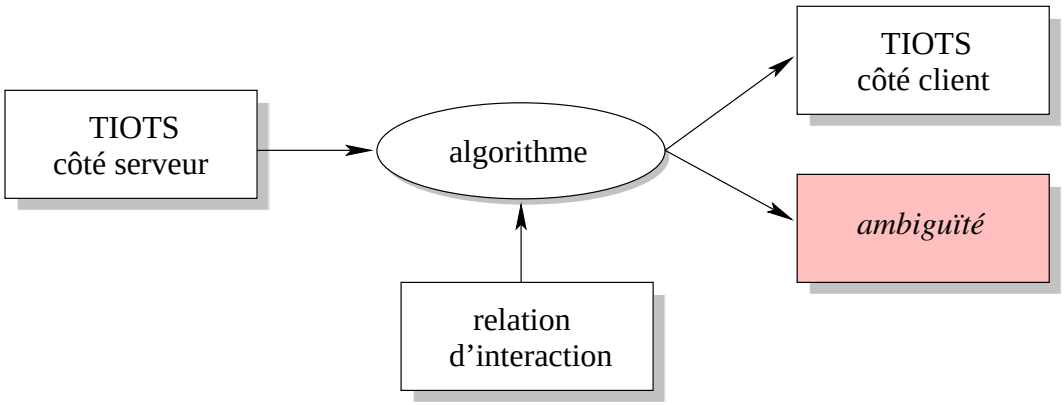


FIG. 4.7 – Principe de l'algorithme de synthèse du client.

Remarque : en cas d'ambiguïté, le TIOTS client n'est pas généré puisqu'il n'est pas possible de garantir une interaction « correcte » (exécution de chemins différents, blocage possible, etc.).

Algorithme de synthèse

L'algorithme 4.4.1 (représentant une étape de la boucle principale) construit le TIOTS *déterministe* du client (noté T_1) en suivant un algorithme proche de la déterminisation du TIOTS du service (noté T_2).

Remarque : pour que client soit implémentable (l'exécution du client suit les actions décrites par le TIOTS T_1), il est indispensable que T_1 soit déterministe (voir section 7.3).

À l'initialisation de l'algorithme, chaque état potentiel s_1 du client est associé au sous-ensemble $S_2(s_1)$ des états du service. Durant la construction du client, il existe alors une pile contenant des couples $(s, S_2(s))$ en cours de traitement, avec s un état possible du client et $S_2(s)$ un sous-ensemble des états du service qui sont en relation avec s par la relation d'interaction.

L'algorithme commence avec le couple correspondant aux états initiaux $(s_{0_1}, \{s_{0_2}\})$ de la pile (s_{0_2} correspond à l'état initial du service). Il arrête son exécution lorsque la pile est vide (c'est-à-dire le TIOTS du client est construit) ou quand il détecte une ambiguïté dans le service.

Dans cet algorithme, à plusieurs reprises, il faut vérifier la non-ambiguïté du service. Pour cela, nous calculons l'ensemble des composantes fortement connexes (SCC) par la transition τ d'un ensemble d'états S_2 du service. Notre relation d'interaction implique les propriétés suivantes :

– si la première composante fortement connexe terminale (TSCC, et première dans le sens de l'implémentation) ne possède pas d'actions du type $\xrightarrow{!m}_2$, alors aucun message ne doit être envoyé par l'ensemble d'états S_2 ;

Algorithm 4.4.1 SYNTHESEDUCLIENT

begin *// une étape de la boucle principale*
dépiler le premier couple de la pile $(s_1, S_2(s_1))$
compléter $S_2(s_1)$ avec les états atteignables par une transition $\stackrel{\epsilon}{\Longrightarrow}_2$
  // (ce nouvel ensemble est noté $S_2(s_1)$ ou plus simplement S_2)
if un état s'_1 du TIOTS client est déjà associé à un état $S_2(s_1)$ **then**
  rediriger toutes les transitions entrantes de s_1 sur s'_1
  effacer s_1.
else
  // définir l'ensemble des arcs sortants de s_1 (s_1 est un nouvel état possible)
  calculer les composantes fortement connexes (SCC) de S_2, puis les composantes terminales (TSCC)
  dans la première TSCC, déterminer l'ensemble possible des actions (messages envoyés, reçus, χ et $\surd$)
  for toutes les autres TSCC **do** *// visiter les autres TSCCs*
    tester si l'interaction est correcte *// voir l'explication*
    if conditions sont vérifiées **then**
      calculer l'ensemble des actions $\stackrel{a}{\longrightarrow}_2 s'_2$
      mettre à jour l'ensemble possible des messages envoyés
    else return "service ambigu" **endif**
  endfor
  for les autres SCC **do** *// visiter les autres SCCs*
    tester si l'interaction est correcte *// voir l'explication*
    if conditions sont vérifiées **then** calculer l'ensemble des actions $\stackrel{a}{\longrightarrow}_2 s'_2$
    else return "service ambigu" **endif**
  endfor
  calculer les ensembles $S_2^{(a)}$ de s'_2 tel que $s_2 \stackrel{a}{\longrightarrow}_2 s'_2$
  créer les nouvelles transitions $s_1 \stackrel{\overline{a}}{\longrightarrow}_1 s'_1$ dans le TIOTS T_1 et ajouter dans la pile le couple $(s'_1, S_2^{(a)})$
endif
end

- dans le cas contraire, chaque TSCC doit envoyer au moins un message. L'ensemble des messages envoyés par S_2 est l'ensemble des messages envoyés par les TSCC ;
- l'ensemble des messages reçus par S_2 doit être exactement le même que l'ensemble des messages reçus par la première TSCC. Cet ensemble est également l'ensemble des messages reçus sur chaque TSCC ;
- finalement, soit $a \in \{\chi, \surd\}$; si la première TSCC de S_2 est capable d'exécuter une action tel que $\xrightarrow{a}_2$, alors les autres doivent le faire également, et dans ce cas, S_2 peut le faire.

En cas de non respect de ces règles, la relation d'interaction n'est pas vérifiée : donc le service est déclaré ambigu.

Exemple de TIOTS service et client

La figure 4.8 représente les TIOTS d'un service et de son client. Le service est non ambigu et il est défini par :

$$?Hello; while[!Hello; ?Hello]; !End$$

Le rôle du service est de recevoir le message $Hello$ de la part d'un client, puis ensuite, d'exécuter ou non un certain nombre de fois la boucle while donc le corps est constitué de deux étapes :

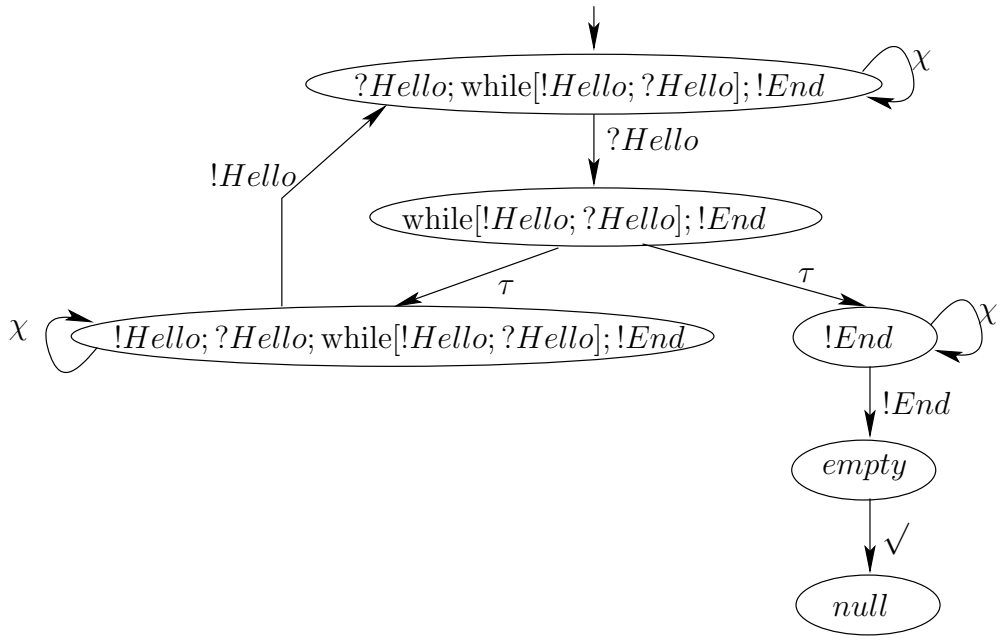
1. la première consiste à envoyer le message $Hello$ au client ;
2. la deuxième consiste à recevoir à nouveau le message $Hello$ du client.

À la fin de cette boucle, le processus se termine par l'envoi du message End de la part du serveur.

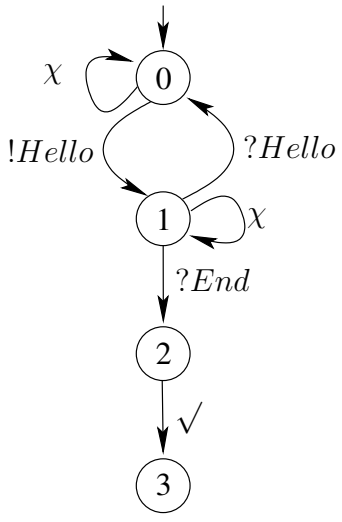
Voici quelques remarques concernant la construction des ces TIOTS :
- le passage du temps est possible sur les états dont les seules transitions sortantes sont des réceptions ou des envois de messages ;
- l'algorithme de génération du client agrège les états du serveur reliés par des τ (en effet, le client ne peut discerner ces états) ;
- enfin, la terminaison n'est pas en concurrence avec une autre action ; lorsque la transition envoyant le message End pour le service est franchie (en recevant ce message pour le client), une seule possibilité reste : celle de terminer l'exécution. Le client et le service terminent bien leur exécution en même temps.

4.5 Synthèse

Sémantique formelle Dans un premier temps, ce chapitre a mis en évidence les différentes méthodes formelles et outils de modélisation utilisés dans nos travaux. Dans un deuxième temps, nous avons montré les actions à modéliser dans l'interaction d'un service web avec son client, éléments indispensables sur lesquels la relation d'interaction se base. Ensuite, dans un troisième temps, nous avons montré les étapes de la formalisation des opérations présentes dans un langage de description comportementale, amenant à la représentation, sous forme d'algèbres de processus, le processus métier du service web. La sémantique des algèbres de processus temporisés étant un système de transition, nous avons montré comment construire ce système de transition pour un service web donné, obtenant ainsi un modèle global de l'exécution du service. Ce modèle est la base de la relation d'interaction permettant de détecter la possible ambiguïté d'un service et la génération du modèle d'un client adapté.



(a) TIOTS du service.



(b) TIOTS du client.

FIG. 4.8 – TIOTS d'un service et le TIOTS client correspondant.

Relation d'interaction et génération d'un client adapté Nous avons donc apportés des réponses pour la formalisation de langages de descriptions comportementales de services web, dans le cas d'une modélisation en temps discret. Cette formalisation aide alors à la génération d'un modèle, ici un TIOTS, représentant le côté serveur du service web. Par la biais de notre relation d'interaction développée pour l'occasion, nous sommes capable de générer un modèle correspondant, représentant un client adapté à ce service web en cas d'absence d'ambiguïté sur ce dernier. La détection d'ambiguïté consistant à vérifier un ensemble de contraintes concernant l'envoi et la réception de messages, la gestion du temps et la gestion de la terminaison dans l'interaction entre le client et le service.

Ouvertures et extensions Cette première approche laisse entrouvrir des extensions. En effet, en se plaçant dans le cadre d'une composition de services, un service est à la fois client et service d'un autre service. Serait-il possible de modéliser leur interaction commune et vérifier, par une relation d'interaction adaptée, leur compatibilité ? Ce sont ces réflexions qui nous ont amené aux travaux que l'on présente dans le chapitre suivant, basé sur l'extension de notre méthode à la vérification de l'interaction interne d'une chorégraphie.

L'utilisation de la discrétisation du temps par *tick* d'horloge peut entraîner une explosion du nombre d'états représentés dans les modèles du service et du client. Pour remédier à ce problème, intrinsèque à cette modélisation, une des solutions est d'utiliser une approche temps dense. Cette approche temps dense utilise alors les horloges pour une représentation continue du temps. C'est cette approche que nous détaillons dans le chapitre 6.

Chapitre 5

Vers la vérification d'une chorégraphie

Ce chapitre étend la méthode de vérification basée sur la détection d'ambiguïté d'un service, en temps discret (présentée chapitre 4), à une vérification basée sur la détection d'ambiguïté de la possible interaction entre plusieurs services web, et plus exactement des services web évoluant dans une *chorégraphie*. Pour cela, après avoir présenté, dans une première partie, une introduction de notre méthode, nous présenterons, dans une deuxième partie, l'environnement et les définitions adoptés, suivie d'une troisième partie précisant les points modifiés ou ajoutés concernant la sémantique opérationnelle, et enfin, la dernière partie définira les algorithmes de vérifications de la compostion. Ces travaux ont été acceptés dans une publication à paraître lors de la conférence ECOWS'2006 [MBSR06].

5.1 Introduction

L'approche adoptée ici est similaire à la précédente basée sur un unique service web dont le principe est de construire le modèle client permettant d'interagir de façon adaptée (sans possibilité de blocage ou d'ambiguïté) avec celui-ci. Nous nous plaçons dans le cadre d'une *chorégraphie* (voir la définition section 2.3.3), où chaque service web (appelé *partenaire* dans la suite de ce chapitre, voir la définition formelle un peu plus loin) évolue dans celle-ci et connaît ce qu'il doit réaliser et quand il sera contacté par les autres partenaires.

Chaque partenaire est un service web à part entière. Il est donc publié et possède sa description syntaxique (WSDL) et comportementale (BPEL par exemple). Ces descriptions peuvent être modélisées par un système de transitions, exactement comme dans l'approche précédente. Ici encore, nous nous plaçons dans le cas d'une discrétisation du temps : le système de transitions utilisé ici sera alors le modèle des TIOTS.

5.1.1 Principe de la vérification

L'algorithme d'obtention des TIOTS de chaque partenaire évoluant dans la chorégraphie reste similaire à l'approche service web unique. De plus, cet algorithme sera toujours nécessaire pour modéliser nos services web. Par contre, l'algorithme de détection d'ambiguïté, appelé relation d'interaction, doit être adapté à la validation de l'interaction des différents partenaires deux à deux. Ainsi, plutôt que d'appliquer la relation d'interaction sur chaque nouvel état du modèle client généré, elle sera appliquée sur deux modélisations complètes de deux services web donnés. En effet, aucun client n'est à générer : les partenaires (ou services web) de la chorégraphie sont à la fois client et service,

suivant l'interaction observée. Ainsi le client d'un service A est le service B, et inversement, le client du service B est le service A.

Une remarque vient immédiatement à l'esprit. Jusqu'à maintenant, l'interaction entre le service et ses différents sous-services n'était pas modélisée (les échanges de messages générés par cette interaction ne font pas partie de l'interaction entre le client et le service). Il est maintenant indispensable de modéliser les interactions liées à l'appel de sous-service depuis un service donné : étape non réalisée dans les algorithmes précédents. En effet, le sous-service est considéré comme un partenaire de la chorégraphie et intervient donc dans l'interaction visible de cette chorégraphie sur laquelle la relation d'interaction doit être vérifiée.

5.1.2 Exemple d'application

La suite de ce chapitre fera souvent référence à un exemple que nous allons détailler ici.

Cet exemple repose sur l'évolution de trois partenaires :
- *une compagnie aérienne*, nommée le partenaire A, vend des billets d'avions aux agences de voyages ;
- *une banque*, nommée le partenaire B, est intermédiaire financier entre la compagnie aérienne et l'agence de voyage ;
- *une agence de voyage*, nommée le partenaire C, achète des billets d'avions aux compagnies aériennes.

Les trois partenaires veulent offrir un service sécurisé et fiable de vente de billets d'avions à de futurs passagers (par l'intermédiaire des agences de voyage) reposant sur des services web.

Une fois le choix de réservation effectué par l'agence de voyage avec l'accord de son client, l'agence (C) contacte l'intermédiaire financier (B) pour payer la compagnie aérienne (A). En fonction du succès ou non de l'opération de paiement entre les partenaires B et A, l'agence de voyage reçoit une confirmation ou une annulation qu'elle transmettra à son client.

Par la suite, nous supposerons que chaque partenaire possède une description abstraite de son comportement observable, c'est-à-dire concernant ses envois et réceptions de messages ainsi que ses contraintes temporelles, mais il possède également les descriptions abstraites des autres partenaires (récupérables car les services web sont publiés). Ces descriptions vont nous permettre de vérifier la composition de chacun de ces trois services (vérification au sens de la relation d'interaction).

5.2 Présentation de l'environnement

Avant d'expliciter les algorithmes de vérification, nous allons définir l'environnement de chorégraphie utilisé. Dans un premier temps, nous aborderons les actions et les processus d'un service web BPEL concernés par la modélisation, ensuite, nous formaliserons le concept de partenaire puis le concept de chorégraphie. Et enfin, nous définirons formellement notre exemple.

5.2.1 Actions et processus d'un service web BPEL

Actions d'un service web BPEL

Les actions observables lors de l'interaction d'un service web avec son client (dans le cas de notre chorégraphie, l'interaction concerne deux partenaires) sont les échanges de messages. Les messages

correspondent aux messages WSDL regroupés en opérations WSDL (voir présentation de WSDL, section 2.3.2).

Soit o une opération, $o \in O$, avec O l'ensemble des opérations possibles que chaque service web met à disposition de la chorégraphie. L'opération o peut être du type :
- *notification*, symbolisée par $o[!m]$, correspondant à l'envoi du message m ;
- *solicitation*, symbolisée par $o[?m]$, correspondant à la réception du message m ;
- *requête-réponse*, symbolisée par $o[!m_1, ?m_2]$, correspondant à l'envoi du message m_1 suivi de la réception du message m_2. Le processus peut également s'écrire $o[!m_1]; o[?m_2]$;
- *solicitation-réponse*, symbolisée par $o[?m_1, !m_2]$, correspondant à la réception du message m_1 suivi de l'envoi du message m_2. Le processus peut également s'écrire $o[?m_1]; o[!m_2]$.

Il est important de préciser qu'un message est envoyé d'un partenaire donné à un partenaire destinataire bien précis. Si le but est d'observer seulement l'interaction entre deux partenaires et non pas l'ensemble de la chorégraphie, il faudra « *filtrer* » les opérations et les messages échangés pour ne modéliser que ceux concernés par l'interaction observée.

Processus d'un service web BPEL

Nous utiliserons les mêmes processus BPEL que ceux définis dans les chapitres précédents, à l'exception du processus invoke que nous ajouterons. Ces processus sont regroupés en plusieurs catégories :
- les processus de base : tels que empty, time, throw ;
- les processus basés sur les messages : tels que $\mathrm{receive}$, reply, invoke ; Nous définirons formellement par la suite ces processus dans notre algèbre de processus temporisés ;
- les processus de contrôles structurés et évolués, contenant généralement d'autres processus de cet ensemble ou des précédents. Les processus appartenant à cet ensemble sont : $\mathrm{sequence}$, switch, while, flow, scope ;

5.2.2 Définition d'un partenaire

Un *partenaire* est un service web complet (c'est-à-dire publié, décrit et répondant aux critères de la définition des services web) et évoluant dans une chorégraphie. Chaque partenaire est représenté par une description BPEL incluant :
- la description de ses partenaires (ou un lien permettant de l'obtenir) ;
- la description de son interface (et de ses opérations locales) ;
- la description du processus abstrait qu'il représente, définissant son comportement. Cette description sera modélisée par la suite par un système de transitions.

Définition 5.2.1 (Partenaire) Une description de processus BPEL abstrait d'un partenaire ω est un tuple $\langle In, P, B \rangle$ tel que :
- $In \subset O$ représente l'interface WSDL du processus, c'est-à-dire l'ensemble des opérations proposées par le service web ($In = \{\omega.o_n \,|\, 0 \leq n \leq n_\omega\}$) ;
- $P \subset O$ est l'ensemble des opérations des partenaires $\omega_i (i \in I)$ de ω, défini par $P = \{\omega_i.o \,|\, \omega_i \neq \omega \text{ et } \exists\, i \in I,\ \omega_i.o \in In_i\}$;
- B représente une expression du processus associée au partenaire ω. Ce processus est généralement décrit dans le langage de description comportementale BPEL.

5.2.3 Définition d'une chorégraphie (utilisant les partenaires)

La chorégraphie est une vue globale de l'interaction entre les différents partenaires. Nous allons définir formellement la chorégraphie en associant chaque opération d'un partenaire à son partenaire cible (réalisée implicitement par la description du service, en utilisant le processus invoke).

Définition 5.2.2 (Correspondance d'opérations) L'opération $\omega_i.o$ liée à un processus invoke (dont le partenaire cible est ω_j) est en correspondance avec l'opération $\omega_j.o$ (noté $\omega_i.o \sim \omega_j.o$) si et seulement si l'un des cas suivant est vérifié :
- si $\omega_i.o = !m$, alors $\omega_j.o = ?m$;
- si $\omega_i.o = ?m$, alors $\omega_j.o = !m$;
- si $\omega_i.o = ?m_1; !m_2$, alors $\omega_j.o = !m_1; ?m_2$;
- si $\omega_i.o = !m_1; ?m_2$, alors $\omega_j.o = ?m_1 : !m_2$.

Définition 5.2.3 (Chorégraphie) Une chorégraphie est un ensemble de partenaires $\{\omega_i = \langle P_i, In_i, B_i \rangle\}_{i \in I}$ tels que toute opération $\omega_i.o$ liée à un processus invoke de cible ω_j soit en correspondance avec une opération $\omega_j.o$ liée à un processus invoke.

La suite de notre travail se restreint volontairement aux chorégraphies vérifiant la définition 5.2.3. Ces chorégraphies pouraient s'appeler *chorégraphies valides*, mais nous gardons le terme *chorégraphie*.

5.2.4 Exemple

Dans cette section, nous définissons notre exemple de chorégraphie en appliquant les définitions précédentes (voir l'explication en section 5.1.2). Chaque processus abstrait associé aux partenaires sera ainsi présenté et détaillé à l'aide des opérateurs BPEL.

Partenaire A

Le partenaire A (la compagnie aérienne) publie les opérations suivantes :
- $o_1[?getFlights, !ListFlights]$: retourne la liste des vols disponibles ;
- $o_2[?getChoice]$: récupère le choix du client ;
- $o_3[?cancel]$: reçoit une information d'annulation de la part du client ;
- $o_4[?payment]$: reçoit la confirmation de la banque pour le paiement.

La définition complète du partenaire A est $A = \langle In_A, P_A, B_A \rangle$ telle que :
- $In_A = \{o_1, o_2, o_3, o_4\}$, correspond à l'interface publiée du service web du partenaire A ;
- $P_A = \{C.o_2\}$, cet ensemble annonce que dans la description du processus abstrait B_A, l'opération o_2 du partenaire C sera utilisée (se traduisant par l'envoi du message de confirmation $!confirm$ lors de son exécution) ;
- B_A[1] = (voir le TIOTS correspondant figure 5.1)
  $o_1[?getFlights, !ListFlights]$;
  $o_2[?getChoice]$;
  $\text{scope}[\text{time}, \{(o_3[?cancel], \text{empty}), (o_4[?payment], C.o_2[!confirm])\}]$

Remarque : l'opération $C.o_2$ consiste en une invocation de l'opération o_2 du partenaire C. Pour le partenaire A, cette invocation correspond à l'envoi du message $confirm$, alors que le partenaire

[1] les retours à la ligne ne sont là que pour la visibilité de l'expression

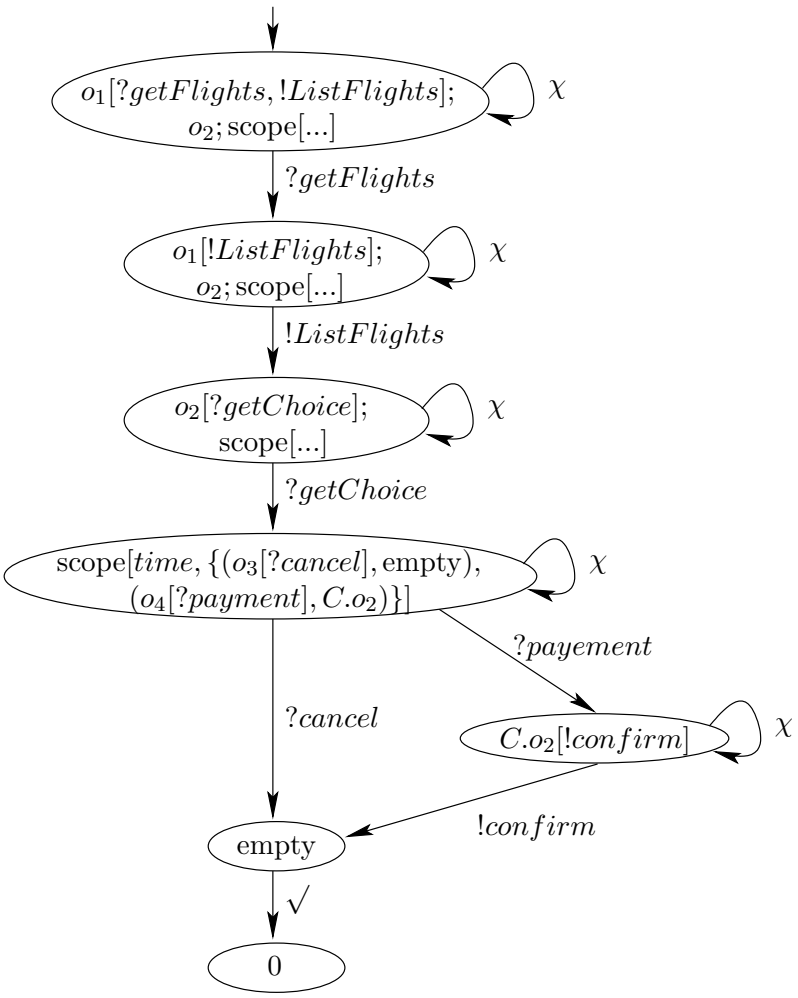


FIG. 5.1 – TIOTS du partenaire A

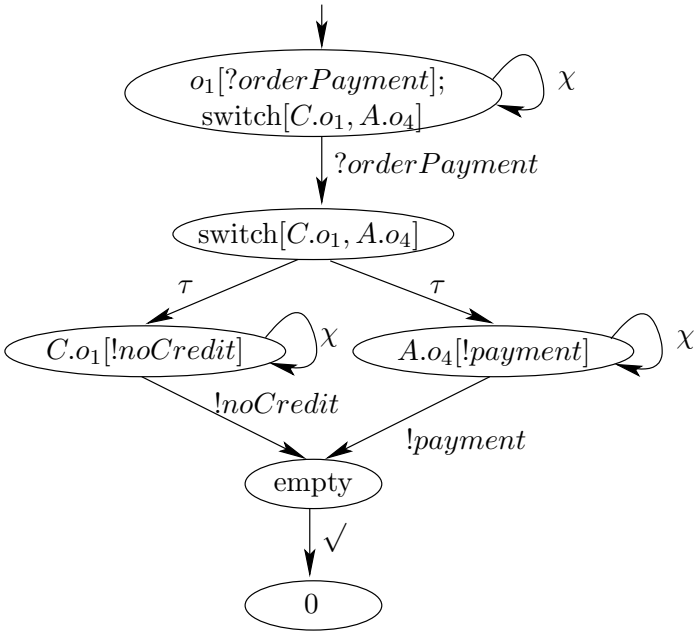


FIG. 5.2 – TIOTS du partenaire B

C, celle-ci correspond à une réception du même message (d'où l'utilisation de $C.o_2[!confirm]$ dans l'expression de B_A).

Partenaire B

Le partenaire B (la banque) publie les opérations suivantes :
– $o_1[?orderPayment]$: réception de l'ordre de paiement depuis l'agence de voyage.

La définition complète du partenaire B est $B = \langle In_B, P_B, B_B \rangle$ telle que :
– $In_B = \{o_1\}$:
– $P_B = \{C.o_1, A.o_4\}$: cet ensemble représente les opérations externes au partenaires B qui seront utilisées lors de la description du processus abstrait B_B. L'opération $C.o_1$ correspond à un paiement échoué et l'opération $A.o_4$ confirme le paiement auprès de la compagnie ;
– $B_B =$ (voir le TIOTS correspondant figure 5.2)
$o_1[?orderPayment]$; switch$[C.o_1[!noCredit], A.o_4[!payment]]$

Partenaire C

Le partenaire C (l'agence de voyage) publie les opérations suivantes :
– $o_1[?noCredit]$: réception de l'annulation du paiement de la part de la banque ;
– $o_2[?confirm]$: réception de la confirmation de paiement de la part de la compagnie aérienne.

La définition complète du partenaire C est $C = \langle In_C, P_C, B_C \rangle$ telle que :
– $In_C = \{o_1, o_2\}$;
– $P_C = \{A.o_1, A.o_2, A.o_3, B.o_1\}$ ensemble des opérations externes au partenaires C qui seront utilisées lors de la description du processus abstrait B_C ;

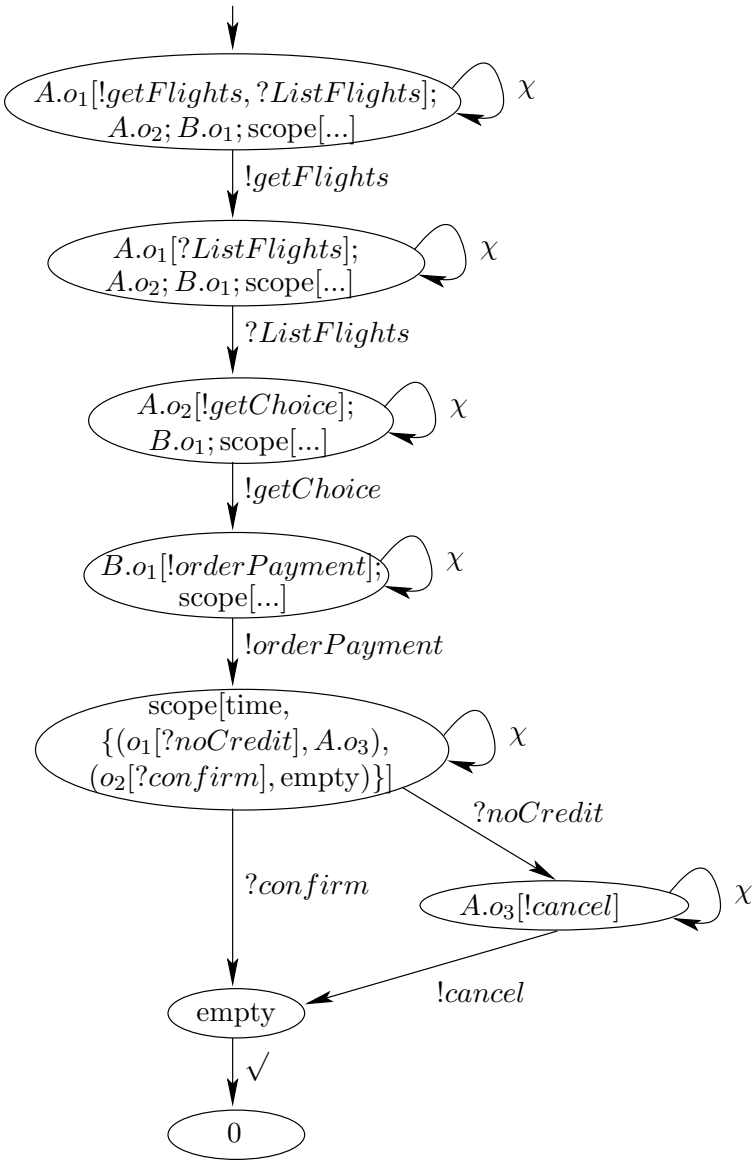


FIG. 5.3 – TIOTS du partenaire C

– $B_C =$ (voir le TIOTS correspondant figure 5.3)
$A.o_1[!getFlights, ?ListFlights]$;
$A.o_2[!getChoice]$;
$B.o_1[!orderPayment]$;
$\text{scope}[\text{time}, \{(o_1[?noCredit], A.o_3[!cancel]), (o_2[?confirm], \text{empty})\}]$

La chorégraphie

La chorégraphie exemple est composée des trois partenaires décrit ci-dessus, c'est-à-dire $\{A, B, C\}$.

5.3 Sémantique opérationnelle et relation d'interaction

Cette section présente les différentes modifications et adaptations nécessaires de notre sémantique formelle et de notre relation d'interaction au vu de la vérification d'une chorégraphie et non plus de la simple interaction client/serveur.

5.3.1 Modélisation du processus BPEL invoke

La sémantique proposée dans le chapitre 4 doit être légèrement modifiée. En effet, celle-ci ne prend pas en compte le processus invoke. Le processus invoke ne définit aucun échange entre le service et le client : il se contente d'appeler une opération d'un autre service, à partir du service l'utilisant, donc cela n'introduit aucun message supplémentaire dans l'interaction service/client. Ainsi, lors de la modélisation de cette interaction, ce processus était simplement ignoré.

Par contre, ici, la modélisation se porte sur l'interaction entre tous les partenaires de la chorégraphie, ce qui veut dire qu'à certains moments, une opération d'un partenaire peut être invoquée par un autre partenaire de la chorégraphie, ajoutant alors des messages supplémentaires à l'interaction observée. Il est alors indispensable de les modéliser, et, par extension, de modéliser le processus invoke.

Les processus receive **et** reply

Le processus receive est un processus BPEL permettant au service web l'exécutant de se mettre en attente de réception d'un message (voir la section 2.3.5 pour plus de détails). Ce processus peut être suivi, dans certains cas, par le processus reply envoyant une réponse à l'émetteur du message reçu par receive. Le processus reply n'a donc pas d'existence seule, il suit obligatoirement un processus receive.

En cas d'utilisation du processus receive seul, le processus obtenu est du type $o[?m]$ (correspondant à une opération « *input* » de WSDL), capable soit de recevoir le message m, soit de laisser passer le temps (en attendant le message m). La sémantique de ce processus est la suivante :

$$o[?m] \xrightarrow{?m} \text{empty}$$

$$o[?m] \xrightarrow{\chi} o[?m]$$

Le processus receive est, par exemple, utilisé dans le partenaire A pour réaliser les opérations o_2, o_3 et o_4.

L'exécution séquentielle des deux processus receive et reply se traduit par l'expression $o[?m_1, !m_2]$ correspondant à la réception du message m_1 suivi de l'envoi du message m_2 (opération « *input-output* » de WSDL). La sémantique est alors la suivante :

$$o[?m_1; !m_2] \xrightarrow{?m_1} o[!m_2]$$

$$o[?m_1; !m_2] \xrightarrow{\chi} o[?m_1; !m_2]$$

$$o[!m_2] \xrightarrow{!m_2} \text{empty}$$

$$o[!m_2] \xrightarrow{\chi} o[!m_2]$$

Un exemple d'utilisation de ces deux processus est fourni par l'opération o_1 du partenaire A.

Le processus invoke$[o]$

Grâce au processus invoke, un partenaire ω_i peut invoquer une opération d'un partenaire ω_j. L'invocation est réalisée en une seule étape si l'opération à invoquer est une opération simple (c'est-à-dire si ω_j utilise le processus receive de manière autonome) et en deux étapes dans le cas contraire (ω_j utilise les processus receive et reply).

Les différents cas possibles sont donc (pour le partenaire ω_i invoquant une opération du partenaire ω_j) :

- $\omega_j.o[!m]$ dans le cas d'une opération simple (sans retour) ;
- $\omega_j.o[!m_1, ?m_2]$ dans le cas d'une opération requête-réponse.

La sémantique de ces opérations se déduit aisément du paragraphe précédent :

$$\omega_j.o[!m_1; ?m_2] \xrightarrow{!m_1} \omega_j.o[?m_2]$$

$$\omega_j.o[!m_1; ?m_2] \xrightarrow{\chi} \omega_j.o[!m_1; ?m_2]$$

$$\omega_j.o[?m_2] \xrightarrow{?m_2} \text{empty}$$

$$\omega_j.o[?m_2] \xrightarrow{\chi} \omega_j.o[?m_2]$$

5.3.2 L'algorithme de compatibilité de deux TIOTS

L'*algorithme de compatibilité* est issu de la relation d'interaction et de la détection d'ambiguïté vus précédemment (section 4.4). Il prend en entrée deux TIOTS et il a pour but de vérifier l'interaction de tous leurs états, contrairement aux algorithmes précédents qui construisaient un TIOTS à partie d'un autre (si possible).

Pour cela, entre-autres, il calcule et utilise les ensembles d'états accessibles par des transitions silencieuses (étiquetées τ) depuis chaque état. L'étape d'initialisation consiste à calculer ces ensembles sur les états initiaux des deux TIOTS, qui doivent également être en relation d'interaction.

Une étape de l'algorithme

Nous allons maintenant décrire une étape de l'algorithme, dont voici les notations utilisées :
- s_{01} et s_{02} sont les états initiaux des TIOTS T_1 et T_2 ;
- la fonction $\tau - clôture(s)$ permet de calculer l'ensemble des états accessibles par une transition silencieuse depuis l'état s ou l'ensemble d'états s, les $\tau - clôture$ des états s_{01} et s_{02} se nomment respectivement $\tau_{s_{01}}$ et $\tau_{s_{02}}$;
- une pile P de couples d'états à traiter, contenant initialement le couple $(\tau_{s_{01}}, \tau_{s_{02}})$;
- un ensemble de couples R correspondant aux couples déjà traités ;
- l'ensemble des messages envoyés se nomme $M_!$, l'ensemble des message reçus se nomme $M_?$ et $M_! \cup M_? = M$;
- la fonction $suivant(s, a)$ calcule l'ensemble des états atteignables par l'action a depuis l'état s.

Une étape de l'algorithme, pour le couple (s_1, s_2) sommet de la pile P, exécute le code suivant :
- si $(S_1, S_2) \in R$ (c'est-à-dire $\exists (S'_1, S'_2) \in R$ avec $S_1 \subseteq S'_1$ et $S_2 \subseteq S'_2$) alors dépiler (s_1, s_2) et passer au nouveau sommet de la pile ;
- pour chaque état $s_1 \in S_1$ faire :
  - soit $a \in M_!$ tel que $s_1 \stackrel{!a}{\Longrightarrow}_1 s'_1$, si $\exists s_2 \in S'_2$ tel que $\nexists s_2 \stackrel{?\bar{a}}{\Longrightarrow}_2 s'_2$:
    - alors retourner la non compatibilité des deux TIOTS.
    - sinon ajouter à la pile P le couple $\left(\bigcup_{s_i \in S_1} \tau_{suivant(s_i,!a)}, \bigcup_{s_j \in S_2} \tau_{suivant(s_j,?\bar{a})} \right)$ et ajouter à la pile R le couple (S_1, S_2).
  - soit $a \in M_?$ tel que $s_1 \stackrel{?a}{\Longrightarrow}_1 s'_1$, si $\exists s_2 \in S'_2$ tel que $\nexists s_2 \stackrel{!a'}{\Longrightarrow}_2 s'_2$ ($a' \in M_!$) :
    - alors retourner la non compatibilité des deux TIOTS.
    - sinon ajouter à la pile P le couple $\left(\bigcup_{s_i \in S_1} \tau_{suivant(s_i,?a)}, \bigcup_{s_j \in S_2} \tau_{suivant(s_j,!a')} \right)$ et ajouter à la pile R le couple (S_1, S_2).
- les même étapes sont répétées pour chaque état $s_2 \in S_2$.

L'algorithme est très similaire à un algorithme de bisimulation, mais ici, à chaque étape, nous vérifions la relation d'interaction à la place de la relation de bisimulation.

Partenaire non compatible et ambiguïté

Deux TIOTS sont non compatibles quand l'un des deux présente une ambiguïté, au sens des définitions vues dans la partie 4.4.2. En effet, la non compatibilité de deux TIOTS réside dans le fait qu'un système modélisé par son TIOTS (que l'on qualifiera de serveur) n'est pas capable de suivre un chemin identique dans son exécution par rapport à l'autre TIOTS (que l'on qualifiera de client) : l'un des deux systèmes prendra donc un choix qui empêchera le bon déroulement de l'interaction, ou, en d'autres termes, un des choix d'exécution d'un des deux systèmes est ambigu au niveau des décisions que l'autre système doit prendre.

5.4 Vérification d'une chorégraphie

Comme nous l'avons décrit, une chorégraphie est composée de partenaires. Chaque partenaire possède une description de son comportement observable (au niveau des communications et des

contraintes de temps). La chorégraphie est définie en liant les différentes interactions de chacun des partenaires. La vérification de la chorégraphie que nous allons présenter se base sur l'ambiguïté ou non de ces interactions.

5.4.1 Partenaires, interactions et vérification d'une chorégraphie

Tout d'abord, il est nécessaire de vérifier l'interaction entre un partenaire P et l'ensemble des autres partenaires de la chorégraphie : l'ensemble agrégé de ces autres partenaires formant le client du partenaire P. Si cette vérification est effectuée pour chacun des partenaires P, alors il est possible d'en déduire que l'ensemble de la chorégraphie est vérifiée au niveau de l'interaction et du comportement observable.

Pour une chorégraphie donnée, si un des partenaires présente un comportement ambigu, alors la chorégraphie n'est pas exempte de point de blocage ou d'ambiguïté ! En effet, si un partenaire dévie son exécution par rapport à son client qui n'est autre qu'un partenaire de la chorégraphie, alors l'interaction entre ces deux partenaires peut se bloquer ou aboutir sur une incohérence, entraînant l'interaction de la chorégraphie en erreur également.

D'où la *condition nécessaire* suivante : si l'un des partenaires présente une ambiguïté, la chorégraphie est déclarée ambiguë.

5.4.2 Composition ou agrégation de deux partenaires

Chaque partenaire interagit avec un sous-ensemble de partenaires évoluant dans la chorégraphie. Ce sous-ensemble de partenaires peut être considéré comme un unique partenaire simulant son comportement. Pour cela, notre algèbre de processus temporisés a été étendue par l'opérateur $\|_A$ pour permettre l'« agrégation » de deux partenaires.

Plus exactement, ces nouvelles règles décrivent les différentes exécutions possibles du partenaire virtuel, en fonction des actions exécutées par chacun des deux partenaires qu'il agrège. Elles ne décrivent pas comment les partenaires agrégés vont évoluer, mais quel est le comportement visible de l'extérieur de l'ensemble de ces partenaires : par exemple quel sera le comportement visible pour un client interagissant avec une telle agrégation (ou une chorégraphie composée ici de deux partenaires, mais extensible récursivement à bien plus).

Définition 5.4.1 (Agrégation de deux partenaires) Soient $\omega_i = \langle In_i, P_i, P\rangle$ et $\omega_j = \langle In_j, P_j, Q\rangle$ deux partenaires. L'agrégation de ω_i et ω_j, notée $\omega_i \ \|_A \ \omega_j$, est le partenaire $\omega_k = \langle In_k, P_k, B_k\rangle$, avec :

- $In_k = (In_i \cup In_j)$;
- $P_k = \{\omega_l.o \in (P_j \cup P_i) | l \neq i, j\}$;
- $B_k = P \ \|_A \ Q$ (voir ci-dessous).

Cette définition peut être étendue à une chorégraphie dont le nombre de partenaires est supérieur à deux. Ainsi, le comportement d'une chorégraphie est définie en agrégeant récursivement l'ensemble des partenaires deux à deux.

La suite présente les règles de notre sémantique définissant l'opérateur $\|_A$, traduisant le comportement observable de l'agrégation de deux partenaires ω_i et ω_j, noté $P \ \|_A \ Q$. Les règles de cet opérateur sont regroupées en quatre catégories :

– la gestion des échanges de messages entre partenaires ;
– la gestion de la terminaison ;
– la gestion de l'écoulement du temps ;
– les cas non couverts par les catégories précédentes.

Échanges de messages entre partenaires

Plusieurs cas possibles d'exécution peuvent être rencontrés durant l'évolution des partenaires agrégés. Le premier cas, ici traité, correspond à l'échange de messages entre les partenaires de l'agrégation eux-mêmes. Ces échanges sont silencieux d'un point de vue extérieur : c'est-à-dire qu'ils n'apparaissent pas dans l'interaction entre la chorégraphie formée de ces deux partenaires et de son client. Ces échanges sont alors modélisées par l'action silencieuse τ.

$$\forall\, a \in *m \quad \frac{P \xrightarrow{a} P', Q \xrightarrow{\overline{a}} Q'}{P \parallel_A Q \xrightarrow{\tau} P' \parallel_A Q'}$$

La terminaison

Pour la terminaison des partenaires, deux cas se présentent. Dans le premier cas, les deux partenaires se terminent au même moment, alors la chorégraphie se termine, comme l'indique la règle suivante :

$$\frac{P \xrightarrow{\surd} 0, Q \xrightarrow{\surd} 0}{P \parallel_A Q \xrightarrow{\surd} 0}$$

Le deuxième cas se présente lorsque l'un des partenaires (nommé P) de la chorégraphie se termine avant l'autre. Alors, la chorégraphie ne termine pas puisque l'un des partenaire peut encore évoluer. La terminaison du partenaire P se réalise de manière silencieuse en exécutant la première action possible du processus Q :

$$\frac{P \xrightarrow{\surd} P', Q \xrightarrow{a} Q'}{P \parallel_A Q \xrightarrow{a} Q} \text{ avec } a \neq \surd$$

L'écoulement du temps

L'écoulement du temps présente deux cas. Le premier cas synchronise l'écoulement du temps discret sur les deux partenaires, cela entraîne que la chorégraphie permet également l'écoulement du temps.

$$\frac{P \xrightarrow{\chi} P', Q \xrightarrow{\chi} Q'}{P \parallel_A Q \xrightarrow{\chi} P' \parallel_A Q'}$$

Le deuxième cas se rencontre lorsque l'un des deux partenaires (ici P) peut exécuter l'action écoulement du temps et l'autre non. Alors l'écoulement du temps sur P est exécuté et la chorégraphie permet également un écoulement du temps.

$$\frac{P \xrightarrow{\chi} P',\ \neg\, Q \xrightarrow{\chi}}{P \parallel_A Q \xrightarrow{\chi} P' \parallel_A Q}$$

Déroulement normal

Enfin, si l'exécution des processus P et Q ne rentre pas dans les cas précédents, la chorégraphie correspondante autorise l'exécution « séquentielle » des deux processus.

$$\forall\, a \notin \{\chi, \surd\} \quad \frac{P_j \xrightarrow{a} P'}{P \parallel_A Q \xrightarrow{a} P' \parallel_A Q}$$

$$\forall\, a \notin \{\chi, \surd\} \quad \frac{Q \xrightarrow{a} Q'}{P \parallel_A Q \xrightarrow{a} P \parallel_A Q'}$$

Exemple d'agrégation de partenaires

Reprenons notre exemple de la section 5.1.2 et dont la formalisation a été réalisée lors de la section 5.2.4. Plus exactement, intéressons nous au partenaire A (nommé ω_A) et de la vue agrégée des partenaires B et C depuis ce partenaire, nommée ω_{BC}.

Tout d'abord, il est évident, du point de vue de l'interaction entre ω_A et ω_{BC}, que toutes les invocations des opérations du partenaire C depuis le partenaire B ne seront pas présentes dans celle-ci, tout comme les invocations des opérations du partenaires B depuis le partenaire C. Il faut donc « supprimer » les opérations suivantes :

- pour ω_B, l'invocation de l'opération $C.o_1$ n'entraînera aucun échange dans l'interaction entre ω_A et ω_{BC} ;
- pour ω_C, l'invocation de l'opération $B.o_1$ est à « supprimer » pour les mêmes raisons.

Par contre, les invocations des opérations $A.o_4$ du partenaire B et $A.o_1$, $A.o_2$ ainsi que $A.o3$ du partenaire C sont à conserver puisqu'elles sont directement concernées par l'interaction observée ici.

Par extension, la suppression de l'invocation de l'opération $C.o_1$ du partenaire B entraîne que l'opération o_1 de partenaire C ne sera pas non plus visible dans l'interaction qui nous intéresse. De même pour l'opération o_1 du partenaire B.

En appliquant toutes ces observations, la modélisation obtenue du comportement observable de l'agrégation des partenaires B et C, du point de vue de A, est représentée par le TIOTS de la figure 5.4.

5.4.3 Vérification et détection d'ambiguïté d'une chorégraphie

Pour chaque partenaire d'une chorégraphie, il est possible d'agréger l'ensemble des partenaires restant par la méthode présentée précédemment. Une fois l'agrégation effectuée, chaque partenaire et le sous-ensemble correspondant sont modélisés sous forme de TIOTS. Nous pouvons leur appliquer l'algorithme de compatibilité présenté en section 5.3.2, permettant ainsi de vérifier la relation d'interaction sur chacun des partenaires de la chorégraphie.

Enfin, une chorégraphie est dite sans ambiguïté si chaque partenaire évoluant dans cette chorégraphie interagit correctement (échanges de messages, absence de blocage, et détection de la terminaison) avec ses partenaires.

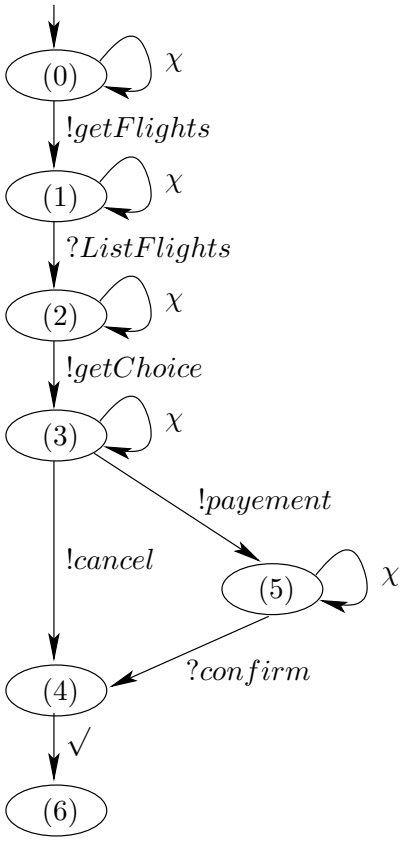


FIG. 5.4 – TIOTS, observé par le partenaire A, de l'agrégation des partenaires B et C

Application de la vérification d'une chorégraphie sur l'exemple

Dans notre exemple, une partie de la vérification de la chorégraphie consiste à appliquer l'algorithme de compatibilité aux TIOTS obtenus pour les partenaires A (figure 5.1) et l'agrégation des partenaires B et C (figure 5.4). Puis, les étapes suivantes de la vérification consistent à réaliser la même vérification pour le partenaire B et la vue agrégée des partenaires A et C et enfin pour le partenaire C et la vue agrégée des partenaires A et B.

Extensions, avantages et implémentation de la méthode

Il est possible d'implémenter notre méthode de sorte qu'une seule vérification se fasse depuis chaque partenaire de la chorégraphie. Chaque partenaire, ayant connaissance des descriptions comportementales des autres partenaires, peut alors exécuter localement les algorithmes décrits précédemment, afin de vérifier sa propre interaction avec la chorégraphie. Cela nous amène à réaliser de la répartition de calcul, évitant l'existence d'une entité supplémentaire aux partenaires de la chorégraphie devant réaliser ce travail de vérification. Une extension du client générique pourrait être envisagée en ce sens (le client générique sera présenté au cours du chapitre 7.3).

De même, imaginons un contexte de services web évoluant dans une composition automatique (ou semi-automatique). Un service web peut alors vérifier sa propre interaction vis-à-vis de l'ensemble des services web (se ramenant alors aux partenaires d'une chorégraphie) et détecter si la substitution d'un service à un autre est appropriée ou non.

5.5 Synthèse

Ce chapitre montre notre apport concernant la vérification de l'interaction interne d'une chorégraphie de services web, dans le cas d'une modélisation en temps discret. Ainsi, la première étape consiste à étendre notre algèbre de processus à un ensemble de partenaires et plus particulièrement à ajouter un opérateur d'agrégation permettant de ne représenter que les actions visibles, pour un partenaire donné, de l'ensemble des autres partenaires de la chorégraphie. Ceci nous amène ensuite à permettre la vérification, pour chaque partenaire, de sa propre modélisation (par un TIOTS) avec la modélisation de la vue agrégée des autres partenaires. Cette vérification utilise une version adaptée à deux modèles, déjà générés, de la relation d'interaction vu au chapitre précédent. Si l'ensemble de ces vérifications aboutissent à une absence d'ambiguïté, l'interaction interne de la chorégraphie est déclarée non ambiguë.

Chapitre 6

Sémantique en temps dense

Ce chapitre reprend la sémantique présentée lors du chapitre 4 et l'adapte pour proposer une solution en *temps continu* ou *temps dense*. Pour cela, dans un premier temps, nous allons présenter les motivations d'un tel travail, dans un deuxième temps, la nouvelle sémantique sera présentée, en précisant les points qui ont nécessité des améliorations, ainsi que la génération de l'automate temporisé de la partie service ; puis, dans un troisième temps, nous présenterons les algorithmes de synthèse de client, adaptés au temps dense. Ces travaux ont été présentés dans deux publications [HMMR04a] (en se focalisant principalement sur la sémantique) et [HMR06] (en se focalisant principalement sur la génération du modèle de la partie client).

6.1 Introduction

Nous avons adopté le mode temps dense principalement pour deux raisons : d'une part c'est un modèle qui évite l'explosion du nombre d'états liée à la discrétisation du temps ; d'autre part, il est plus proche de la réalité et il peut modéliser plus finement les systèmes asynchrones.

6.1.1 Limitations du temps discret

La discrétisation du temps consiste à modéliser, dans le cas des systèmes de transitions, l'action *écoulement du temps* comme une véritable action à elle toute seule, étiquetant une transition, et d'une durée fixée. Ainsi, dès qu'un *tic* d'horloge (symbolisé dans notre modèle par χ) se présente, une transition est ajoutée au système de transitions.

Bien évidement, si un état présente une attente infinie, il suffit de lui ajouter une transition bouclant sur le même état. Mais pour modéliser un temps maximal d'exécution (très présent dans les opérateurs des services web), il faut alors associer une transition entre deux états différents à chaque *tic* d'horloge.

L'action χ représente donc une durée fixe (par exemple, 1 seconde, 2 secondes, 1 minute, 1 heure, etc.). Si les ordres de grandeur des différentes actions temporisées du système à modéliser sont les mêmes, il suffit alors d'ajuster la durée de l'action χ pour éviter d'avoir 45 transitions de suite pour représenter 45 secondes par exemple. Par contre, regardons maintenant le cas où les ordres de grandeur sont différents : prenons un temps maximal d'exécution de 1 seconde encapsulé dans un temps d'exécution global de 1 heure. Il faut alors fixer la durée de χ à 1 seconde (car les deux durées doivent pouvoir être modélisées avec la même action χ). Et là, le fait d'attendre 1 heure est modélisé par 3600 transitions...

3600 états, ce n'est pas énorme, mais il faut bien imaginer que notre système présente plusieurs états qui évolueront indépendamment de ce contrôle du temps, et qu'il faudra donc, pour chaque état, envisager toutes les possibilités d'évolution du temps. La modélisation va générer un grand nombre d'états et de transitions pour modéliser tous les comportements du système ! Il est alors facile d'imaginer que le nombre d'états va très vite exploser en temps discret, même sur des systèmes relativement simples, rendant ainsi beaucoup plus longs, voire impossibles l'analyse et le traitement de tels modèles.

Le temps dense, par contre, représente, comme nous l'avons vu dans la section 3.1.3, un écoulement de temps grâce à une horloge. Un temps maximum pour franchir une transition est alors représenté par une condition sur cette transition comme par exemple $h < 3$ indiquant que la transition doit être franchie tant que le système n'a pas laissé écouler plus de 3 unités de temps (depuis la dernière initialisation de h). Nous avons donc alors affaire à des modèle nettement plus petits en nombre d'états.

6.1.2 Modélisation des modèles asynchrones

Les services web sont, par nature, des systèmes asynchrones. En effet, ils font parties des systèmes répartis (le client n'est pas sur la même machine que le service, et le service peut lui-même être réparti sur plusieurs serveurs ou être composé de plusieurs services) et les différentes entités (client et service) communiquent par des envois de messages à travers un réseau de façon asynchrone.

Dans le chapitre 4, le *modèle temps discret* utilisé présentait une approximation : les services web étaient considérés comme des systèmes dont le temps évolue de façon synchrone entre toutes les entités. Le fait d'adopter un *modèle temps dense* plutôt qu'un modèle temps discret permet de représenter le temps s'écoulant de manière continue (donc à chaque instant) sur l'ensemble du système. Ce qui est la réalité au niveau d'un système asynchrone tel que les services web.

Cependant, une dernière approximation est toujours faite dans ce modèle : l'échange de messages est supposé instantané, nous rapprochant encore des systèmes synchrones.

6.1.3 Adaptation de notre sémantique *temps discret* au modèle *temps dense*

Cette évolution vers le *temps dense* est une adaptation. Nous utilisons toujours les algèbres de processus temporisés, et le mécanisme de systèmes de transitions est conservé, mais nous adoptons le modèle des automates temporisés à la place du modèle des TIOTS, très proche. Au niveau de l'implémentation des algorithmes de génération, tout n'a pas été réécrit. La majeure partie a été conservée et est commune aux deux méthodes, seuls des mécanismes de gestion des gardes ont été ajoutées (voir chapitre 7 pour plus de détails concernant l'implémentation).

Utilisation des algèbres de processus temporisés

La sémantique temps dense utilise les expressions d'algèbres de processus temporisés, avec une légère adaptation : le passage explicite du temps n'est plus exprimé directement dans les règles. Il n'y a donc pas de mention d'écoulement de temps sur ces règles : en effet, le temps s'écoule en continu, à tout moment ! Par contre, une information concernant la possibilité de déclenchement d'un mécanisme de *timeout* est ajoutée.

L'utilisation de la méthode générique, aussi bien au niveau de la modélisation qu'au niveau du code du logiciel, a été très utile lors de cette adaptation. La plupart des règles de la sémantique sont communes au temps discret et ne nécessitent que quelques adaptations. Quant à la partie du code de

génération du modèle, à partir des règles et de la description du service web, elle est commune aux deux méthodes : seuls des ajouts explicites d'informations de garde (en cas de *timeout*) et de remontée d'informations pour ces gardes ont été ajoutées.

Utilisation des automates temporisés

Le modèle de systèmes de transitions associé à notre sémantique temps discret était le TIOTS, représentant l'écoulement du temps comme une action à part entière. Cette distinction n'est plus vérifiée dans le cadre du temps dense : le temps s'écoule à chaque instant du modèle (donc sur chaque état). Par contre, certaines émissions/réceptions de messages ou certaines actions sont gardés par des limites de temps : au delà d'un certain temps, la transition n'est alors plus franchissable. Pour modéliser cela, nous utilisons le modèle des *automates temporisés* et ses horloges associées.

6.2 Les actions d'un service web

Les actions d'un service web, dans le cadre de la modélisation temps dense, sont similaires, ici, à celle présentées dans le cadre de la modélisation temps discret (voir section 4.2). Nous ne présenterons ici que les différences.

6.2.1 Les différentes actions

Les étiquettes de transitions des automates temporisés modélisent les différentes actions observables qu'un service web peut réaliser durant son interaction avec un client.

- l'échange d'un message m est symbolisé, comme dans le cas temps discret, par le symbole $?m$ pour la réception d'un message et le symbole $!m$ pour l'envoi d'un message. $*m$ doit être interprété comme un envoi ou une réception du message m ;
- le *comportement inobservable* est modélisé par l'action τ, correspondant à une évaluation interne au service d'une condition (sans échange de messages) ;
- dans le cadre temps dense, une deuxième action silencieuse est ajoutée : l'action correspondant à un *timeout*. Cette action est symbolisée par to (voir également la section 6.2.2) ;
- un service web peut générer une *exception*, qui peut être interceptée ou non. En cas de non interception (voir les processus scope et pick), le symbole e apparaît ;
- la *terminaison* de l'exécution d'un service web est symbolisée par l'action $\surd$. Cette action, comme dans le cadre discret, est indispensable pour éviter des problèmes d'ambiguïté comme par exemple : le service attend une confirmation de plus alors que le client a déjà terminé.

6.2.2 Les gardes d'horloges

Les automates temporisés générés par nos algorithmes présentent quatre types de « conditions » sur les horloges (voir figure 6.1) :

- une horloge doit être initialisée avant toute utilisation. Cette initialisation peut être rencontrée sur toutes les transitions (sur la transition indiquant l'état de départ dans l'exemple) ;
- un état peut présenter un invariant. C'est le cas de l'état (0). Cet invariant indique le temps, au maximum, que le système peut rester sur l'état. Au delà de ce temps, l'exécution devra avoir évoluée vers un des autres états ;
- les gardes (par exemple $H < 2$) permettent d'indiquer quand la transition est franchissable, en fonction du temps écoulé et mesuré par l'horloge correspondante. En cas de plusieurs gardes

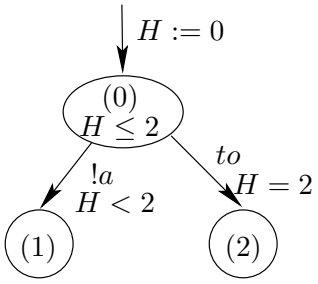


FIG. 6.1 – Exemples de gardes d'horloges.

sur des horloges différentes, la conjonction de l'ensemble des gardes doit être vérifiée pour que la transition soit franchissable ;
– les transitions symbolisant le *timeout* sont étiquetées par le symbole to. Cette action est accompagnée d'une garde du type $H = 2$ indiquant que la transition n'est franchissable qu'au moment ou H attend la valeur 2 (l'invariant de l'état assure que cette transition sera franchissable).

6.3 Sémantique des opérations

Cette partie reprend les différentes règles exposées dans la section 4.3 pour les adapter à la sémantique temps dense. Tous les processus du temps discret se retrouvent ici, sauf le processus time qui n'a plus lieu d'être car le temps n'est plus discrétisé.

6.3.1 Les règles pour les éléments de base

Le processus empty

L'élément de base de la sémantique est le processus empty. Comme en temps discret, le processus ne peut que se terminer en exécutant l'action $\surd$, et ainsi devenir le processus 0 (ou *null*). Par contre, contrairement au temps discret, ici, il laisse passer le temps : mais cela n'est pas symbolisé, en effet, tous les processus laissent passer le temps dans notre sémantique temps dense.

$$\mathrm{empty} \xrightarrow{\surd} 0$$

Le processus throw

Le processus $\mathrm{throw}[e]$ (avec $e \in E_X$, l'ensemble des exceptions que le processus peut lever) permet de déclencher une exception. Une exception peut être interceptée par un processus la gérant (voir les processus scope et pick) ; si ce n'est pas le cas, le processus principal dérive sur le processus *null*.

$$\forall\, e \in E_X,\ \mathrm{throw}[e] \xrightarrow{e} 0$$

6.3.2 Les règles pour les éléments basés sur les messages

Les processus receive **et** reply

Comme dans le cas temps discret, nous avons deux processus receive et reply correspondant respectivement à la réception et l'envoi d'un message. La suite d'opération *envoi suivi d'une réception* correspond à une exécution séquentielle des deux sous-processus.

Par contre, ici encore, aucune indication sur l'écoulement du temps : cet écoulement se réalise de façon implicite.

$$*o[m] \xrightarrow{*m} \text{empty} \quad \text{avec} * \in \{?, !\}$$

Le processus invoke

Le processus invoke consiste à invoquer une opération d'un sous-processus. Les échanges de messages ne sont donc pas visibles entre l'interaction client/service que nous modélisons. Comme dans le cas temps discret, nous ne modélisons donc pas ce processus.

6.3.3 Les règles pour les éléments de contrôle structuré

La sémantique des éléments de contrôle structuré est totalement similaire au temps discret : aucune condition n'est explicitée quant à l'écoulement du temps pour ces processus, seules des conditions sur le choix des branches est réalisées (et de façon interne au service).

Le processus sequence **(;)**

L'exécution du processus séquentiel $P \ ; \ Q$ se réalise de la même façon qu'en temps discret. Le passage du processus P au processus Q se fait sans aucun échange de message et sans aucune transition le symbolisant.

$$\forall\, a \neq \surd,\ \frac{P \xrightarrow{a} P'}{P \ ; \ Q \xrightarrow{a} P' \ ; \ Q} \quad \text{et} \quad \forall\, a,\ \frac{P \xrightarrow{\surd} \ \wedge \ Q \xrightarrow{a} Q'}{P \ ; \ Q \xrightarrow{a} Q'}$$

Le processus switch

De la même façon que dans le cadre du temps discret, le processus switch$[\{P_i\}_{i \in I}]$ exécute un des processus P_i, après avoir évalué une condition interne.

$$\forall\, i \in I,\ \text{switch}[\{P_i \,|\, i \in I\}] \xrightarrow{\tau} P_i$$

Le processus while

Même remarque pour le processus while$[P]$ qui exécute un certain nombre de fois le processus P avant de terminer son exécution en devenant le processus empty.

$$\text{while}[P] \xrightarrow{\tau} P \ ; \ \text{while}[P]$$

$$\text{while}[P] \xrightarrow{\tau} \text{empty}$$

6.3.4 Les règles pour les éléments de contrôle évolué

Le processus flow

Le processus $\text{flow}[\{P_i\}_{i\in I}\}]$ exécute simultanément l'ensemble des processus $\{P_i\}$.

Remarque : comme précisé dans la version temps discret (section 4.3.4), la version de la sémantique présentée ici décrit la version sans *synchronisation temporelle*, mais l'implémentation des algorithmes de générations de l'automate serveur gère ces synchronisations. Cette synchronisation temporelle des processus n'est pas présente dans XLANG mais existe dans BPEL.

Contrairement au temps discret, la version temps dense permet d'exécuter toutes les actions de chaque processus P_i en parallèle. La seule action à synchroniser est l'ensemble des actions de terminaison pour les sous-processus.

Actions individuelles : Dans un premier temps, si l'action est une exception, une action interne ou un *timeout*, l'action doit être exécutée de façon autonome :

$$\forall\, a \in E_X \cup \{\tau, to\},\ \frac{\exists\, j \in I,\ P_j \xrightarrow{a} P'}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{a} \text{flow}[\{P_i \mid i \in I\backslash\{j\}\} \cup \{P'\}]}$$

Dans un deuxième temps, si l'action ne correspond pas à une exception, une action interne ou un *timeout*, c'est-à-dire que l'action est un envoi ou une réception de message, l'action est exécutée :

$$\forall\, m \in M,\ \frac{\exists\, j \in I,\ P_j \xrightarrow{*m} P' \text{ et } \forall\, a \in E_X \cup \{\tau, to\},\ \neg\exists i \in I\backslash\{j\}\ \ (P_i \xrightarrow{a})}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{*m} \text{flow}[\{P_i \mid i \in I\backslash\{j\}\} \cup \{P'\}]}$$

Action de terminaison : si l'ensemble des sous-processus (P_i) peut terminer, alors le processus flow se termine.

$$\frac{\forall\, i \in I,\ P_i \xrightarrow{\surd} P'_i}{\text{flow}[\{P_i \mid i \in I\}] \xrightarrow{\surd} 0}$$

Le processus scope

Soit $M_I = \{m_i \mid i \in I\}$ un ensemble de messages, $E_J = \{e_j \mid j \in J\}$ un ensemble d'exceptions, et E_X l'ensemble des exceptions ($E_J \subseteq E_X$).

Le processus $\text{scope}(P, E)$ est défini par un processus P et l'expression :

$$E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$$

Le processus scope est associé à une horloge H permettant d'indiquer la durée d'exécution de ce processus. Son exécution est la suivante :

- l'exécution de P doit prendre au plus d unités de temps. Si ce n'est pas le cas, alors le processus Q est exécuté ;
- si une exception $e_j \in E_J$ est déclenchée, alors le processus R_j associé est exécuté ;
- si un message $m_i \in M_I$ non intercepté par P arrive, alors le processus P_i associé est exécuté.

Exécution du processus P : premier cas, comme pour la sémantique temps discret, si P se termine, alors le scope se termine également.

$$\frac{P \xrightarrow{\surd}}{\mathrm{scope}(P,E) \xrightarrow{\surd} 0}$$

Deuxième cas, si P peut exécuter une action (autre que l'action de terminaison ou une exception), alors le processus scope exécute cette action.

$$\forall\, a \notin \{\surd\} \cup E_X \cup M_I,\ \frac{P \xrightarrow{a} P'}{\mathrm{scope}(P,E) \xrightarrow{a} \mathrm{scope}(P',E)}$$

Gestion du *timeout* : si P ne se termine pas et qu'il exécute une action autre que la terminaison, qu'une exception ou qu'une action silencieuse, alors, le processus scope peut se terminer par un *timeout* (les gardes et conditions d'horloges seront expliquées plus loin).

$$\forall\, a \notin \{\surd, \tau\} \cup E_X,\ \frac{\neg\, P \xrightarrow{a}}{\mathrm{scope}(P,E) \xrightarrow{to} Q}$$

Réception d'un message $m_i \in M_I$: en recevant le message m_i, l'événement est traité et le processus scope arrêté.

$$\forall\, i \in I,\ \frac{\forall\, a \in E_X \cup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\mathrm{scope}(P,E) \xrightarrow{?m_i} P_i}$$

Traitement des exceptions : premier cas, l'exception e_j est prévue, le processus scope l'intercepte et exécute le processus R_j associé :

$$\forall\, j \in J,\ \frac{P \xrightarrow{e_j}}{\mathrm{scope}(P,E) \xrightarrow{\tau} R_j}$$

Deuxième cas, l'exception e n'est pas prévue, le processus scope passe dans un état de terminaison (processus *zombie*). Cette exception peut être interceptée à un niveau plus élevé (dans le cas d'un processus scope contenu dans un autre processus scope par exemple).

$$\forall\, e \notin E_J,\ \frac{P \xrightarrow{e}}{\mathrm{scope}(P,E) \xrightarrow{e} 0}$$

Remarque : en cas d'une exception jamais interceptée, le processus métier la soulevant est alors directement dérivé sur le processus *null*, comportement le plus proche de la réalité et laissant un élément extérieur gérer le problème. Le langage BPEL (contrairement à XLANG) prévoit ce cas : un processus peut être gardé globalement par un ensemble d'exception, et également un ensemble d'actions à exécuter en cas d'exception non prévue.

Le processus pick

Comme dans le cas du temps discret, le processus pick est considéré comme un cas particulier du processus scope : c'est un processus scope dont le processus principal P est défini comme étant le processus empty.

$$\text{pick}[E] = \text{scope}(\text{empty}, E)$$

$$\text{avec} \quad E = [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}$$

6.3.5 Génération de l'automate temporisé du service

Principe de génération de l'automate temporisé

La génération de l'automate temporisé est très similaire à celle du cas temps discret et une grande partie du code est commune (voir le chapitre 7 pour plus de détails). Voici une description des différentes étapes.

- La première étape consiste à générer le premier état de l'automate temporisé correspondant au processus métier dans sa globalité (tel que le décrit le fichier XLANG ou BPEL).
- À chaque étape, l'algorithme examine un processus et construit les arcs correspondant aux règles de la sémantique. Chaque nouvel état cible non encore présent dans l'automate est placé dans une liste d'états (associés à des processus) à examiner.
- Ensuite, l'algorithme détermine l'ensemble des horloges actives pour le processus courant (voir détails des algorithmes ci-dessous). En se basant sur les informations des états et arcs précédents, il détermine les invariants de l'état et complète les arcs *timeout*.
- L'ajout d'informations sur les arcs concernant la remise à zéro des horloges est réalisé à deux moments. Tout d'abord, si une transition a pour cible un état déjà analysé, l'information est obtenue directement et mise à jour. L'autre cas concerne les cibles non encore examinées : ces transitions seront mises à jour lors de l'examen de celles-ci.

La construction de l'automate temporisé de la partie service est terminée quand la liste d'états à traiter est vide.

Les horloges de l'automate temporisé

Une horloge H_i est associée à chaque processus scope (et par extension à chaque processus pick). Ensuite, l'algorithme 6.3.1 permet de déterminer si un processus donné a une ou plusieurs horloges actives (c'est-à-dire si un sous-processus scope est activé).

Plusieurs actions sont à réaliser lorsqu'un état possède au moins une horloge active :

- mettre un invariant sur l'état (du type $H_i \leq m_i$, m_i étant la valeur maximale que l'horloge peut atteindre suivant la définition du processus associé à H_i) ;
- si l'horloge (et donc le processus scope) est activée pour la première fois sur cet état, il faut ajouter une initialisation sur toutes les transitions ayant pour cible cet état ;
- si cet état possède des transitions sortantes étiquetées to, alors l'algorithme ajoute une garde d'horloge du type $H_i = m_i$ sur cette transition, et sur les autres transitions, il ajoute une garde du type $H_i < m_i$.

En cas de concurrence entre les horloges, un arbre d'horloge est réalisé, en correspondance avec l'arbre syntaxique des différents sous-processus du processus métier. La figure 6.2 représente un arbre syntaxique pour un processus flow exécutant, à un instant donné, en parallèle deux processus scope (nommé $scope_A$ et $scope_B$), le processus $scope_A$ contenant à son tour un processus scope nommé

Algorithm 6.3.1 LISTEHORLOGEACTIVE

Paramètres

Arbre : *graphe* (entrée)
ListeHorloge : *liste* (sortie)

Variables locales

ListeHorloge : *liste*

```
 1: ListeHorloge ← new liste() ;
 2: switch Arbre.racine of
 3:   case empty :
 4:   case ∗o[m] :
 5:   case throw :
 6:   case switch :
 7:   case while :
 8:     return null ;
 9:   case sequence :
10:     return LISTEHORLOGEACTIVE(Arbre.sousArbreGauche) ;
11:   case flow :
12:     for each LISTEHORLOGEACTIVE(Arbre.sousArbre) do ;
13:       ListeHorloge.add(LISTEHORLOGEACTIVE(Arbre.sousArbre)) ;
14:     end for each
15:     return ListeHorloge ;
16:   case scope :
17:     if √ ∉ EnsembleDesActionsPossibles(Arbre.sousArbreGauche) then
18:       ListeHorloge.add(Arbre.racine.getHorloge) ;
19:     end if
20:     ListeHorloge.add(LISTEHORLOGEACTIVE(Arbre.sousArbreGauche)) ;
21:     return ListeHorloge ;
22: end switch
23: return null ;
```

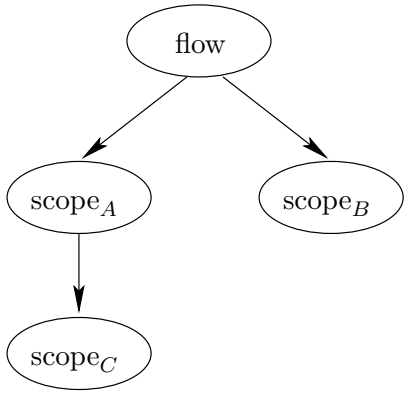


FIG. 6.2 – Exemples d'arbre syntaxique permettant de hiérarchiser les horloges.

$scope_C$. Les horloges associées aux processus $scope_A$, $scope_B$, $scope_C$ sont respectivement H_A, H_B, H_C et leurs valeurs maximales respectivement m_A, m_B, m_C.

Les processus $scope_A$ et $scope_B$ sont de même niveau (se sont des noeuds frères dans l'arbre) et de niveau supérieur à $scope_C$. L'invariant dans ce cas est $H_A \leq m_A \wedge H_B \leq m_B \wedge H_C \leq m_C$. Dans ce cas, si plusieurs *timeout* sont déclenchés en même temps, seuls ceux de plus haut niveau sont conservés (ici $scope_A$ et $scope_B$). Sur l'état correspondant au processus global, les conditions d'horloges se regroupent en plusieurs cas :

- aucun des processus ne peut être en *timeout* : $H_A < m_A$, $H_B < m_B$, $H_C < m_C$;
- $scope_C$ peut tomber en *timeout* à la condition que $scope_A$ ne le soit pas, et aucune condition sur $scope_B$, soit deux cas possibles :
  - $scope_B$ ne déclenche pas son *timeout*, alors $H_C = m_C \wedge H_A < m_A \wedge H_B < m_B$;
  - $scope_B$ déclenche son *timeout*, alors $H_C = m_C \wedge H_A < m_A \wedge H_B = m_B$.
- $scope_A$ peut tomber en *timeout*, donc surpasse $scope_C$, plusieurs cas :
  - $scope_A$ est tout seul à déclencher en *timeout* : $H_A = m_A \wedge H_B < m_B$;
  - $scope_A$ déclenche son *timeout* en même temps que $scope_B$: $H_A = m_A \wedge H_B = m_B$.
- $scope_B$ peut tomber en *timeout*, plusieurs cas :
  - $scope_B$ est tout seul à déclencher son *timeout* : $H_B = m_B \wedge H_A < m_1 \wedge H_C < m_C$;
  - $scope_B$ déclenche son *timeout* en même temps que $scope_C$ (cas déjà exploré : $H_C = m_C \wedge H_A < m_A \wedge H_B = m_B$) ;
  - $scope_B$ déclenche son *timeout* en même temps que $scope_A$ (cas déjà exploré : $H_A = m_A \wedge H_B = m_C$).

6.4 Relation d'interaction et synthèse de client

La relation d'interaction permet d'assurer le bon fonctionnement (au sens non blocage et non ambiguïté de l'interaction) du service web par le biais de l'automate temporisé de la partie service obtenu avec les algorithmes précédents. En utilisant cette relation d'interaction et un algorithme approprié, il est alors possible de générer l'automate temporisé de la partie cliente du service web.

Il est important de remarquer qu'un des points clés de l'algorithme de génération du client (algorithme que nous détaillerons plus loin) repose sur le fait que l'action de transition interne est une action immédiate, et que, d'après les règles précédentes, cette action τ ne se retrouve jamais en concurrence avec d'autres actions non immédiates. Ce sont donc ces ensembles d'états que l'algorithme de génération va agréger dans la partie cliente.

6.4.1 Relation d'interaction

L'automate temporisé modélisant le comportement observable d'un processus BPEL est une description compacte de celui-ci. Comme nous l'avons mentionné en section 3.1.3, la sémantique d'un automate temporisé est donnée par un système de transitions temporisés. Les états du système de transitions temporisé sont les configurations atteignables et les arcs sont soit des transitions discrètes (indépendante du temps) soit la modélisation d'un écoulement temps depuis un état de l'automate temporisé.

Rappelons qu'un *système de transitions temporisé* est défini par un tuple $(S, s_0, A, \rightarrow)$ tel que :

- S est un ensemble d'états ;
- $s_0 \in S$ est l'état initial ;
- A est un ensemble fini d'actions, disjoint de l'écoulement du temps ;

– $\rightarrow \subseteq S \times (A \cup \mathbb{R}_{\geq 0}) \times S$ est un ensemble de transitions.

Comme pour la relation de bisimulation, nous avons besoin d'une relation reliant les états des deux systèmes (l'automate temporisé du service et l'automate temporisé du client). Évidement, la paire d'états initiaux doit vérifier cette relation.

Les états de la paire doivent avoir une vue cohérente des possibilités d'événements et donc de l'interaction qui va suivre. Cela implique que la relation doit prendre en compte les étapes mutuellement observables de chacune des deux parties (client et service).

Pour cela, nous introduisons la définition suivante de transition observable :

– une transition est dite observable si au vu de l'extérieur, il est possible d'observer l'action a, même si cette action a est encadrée par des transitions inobservables. C'est-à-dire $s \stackrel{a}{\Rightarrow} s'$ si et seulement si $s \stackrel{\tau^* a \tau^*}{\rightarrow} s'$;
– une transition est non observable (silencieuse) dans le cas de plusieurs transitions τ successives. $s \stackrel{\epsilon}{\Rightarrow} s'$ si et seulement si $s \stackrel{\tau^*}{\rightarrow} s'$;
– enfin, une transition peut être détectée grâce à un écoulement de temps de durée d : $s \stackrel{d}{\Rightarrow} s'$ si et seulement si $s \stackrel{d_1 \tau \ldots \tau d_n}{\rightarrow} s'$ avec $\sum d_i = d$.

Une fois cela réalisé, la relation d'interaction indique que si un état S de la paire $(s,\, s')$ peut évoluer par une transition observable de son système de transitions temporisés vers un nouvel état s_1, s' doit avoir une transition observable sortante similaire vers un état s'_1, qui, associé à s_1, formera une nouvelle paire.

Par contre, si un des systèmes envoie un message, l'autre doit être capable de recevoir ce message. Il est alors nécessaire de définir la notion d'actions complémentaires :

– $\overline{?m} = !m$: le complément de la réception d'un message est l'envoi de ce message ;
– $\overline{!m} = ?m$: le complément de l'envoi d'un message est la réception de ce message ;
– $\forall\, a \notin \{!m\}_{m \in M} \cup \{?m\}_{m \in M}, \overline{a} = a$: le complément des autres actions est l'action elle-même.

Mais ces différentes conditions sont trop strictes et ne font pas la différence entre la nature d'un envoi et la nature d'une réception d'un message. Un envoi est une action alors qu'une réception est une réaction et n'arrive pas spontanément. Il est alors plus approprié que la relation d'interaction vérifie si, sur l'état s (de la paire (s, s')), le système peut recevoir le message m, alors, il doit y avoir un troisième état s'' de l'autre système (indistinguable depuis s' au sens des transitions observables) qui peut envoyer m et s' doit pouvoir envoyer un message (pas nécessairement m). La première condition permet d'exprimer que le système de transitions n'est pas capable de recevoir des messages qui ne sont pas prévus dans la spécification, et la deuxième condition permet d'assurer que les deux systèmes ne vont pas s'attendre indéfiniment.

Ces conditions sont traduites dans la définition formelle suivante.

Définition 6.4.1 (Relation d'interaction) Soient $A_1 = (S, s_{01}, A, \rightarrow_1)$ et $A_2 = (S, s_{02}, A, \rightarrow_2)$ deux systèmes de transitions temporisés. Alors A_1 et A_2 interagissent correctement si et seulement si $\exists \sim \subseteq S_1 \times S_2$ tel que :

– $s_{01} \sim s_{02}$
– $\forall\, s_1, s_2$ tels que $s_1 \sim s_2$
  – soit $a \notin \{?m \,|\, m \in M\}$;
    – si $\exists\, s_1 \stackrel{a}{\Longrightarrow}_1 s'_1$, alors $\exists\, s_2 \stackrel{\complement a}{\Longrightarrow}_2 s'_2$ avec $s'_1 \sim s'_2$ et
    – si $\exists\, s_2 \stackrel{a}{\Longrightarrow}_2 s'_2$, alors $\exists\, s_1 \stackrel{\complement a}{\Longrightarrow}_1 s'_1$ avec $s'_1 \sim s'_2$

- soit $m \in M$; si $s_1 \stackrel{?m}{\Longrightarrow}_1 s'_1$ alors
  - $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2$, $\exists\, s_2^- \stackrel{w}{\Longrightarrow}_2 s_2^+$, $\exists\, s_2^+ \stackrel{!m}{\Longrightarrow}_2 s'_2$ avec $s_1 \sim s_2^+$ et $s'_1 \sim s'_2$ tel que w est un mot
  - $\exists\, s_2 \stackrel{!m'}{\Longrightarrow}_2 s'_2$
- soit $m \in M$; si $s_2 \stackrel{?m}{\Longrightarrow}_2 s'_2$ alors
  - $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1$, $\exists\, s_1^- \stackrel{w}{\Longrightarrow}_1 s_1^+$, $\exists\, s_1^+ \stackrel{!m}{\Longrightarrow}_1 s'_1$ avec $s_1^+ \sim s_2$ et $s'_1 \sim s'_2$ tel que w est un mot
  - $\exists\, s_1 \stackrel{!m'}{\Longrightarrow}_1 s'_1$

6.4.2 Algorithme de synthèse

Un service est déclaré ambigu quand l'algorithme ne peut assurer la construction d'un client : en effet, ce client doit être implémentable, donc il doit être *déterministe*. Si ce n'est pas le cas, alors le service est déclaré ambigu. De plus, comme il doit avoir ses propres horloges pour gérer les *timeout* du serveur, nous décrivons son comportement par un système de transitions temporisé également. Ces considérations mènent à choisir le modèle des automates temporisés pour notre client. La section suivante présentera la génération de l'automate temporisé client qui sera en relation avec l'automate temporisé du processus BPEL.

En plus des cas d'*ambiguïté* vus dans le cadre du temps discret qui sont transposables dans le cadre du temps dense (voir section 4.4.2), il existe un cas d'ambiguïté supplémentaire due à la non discrétisation du temps, nommé ambiguïté temporelle (voir section 6.4.3).

Pour rappel, le processus $\mathrm{switch}[?o[m], ?o[m']]$ est ambigu car le choix de la branche à exécuter est interne au service et se fait de façon non déterministe. Le client n'a alors aucune information quand au choix de cette branche et ne sait pas s'il doit envoyer m ou m'. Par contre, le processus $\mathrm{switch}[!o[m], !o[m']]$ n'est pas ambigu, le client peut attendre indifféremment les deux messages, il exécutera alors la branche correspondant au message reçu.

Principe

L'algorithme ne consiste pas à déterminiser un automate temporisé, mais restreint sa recherche à un automate temporisé qui possède les mêmes horloges que l'automate temporisé du processus BPEL. Si l'algorithme déclare le service ambigu, cela revient à dire qu'il n'existe pas d'automate client de ce service, ayant cette contrainte d'horloges identiques. Notre approche est donc encore perfectible, mais cette restriction semble raisonnable (voir plus loin pour plus de détails).

Le principe général de notre algorithme est schématisé sur la figure 6.3. Un état de l'automate temporisé du client correspond à un sous-ensemble d'états de l'automate temporisé du service (voir la figure 6.4).

Plus précisément, chaque état potentiel s de l'automate temporisé du client est associé à un ensemble d'états $S_2(s)$ de l'automate temporisé du service lesquels sont en relations avec s par la relation d'interaction. Durant la construction, un pile est utilisée pour stocker les états du client restant à traiter. Au départ de l'algorithme, la pile contient seulement l'état initial du client s_{01} tel que $S(s_{01}) = \{s_{02}\}$, l'état s_{02} est l'état initial du service. L'algorithme termine lorsque la pile est vide (c'est-à-dire le client a été construit) ou quand il a détecté une ambiguïté dans le service.

La première étape consiste à calculer la $\tau - clôture$

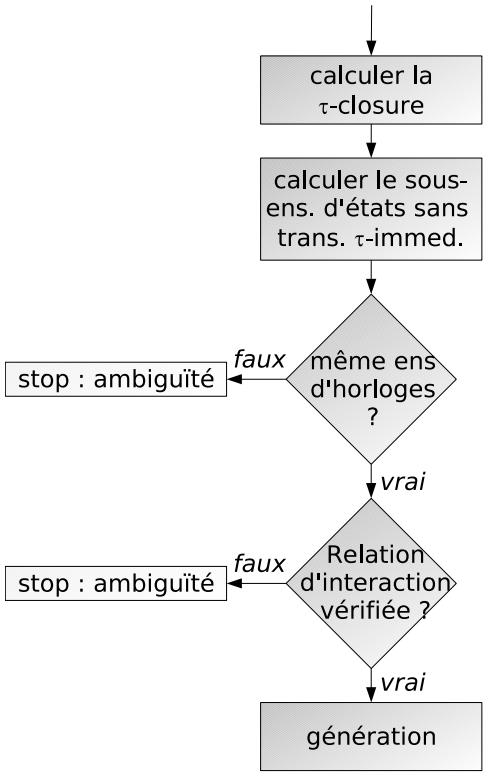


FIG. 6.3 – Une étape de l'algorithme de synthèse du client.

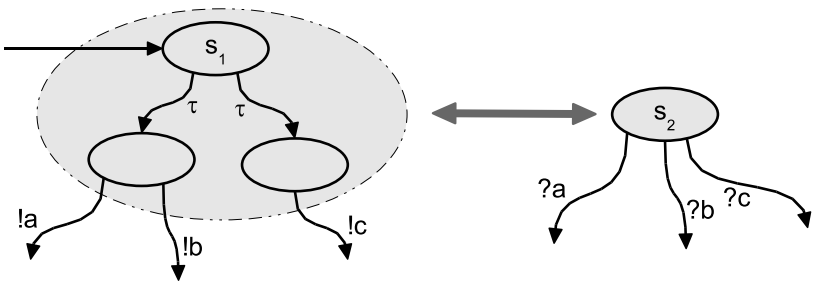


FIG. 6.4 – L'ensemble d'états du service (à gauche) en relation avec l'état du client (à droite)

par les transitions τ (l'ensemble des états accessibles depuis l'état que l'on traite par des τ). Si ce sous-ensemble (appelé S') des états du service est déjà associé à un état s du client, alors, les transitions de l'automate temporisé du client, qui ont généré le sous-ensemble, sont redirigées vers s. Dans le cas contraire, un nouvel état est créé (appelé new).

Ensuite, l'algorithme calcul le sous-ensemble d'états de S' (appelé S'') contenant seulement des états qui ne possèdent pas de transitions τ sortantes.

Ensuite, il calcul l'ensemble des horloges de chaque état de S''. Si cet ensemble d'horloges n'est pas unique, alors il déclare *ambiguïté* (ambiguïté temporelle plus précisément dans ce cas) et arrête la génération. Dans le cas contraire, il vérifie la relation d'interaction pour les transitions discrètes. Si la relation n'est pas vérifiée, alors la construction est stoppée pour ambiguïté.

Enfin, la construction des gardes d'horloges se fait en deux étapes. La première consiste à faire une copie des gardes et des invariants d'horloges des états de l'automate temporisé serveur, et la deuxième, à compléter les transitions nouvellement construites.

Algorithme

L'algorithme 6.4.1 détaille les différentes étapes présentées précédemment. Il utilise les principaux types suivants :

- *graph* : le type de données abstrait *graphe* et ses méthodes associés (addVertex, addEdge, getInitialState, setInitialState, getStates) ;
- *clientState* : un état du client et ses méthodes associées (getSet, setSet) ;
- *clientState* : un état du serveur et ses méthodes associées (getSet, setSet) ;
- *stack* : le type de données abstrait pile et ses méthodes associées (isEmpty, pop, push).

La fonction $interacRelationVerified(graph, set\ of\ serverStates)$, utilisée à la ligne 25, vérifie si un état client peut être créé tel qu'il sera en relation d'interaction avec chaque état de l'ensemble associé (voir la section 6.4.1). Elle retourne un booléen (*vrai* si la relation d'interaction est vérifiée, *faux* sinon).

6.4.3 Ambiguïté et ambiguïté temporelle

Comme nous l'avons précisé plus haut, notre algorithme est restrictif dans le sens où il déclare ambigu (temporellement) certains services qui en fait en le sont pas. Voici un exemple de fausse détection d'ambiguïté (voir la figure 6.5, page 124) correspondant au processus :

$$\mathrm{switch}(!o[c], \mathrm{scope}(!o[a], [\{(?b, \mathrm{empty})\}, (4, \mathrm{empty}), \{\}])$$

Ce processus commence par un processus switch dont une branche est un processus scope et l'autre branche n'a pas de processus contraint par une horloge. L'ambiguïté vient de ces deux branches : comment détecter quand le serveur va initialiser l'horloge, et surtout, s'il va l'initialiser ! Car s'il décide d'exécuter la branche sans horloge, l'initialisation n'est pas nécessaire...

Par contre, en temps discret, notre ancienne méthode permettait de générer un tel client. En effet, l'écoulement de temps étant symbolisé par une action (χ), le modèle reposait sur un niveau différent que les automates temporisés utilisés ici. L'algorithme « restrictif » est le prix à payer pour obtenir une représentation plus compacte du serveur et du client.

6.4.4 Exemples

Cette section présente deux automates temporisés générés par les algorithmes énoncés précédemment : l'un pour un service que nous allons décrire et l'autre pour le client correspondant à ce service.

Algorithm 6.4.1 SYNTHESEDUCLIENT

Paramètres
GServer : *graph* (entré)
GClient : *graph* (entré et sortie)
Variables locales
cs, cs' : *clientState* ; ss, ss' : *serverState*
set, set', front, oldset : *set of serverState*
sa : *set of clocks*
st : *set of transitions* ; t : *transition*
Stk : *stack*

```
 1: new(cs)
 2: cs.setSet({GServer.getInitialState})
 3: Gclient.addVertex(cs) ; Gclient.setInitialState(cs)
 4: Stk.push(cs)
 5: while not Stk.isEmpty() do
 6:    cs ⇐ Stk.pop()
 7:    set ⇐ s.getSet()
 8:    // calcule τ-clôture
 9:    front ⇐ set
10:    while not front.isEmpty() do
11:       oldset ⇐ set
12:       for each ss ∈ front do
13:          set ⇐ set ∪ {ss' | ∃ ss -τ-> ss'}
14:       end for
15:       front ⇐ ss \ oldss
16:    end while
17:    if ∃ s' ∈ GClient.getStates | set = s'.getSet() then
18:       // si un état équivalent existe, le fusionner
19:       rediriger tous les arcs entrant de s sur s' dans GClient
20:    else
21:       cs.setSet(set)
22:       set' ⇐ {ss' ∈ set | ¬ ss' -τ->}
23:       sa ⇐ set'.getfirst().getActiveClocks()
24:       if ∀ ss' ∈ set', ss'.getActiveClocks() == sa then
25:          if interacRelationVerified(GServer, ss) then
26:             // création des nouvelles transitions
27:             st ⇐ {t | ∃ s ∈ ss, ∃ s -t-> s', t ≠ τ}
28:             for each t ∈ st do
29:                set' ⇐ {ss' | ∃ ss ∈ set, ss -t-> ss'}
30:                new(cs')
31:                cs'.setSet(set')
32:                Gclient.addVertex(cs')
33:                Stk.push(cs')
34:                GClient.addEdge(cs, t̄, cs')
35:             end for
36:          else
37:             return ambiguity // ambiguïté liée à l'interaction
38:          end if
39:       else
40:          return ambiguity // ambiguïté temporelle
41:       end if
42:    end if
43: end while
```

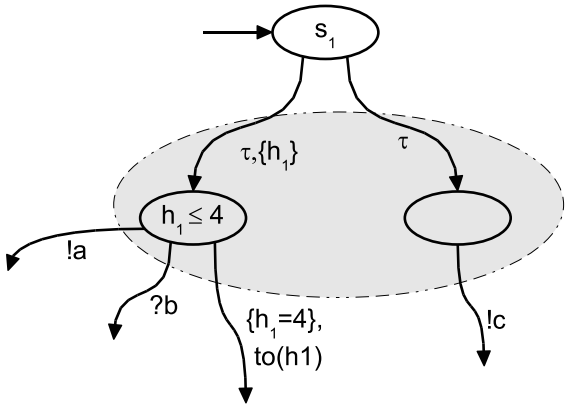


FIG. 6.5 – Exemple d'ambiguïté temporelle.

Automate temporisé du serveur

Le processus serveur que nous voulons modéliser est le suivant :

$$?Start; \text{scope}(\text{while}[!Question]; !End, [\{(?Evt, !Evt)\}, (H1 : 2, !TimeOut), \{\}])$$

Ce processus commence par recevoir un message $Start$. Il exécute ensuite un processus scope dont le temps maximal d'exécution est fixé à 2 unités de temps. En cas de dépassement de ce temps, le message $TimeOut$ est envoyé. Le processus scope gère également l'arrivée d'un événement dont le message s'appelle Evt (il renvoie alors le même message). Enfin, le corps du processus est une boucle while envoyant le message $Question$. Après l'exécution de cette boucle, le message End est envoyé, qui correspond également à la fin du processus scope et, par la même occasion, du serveur.

En appliquant récursivement et sur chaque nouvel état les différentes règles de notre sémantique formelle, nous obtenons l'automate temporisé du processus représenté à la figure 6.6.

Automate temporisé du client

La figure 6.7 donne la représentation de l'automate temporisé du client, généré par notre algorithme de synthèse de client, et correspondant au processus serveur présenté ci-dessus.

Nous pouvons remarquer que certains états du serveur ont été fusionnés dans l'automate temporisé du client.

En fait, l'état appelé (2) dans le serveur est présent dans le client avec la même étiquette (2). Le seul changement concerne la transition sortante : dans le serveur, il reçoit le message $Start$ et dans le client, il envoie le message $Start$.

L'état étiqueté (3) dans le serveur est « *plus complexe* ». Le calcul de la $\tau - clôture$ dans l'algorithme de synthèse retourne l'ensemble d'états $\{(3), (4), (5)\}$. Cet ensemble d'états est important pour le serveur mais le client ne peut connaître l'état courant du serveur : aucun message n'est échangé en passant de l'état (3) à (4) ou (5), ces transitions représentent seulement des actions internes au serveur. Donc, ces états du serveur sont fusionnés en un seul état du client : l'état (3). Les différentes transitions sortantes de l'ensemble sont adaptées à cette fusion dans le client.

Les autres états ont une génération très similaire à l'état (2).

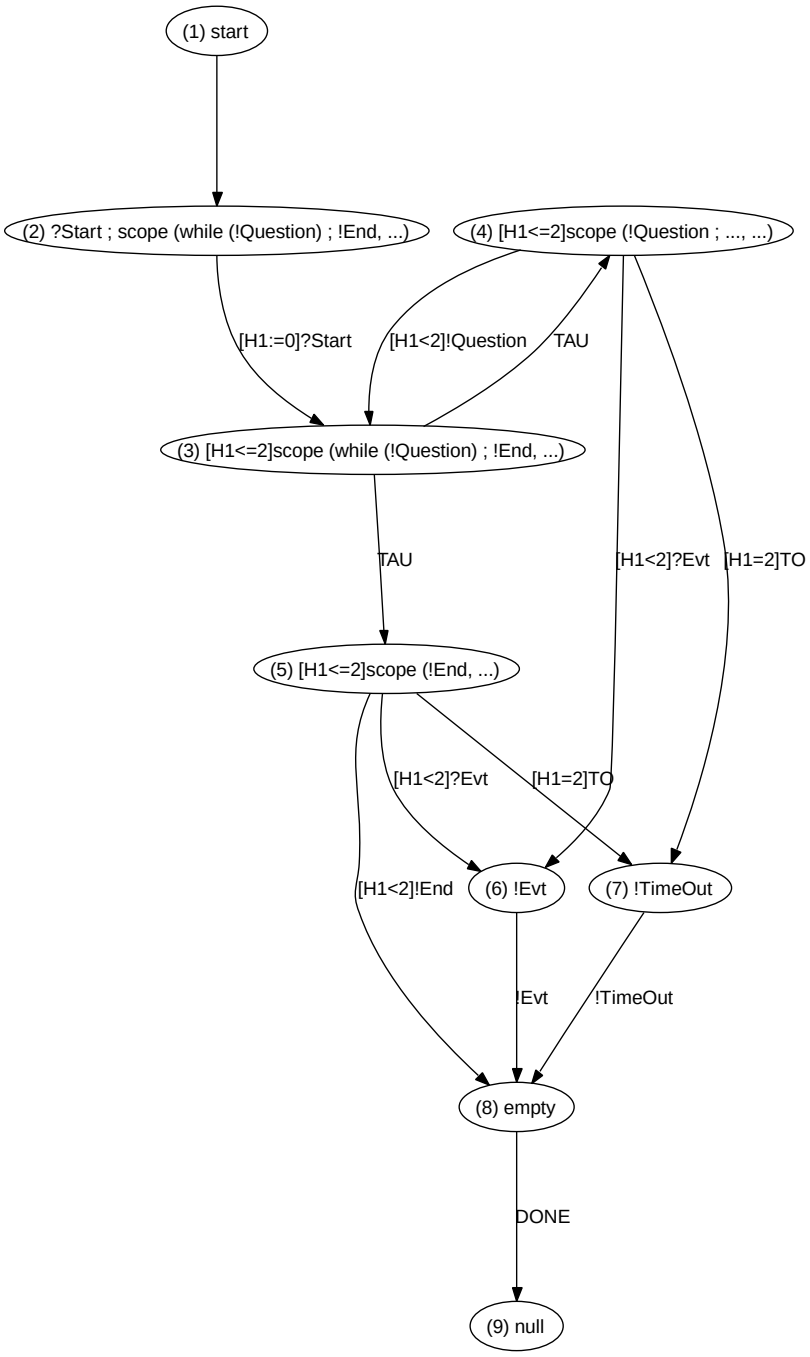


FIG. 6.6 – Automate temporisé du service (processus : $?Start; \mathrm{scope}(\mathrm{while}[!Question]; !End, [\{(?Evt, !Evt)\}, (H1 : 2, !TimeOut), \{\}])$)

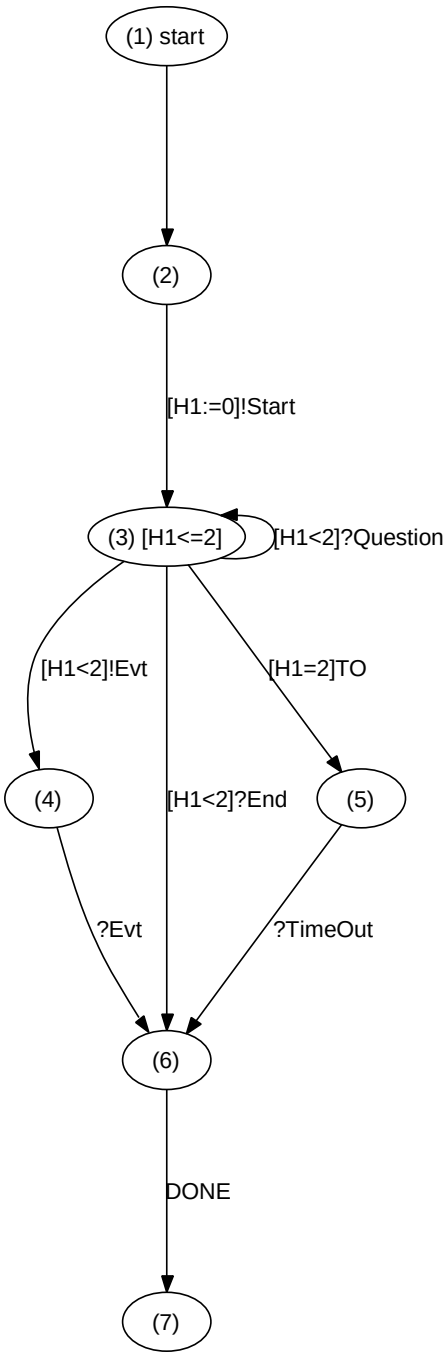


FIG. 6.7 – Automate temporisé du client correspondant au processus de la figure 6.6

6.5 Synthèse

Ce chapitre montre l'évolution de la formalisation en temps discret, vue dans le chapitre 4, vers la formalisation en temps dense. Cela permet d'apporter une solution à la possible explosion du nombre d'états des modèles de la méthode précédente dans le cas de nombreuses gardes de temps imbriquées. Nous avons ainsi ajouté une action supplémentaire représentant l'expiration du délai de garde d'une horloge. Le modèle utilisé n'est plus le TIOTS mais les automates temporisés, très utilisés dans le cas d'une modélisation en temps dense par son mécanisme d'horloges à valuation continue.

Les adaptations concernant la formalisation des langages de description comportementale sont également présentées, suivi des modifications concernant la génération du modèle (ici un automate temporisé) de la partie serveur du service web ; mais également les adaptations sur la relation d'interaction pour tenir compte du temps dense : par exemple, les ensembles d'horloges des états en interaction doivent être identiques. Enfin, la génération de l'automate temporisé représentant le client du service web est présenté.

Ouvertures et perspectives Notre méthode en temps dense, présente cependant un petit défaut par rapport à la méthode en temps discret : certains services sont déclarés ambigus alors que la modélisation temps discret les acceptaient. Cela vient de la concurrence possible, au niveau du serveur, au moment du choix de l'exécution d'une des deux branches (ou plus) dont l'une exécute immédiatement un processus gérant le temps et l'autre non. Le client ne peut savoir au préalable, quelle branche du service sera réellement exécutée et ne peut décider s'il doit initialiser et gérer l'horloge ou non. Cette ambiguïté est alors déclarée ambiguïté temporelle et reste une question ouverte.

Comme dans le cadre temps discret, l'approche temps dense, modélisant le comportement d'un service, peut être étendue aux partenaires d'une chorégraphie. Chaque partenaire, étant à la fois service et client d'un autre partenaire, une relation d'interaction adaptée peut être vérifiée sur les modèles ainsi générés. Cette méthode de vérification de l'interaction interne d'une chorégraphie, dans le formalisme temps dense, est une adaptation envisageable.

Chapitre 7

Implémentation et mise en œuvre

Ce chapitre présente les aspects et les choix technologiques liés à l'implémentation des algorithmes détaillés précédemment, en ce plaçant dans le cadre de la réalisation d'un ensemble d'outils permettant l'invocation de services web. Ces services web devront proposer à l'utilisateur une description comportementale du service métier associé, et ils seront également supposés non contrôlables par ces nouveaux outils. Les objectifs du code généré sont multiples :

- travailler dans le contexte des services web est synonyme de s'adapter à l'évolution rapide de ces technologies ; ce qui entraîne l'écriture de code le plus générique et modulaire possible ;
- comme nous avons pu le remarquer dans la bibliographie, le concept de client générique pour des services web est peu présent (voir absent) dans la littérature, ce qui souligne l'intérêt de notre réalisation.

Ce client générique sera capable d'invoquer des services web disponibles sur un réseau et sans aucune maîtrise sur le déroulement interne de ceux-ci. De plus, il permettra de répondre à un besoin de réutilisation de services web existant sans pour autant développer un client spécifique.

Pour tenir compte de l'aspect générique de l'approche de formalisation et de modélisation présentés dans les chapitres précédents, les outils développés ne seront pas liés à un langage de description comportementale donné. Une transformation préalable de la description sera réalisée pour obtenir, par exemple, un arbre syntaxique indépendant. Ainsi, seul ce module de traduction est dépendant du langage initial, limitant la quantité de code impactée lors d'un changement, même majeure, de technologie.

Nous allons voir, dans une première partie, le contexte nous menant à l'écriture des algorithmes et du client générique. Dans une deuxième partie, nous aborderons le développement de l'outil permettant de générer les modèles serveur et client, et enfin, dans la troisième partie, nous présenterons l'outil « client générique » consistant à invoquer le service web en suivant l'exécution dictée par l'automate temporisé produit par l'outil de modélisation.

7.1 Présentation

La présentation de la mise en œuvre des algorithmes présentés dans les chapitres précédents, se concentre, dans un premier temps, sur le cadre et les objectifs du développement d'une plateforme de services web, suivis, dans un deuxième temps, par la présentation de quelques serveurs permettant de déployer et d'exécuter des services web possédant une description comportemantale.

7.1.1 Plateforme de services web

Les travaux présentés ici s'intègrent dans le cadre d'une plateforme de services web développée principalement par une équipe du laboratoire LAMSADE de l'Université Paris Dauphine. Ce développement est motivé par les enjeux technologiques et formels liés au domaine concerné, permettant de répondre aux questions concernant la vérification des services web, ou encore la fourniture d'outils permettant une composition de services web (la plus automatique possible).

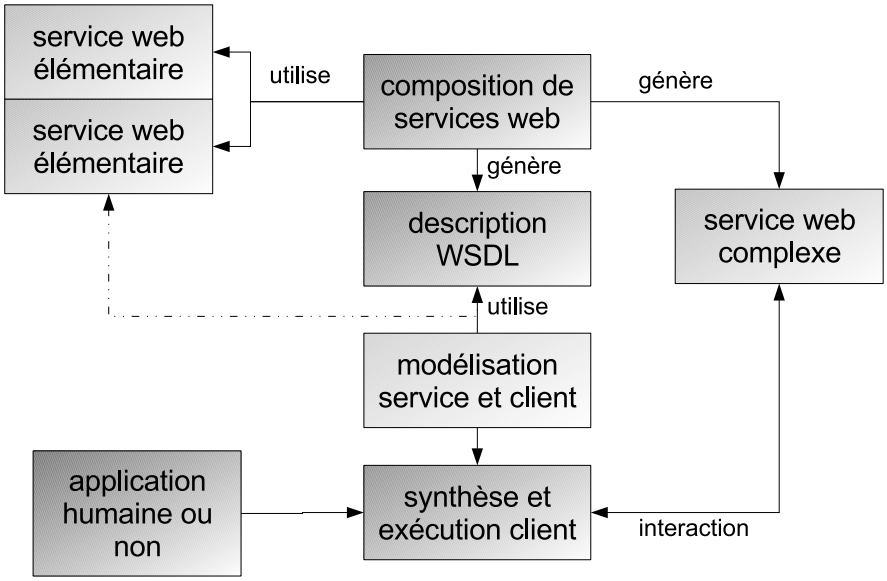


FIG. 7.1 – Plateforme de services web.

Nous nous concentrerons tout particulièrement sur les étapes « modélisation service et client » ainsi que « synthèse et exécution client » (voir figure 7.1). Ces étapes ont été personnellement développées dans le cadre de cette thèse, en s'assurant des aspects d'indépendances du langage de description comportementale et l'implémentation des règles issues de la formalisation de ces langages, permettant ainsi une évolution et une maintenance aisée en cas de changements dans les technologies supportées ou dans la formalisation associée. Quelques précisions sur les autres modules :

- les modules de « *composition de services web* » et génération de la « *description WSDL* » seront détaillés plus loin et sont développés au LAMSADE ;
- le module « *service web complexe* » est un service web généré par le module « *composition de services web* » ;
- les « *services web* » élémentaires sont supposés être des services web existant.

Pour obtenir la plus grande portabilité, le langage Java a été retenu. De plus, de nombreux outils du domaine des services web étant développés principalement en Java, l'utilisation de Java nous permet de rester le plus proche de ceux-ci. Le tout, en utilisant au maximum les outils et bibliothèques libres de droit, permettant ainsi de s'affranchir des problèmes d'un outil non porté sur une plateforme donné ou dont le développement n'est plus assuré.

Avant d'aller plus loin...

Avant de vouloir aller plus loin dans une composition automatique de services web avec validation de certaines propriétés pour assurer que l'interaction se déroulera sans soucis, il est nécessaire de disposer d'un module permettant d'invoquer n'importe quel service web répondant à certains critères.

Ce travail part d'un constat : l'invocation automatique d'un service web *basique* disposant d'une description WSDL est réalisable relativement aisément avec les outils actuels. Par contre, l'invocation automatique d'un service web *complexe* disposant d'une description comportementale avec un langage tel que BPEL n'est pas aussi facilement réalisable avec les outils actuels. L'approche utilisée par les industriels consiste souvent à écrire du code guidant cette invocation pour un service web donné, ce code est alors trop proche de ce service web et n'est pas réutilisable pour un autre service web ou même une nouvelle version de celui-ci.

Description des modules de la plateforme

Nous allons maintenant présenter les différents modules de notre plateforme (voir figure 7.1) :
- les modules « modélisation du service et du client » et « invocation du client » ont été personnellement développés dans le cadre de cette thèse ;
- le module « module de composition de service web » sort du cadre de cette thèse et son développement est en cours au LAMSADE.

La modélisation du service et du client Le module de génération du modèle de la partie service suivie de la génération du modèle de la partie cliente est appelée « *WSMod* ». Il sera présenté en détail dans la section 7.2. C'est, de loin, le module le plus important en terme de développement, d'intérêt et de fonctionnalités.

Suite à une première implémentation réalisée lors de mon stage de DEA, en temps discret, que l'on qualifiera de prototype, le besoin d'utiliser des principes génériques permettant la plus grande adaptation lors des changements aussi bien de l'ordre technologique (par exemple migration du langage XLANG au langage BPEL) que formel (par exemple, les règles sémantiques définies dans les chapitres précédents ont été adaptées très régulièrement pour répondre à nos exigences ou corriger quelques imperfections, ou encore lors de l'adaptation de nos travaux au temps dense) furent nécessaires. C'est donc sur la règle de la généricité que nous détaillerons le développement de ce module dans la section 7.2, développement réalisé dans le cadre de cette thèse.

Le principe est le suivant : après avoir obtenu la description de l'interface (WSDL) et du comportement (BPEL ou XLANG), l'outil exécute les différents algorithmes pour obtenir, dans un premier temps, l'automate temporisé ou le TIOTS du service. Puis, si l'interaction est possible (sans ambiguïté) avec ce service, un modèle équivalent du client est généré.

Invocation du client Les détails du module d'exécution dirigée par le modèle du client seront présentés dans la section 7.3. Ce module récupère les modèles générés par l'outil de modélisation de services web et appelle une bibliothèque d'invocation d'opérations de services web en fonction des transitions franchies dans le modèle.

La bibliothèque d'invocation a été réalisée en plusieurs étapes :
- la première étape consiste à récupérer la description WSDL d'un service web basique pour permettre à l'utilisateur de choisir lui-même les opérations à invoquer, décrites dans cette description WSDL. Cette partie a été réalisée par Tarak Melliti pendant sa thèse de Doctorat, en

s'appuyant sur les outils tel que WSDL2Java (outil permettant de générer des classes Java regroupant les différents types et méthodes décrits par le fichier WSDL).
- la deuxième étape a été réalisée, par moi-même, en même temps que le prototype de l'outil de modélisation de certains services web se rapprochant des services web basiques. L'exécution du module consiste à appeler les différentes méthodes de l'outil d'interfaçage décrit lors de la première étape, en suivant les différentes transitions du TIOTS et en demandant l'intervention de l'utilisateur pour remplir les données ou en cas d'alternative.
- enfin, la troisième étape, encore en développement, consiste à suivre l'exécution dictée par l'automate temporisé client d'un service web et d'utiliser une nouvelle bibliothèque d'invocation, développée personnellement pour l'occasion suite à des changements technologiques trop important liés à l'utilisation de services BPEL.

Le module de composition de service web Ce module dépasse le cadre des travaux présentés ici. Il est actuellement développé par l'équipe du LAMSADE et principalement par Demba Coulibali et Céline Boutrous-Saab. Ils ont écrit un langage proche de Java et BPEL permettant de composer rapidement des services web sans se soucier, pour le programmeur de services web complexes, des besoins liés aux différentes technologies nécessaires dans ce cas. Ensuite, un outil permet de générer :
- du code exécutable, devenant le vrai service web dit complexe ;
- deux fichiers de description : l'un concernant la description de l'interface du service web complexe (description WSDL), et l'autre concernant la description comportementale de celui-ci (description BPEL).

7.1.2 Plateformes d'exécution côté serveur

Nous n'allons pas faire ici l'inventaire de toutes les plateformes d'exécution de services web, mais seulement présenter trois plateformes que nous avons sélectionnées. Les critères de sélections sont basés sur leur disponibilité, leur état d'avancement dans leur développement et la communauté gravitant au tour de ces produits. Ces plateformes évoluent très vite et étaient quasiment inexistantes au début de nos travaux.

JBoss

JBoss[1] est, à la base, un serveur d'application *J2EE* et ses bibliothèques associées, entourés d'une large communauté et de projets *open source*. La société fondatrice a été racheté récemment par la société RedHat[2].

JBoss a l'avantage de proposer une version d'Eclipse modifiée et adaptée pour développer directement avec ce serveur d'application. Eclipse[3] est un IDE, sous la licence libre EPL[4], extensible à l'aide de *plugins*. JBoss propose donc un paquetage tout en un avec Eclipse, le serveur d'application et les bibliothèques de développement associés, permettant de s'affranchir de la configuration de chacun des outils.

Au moment de réaliser les tests nécessaires à l'implémentation du modèle d'exécution client, JBoss fournissait deux extensions nous concernant, à son serveur d'applications :

[1] `http://www.jboss.com/`
[2] `http://www.redhat.com/`
[3] `http://www.eclipse.org/`
[4] `http://www.eclipse.org/org/documents/epl-v10.php`

– JBpm (*Java Business Process Management*, sous licence LGPL, permet de développer des services web basés sur le *workflow* ou le *BPM* (*Business Process Management*), c'est-à-dire utilisant les méthodes d'orchestration des différents sous-processus définissant le service web composite. Ce module est totalement opérationnel et fonctionnel, mais n'est pas lié spécifiquement à BPEL et ne gère donc pas des services BPEL.
– l'extension JBpm-BPEL, quant à elle, permet d'orchestrer les services web BPEL. Malheureusement, cette extension n'a pas encore, à l'heure de rédiger ces lignes, atteint le stade de version finale. Lors des tests, ce moteur montrait déjà certains avantages, liés à l'environnement de développement, mais aussi quelques défauts liés à sa jeunesse, rendant impossible en l'état, son exploitation avec notre développement. Par contre, dans un futur très proche, cet outil sera très utilisable.

Oracle BPEL Process Manager

Oracle propose l'outil *BPEL Process Manager*[5] permettant de concevoir et exécuter des processus BPEL.

L'outil de conception, *BPEL Designer*, à l'avantage de travailler directement sur les descriptions BPEL en mode natif. Il est donc le plus proche du standard BPEL. Il s'interface aux éditeurs Java connus tel que Eclipse ou JDeveloper. Le moteur d'exécution, *BPEL engine*, exécute directement les services BPEL, avec la sauvegarde de l'état du serveur dans une base de données, étape nécessaire lors d'une interaction longue entre plusieurs processus ou services.

C'est un très bon produit, disponible sous Windows et sous certains Linux (à la condition d'avoir la version exacte des outils dont il dépend, donc difficilement évolutif dans le temps).

ActiveBPEL

Le serveur d'applications inclus dans les outils ActiveBPEL[6] est un serveur sous licence libre. Il est agrémenté par des outils, développés par Active Endpoints, de conception et de développement de services BPEL qui eux sont payants.

Il repose sur le serveur Tomcat[7] d'Apache, libre également. L'ajout à Tomcat du serveur ActiveBPEL est relativement simple. De plus, une fois déployé, il possède une interface d'administration entièrement utilisable par une interface web, mais également par des services web fournis avec. Cette interface permet le déploiement et la gestion des services sur le serveur.

La version 2.0 vient tout juste de sortir et respecte pour le moment la spécification BPEL dans sa version 1.1.

Environnement de test

Notre environnement personnel de test est basé sur le serveur ActiveBPEL pour des raisons pratiques concernant l'installation et la configuration sur les différenrtes distributions Linux utilisées pour nos recherches. Bien évidement, notre plateforme du côté client ne se limite pas à ce serveur, mais fonctionne avec tout les serveurs respectant le standard BPEL.

[5] `http://www.oracle.com/technology/products/ias/bpel/index.html`
[6] `http://www.activebpel.org/`
[7] `http://tomcat.apache.org/`

7.2 L'outil WSMod

WSMod est un outil développé en Java, dont le développement a été assuré personnellement dans le cadre de cette thèse. Il contient 12000 lignes de codes pour 70 classes réparties dans 5 paquets Java. Son but est de modéliser la partie service puis d'utiliser les algorithmes de synthèse du client présentés dans les chapitres précédents pour générer le modèle de la partie cliente, en cas de non ambiguïté du service.

7.2.1 Principe

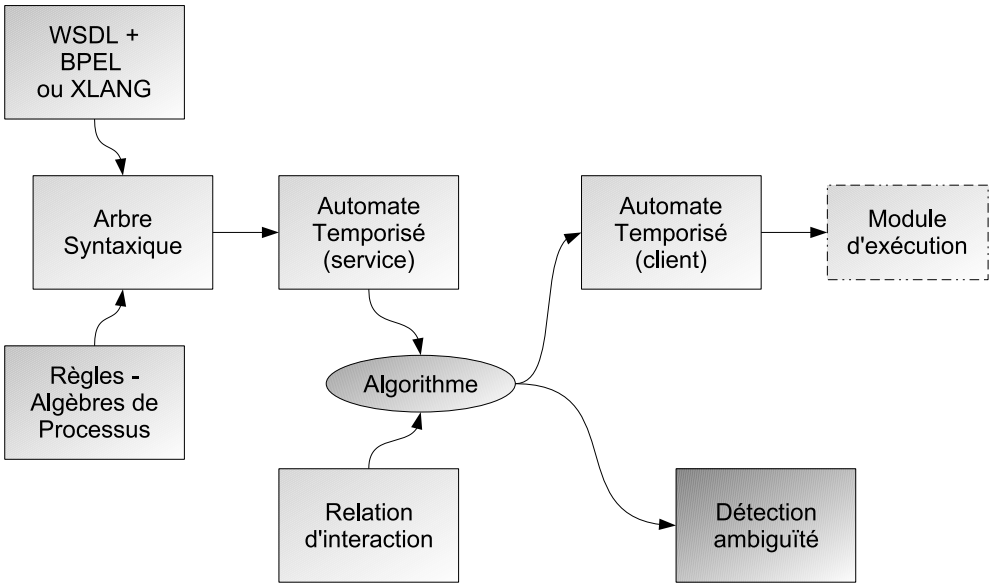


FIG. 7.2 – Architecture de l'outil WSMod.

Nous allons décrire les principes des différentes étapes de l'outil WSMod lui permettant de générer les différents modèles à obtenir pour un service web donné (voir figure 7.2). Les différents modules, dont le fonctionnement est détaillé dans les paragraphes suivants, sont :

- « *WSDL + BPEL ou XLANG* » correspond aux descriptions des services web récupérés depuis Internet ;
- « *Règles Algèbres de Processus* » correspond au fichier contenant les règles de la sémantique ;
- « *Arbre syntaxique* » correspond à la représentation intermédiaire du processus métier du service web, nécessaire pour l'indépendance de BPEL ou XLANG ;
- « *Automate Temporisé service* et *client* » correspond à la génération des modèles services et clients ;
- « *Algorithme*, *relation d'interaction* et *détection d'ambiguïté* » correspond aux étapes de la génération du modèle client, avec possibilité de débouchés sur une ambiguïté ;
- « *Module d'exécution* » correspond au branchement de WSMod vers ExecClient permettant l'exécution guidée du client.

Les sections suivantes présenterons les détails d'implémentation de ces modules.

Récupération des fichiers de description du service

La première étape consiste, pour WSMod, à récupérer les fichiers de description du service. La version actuelle est capable de gérer les services web possédant une description WSDL accompagnée d'une description comportementale utilisant le langage XLANG ou BPEL. Il est possible de l'adapter à d'autre langages de description comportementale.

Transformation en arbre syntaxique

Les descriptions du service obtenues à l'étape précédente sont ensuite transformées en *arbre syntaxique*. Cet arbre syntaxique est une passerelle entre la description du service web et nos algorithmes de génération de modèle de la prochaine étape. En cas de changement d'une des deux parties, l'arbre syntaxique permet d'avoir une modélisation commune évitant de modifier l'autre partie (par exemple, changement du langage de description comportemental du service ou passage du modèle des TIOTS au modèle des automates temporisés).

L'arbre syntaxique est composé d'objets représentant les entités mises en évidence lors de notre formalisation. Les éléments présents sont :

- les éléments de base : time, empty, input, output, throw ;
- les éléments de contrôle structuré : $\mathrm{sequence}$, while, switch ;
- les éléments de contrôle évolué : flow, scope.

L'analyse des expressions contenues dans le fichier de description, à l'aide du langage *XPath* [CD99], permet ensuite de transformer cette description en un arbre syntaxique, qui lui, est indépendant de tout langage.

Les règles d'évolution de chacun de ces éléments sont dictées par un fichier de règles (voir section 7.2.2), permettant ainsi de générer l'automate temporisé.

Génération de l'automate temporisé de la partie service

Grâce aux règles, il est possible de décrire l'évolution d'un processus modélisé. Ainsi, la première étape de modélisation du processus représentant le service web correspond à la construction de l'arbre syntaxique, auquel WSMod applique les différentes règles récursivement décrites dans le fichier de règles. Ceci permet d'obtenir l'ensemble des actions que le processus peut réaliser, et leurs processus dérivés. Comme nous le verrons dans la section 7.2.3, en appliquant le même principe aux nouveaux processus dérivés, WSMod construit la modélisation complète du service web. C'est de loin, la majeure partie du développement réalisé dans le cadre de la thèse, de part sa taille et sa difficulté : la généricité des approches entraîne la gestion, à *l'exécution*, de types pour chaque donnée à traiter (action, horloges, etc...).

La version actuelle de l'outil est capable d'appliquer ces méthodes aussi bien en temps discret qu'en temps dense, pour produire respectivement un TIOTS et un automate temporisé de la partie service. Ces systèmes de transitions sont visualisables directement par l'interface graphique grâce à la bibliothèque JGrapht[8] (bibliothèque implémentant le type de données abstrait des graphes) et JGraph[9] (outil et bibliothèque de représentation graphique de graphe, mais ici, seule la bibliothèque de représentation nous intéresse). Il est également possible de visualiser les graphes ainsi obtenus grâce à l'outil Dot[10] de Graphviz permettant de les convertir au format PostScript par exemple.

[8] `http://jgrapht.sourceforge.net/`
[9] `http://www.jgraph.com/`
[10] `http://www.graphviz.org/`

Génération de l'automate temporisé de la partie cliente et détection d'ambiguïté

Une fois ce modèle de la partie service obtenu, il est alors possible d'exécuter les algorithmes (voir les algorithmes 4.4.1 et 6.4.1) de synthèse du client. Cela n'est possible qu'en utilisant également la relation d'interaction correspondante (temps discret ou temps dense).

Ensuite, si aucune ambiguïté n'est détectée, le système de transitions de la partie cliente est généré dans sa globalité. Ce système de transitions est alors en interaction avec celui de la partie serveur.

Enfin, cet automate est rendu disponible à l'outil ExecClient permettant d'invoquer le service, en suivant l'exécution dictée par le système de transitions de son client.

7.2.2 Fichier de règles (application de notre sémantique, généricité)

Le fichier de règles n'est autre que la traduction dans un « langage », créé pour ce développement, des règles des opérations définies dans la section 4.3 pour le temps discret et dans la section 6.3 pour le temps dense.

L'utilisation d'un fichier permettant de définir les différentes règles d'évolution d'un processus présente plusieurs intérêts :

- l'utilisation d'un fichier de règles, et non du code correspondant à ces règles, permet une modification rapide et aisée de ces règles, sans modifier une seule ligne de code du logiciel : ainsi, il est facile d'adapter une règle à un nouveau comportement d'un opérateur par exemple, sans même réécrire une ligne de code ;
- il est possible d'utiliser plusieurs fichiers de règles avec le même code, typiquement, comme nous l'avons vu dans le cas discret, le processus séquence a évolué de façon significative, nous pouvons donc encore utiliser l'ancienne sémantique sans alourdir notre code ; ou, plus simplement, l'utilisation d'un fichier de règles pour la sémantique en temps discret et la sémantique en temps dense !
- ces aspects visent à la généricité de l'approche. En effet, notre approche devait pouvoir être réutilisable dans un autre domaine sans tout réécrire ! Ce qui est le cas ici : seul ce fichier de règles est à réécrire.

Correspondance entre les algèbres de processus et la syntaxe des règles

Un fichier de règles est un fichier texte répondant à certains critères. La syntaxe est très simple (voir des exemples et une description complète de la syntaxe en annexe A) :

- il est possible d'ajouter des commentaires à tout moment, il suffit d'utiliser le caractère %, et tout ce qui est écrit à la suite, sur la même ligne, est pûrement et simplement ignoré ;
- les lignes vides sont ignorées ;
- une ligne déclarant une instruction commence pas un des caractères : T, O, G, R ou S.

La ligne T – temps dense ou temps discret ? Une ligne commençant par la lettre T (signifiant (T)ype) permet d'indiquer le type de sémantique utilisée : temps dense ou temps discret. En effet, le système de transitions généré est différent, donc il suffit de préciser LTS pour le temps discret et TA pour le temps dense.

Les lignes O –(O)pérations– définissent une nouvelle opération. Ainsi, la ligne :

```
O sequence [(p) (p)]
```

indique que le processus `sequence` prend deux paramètres, de types processus (voir en annexe pour plus de détails). Une ligne opération est accompagnée d'ensembles de lignes de types G et R, éventuellement suivie d'une ligne de type S.

Les lignes G –(G)ardes– définissent les gardes à vérifier. Si toutes les gardes sont vérifiées, la ligne de type R est interprétée.

Les lignes R –(R)ésultats– définissent l'action que le processus peut exécuter dans son état actuel, et également le processus cible qu'il va devenir après avoir exécuté cette action.

Les lignes S –(S)implications– définissent des simplifications. Par exemple, le processus sequence dont le premier paramètre est le processus empty est équivalent au second paramètre de sequence (puisque le processus empty se contente de ne rien faire dans notre sémantique). Donc pour éviter de créer des états supplémentaires (les états sont identifiés et mis en équivalence en termes de processus, voir l'implémentation du système de cache en section 7.2.3), il est plus judicieux de simplifier les écritures au maximum.

7.2.3 Génération du système de transitions de la partie service

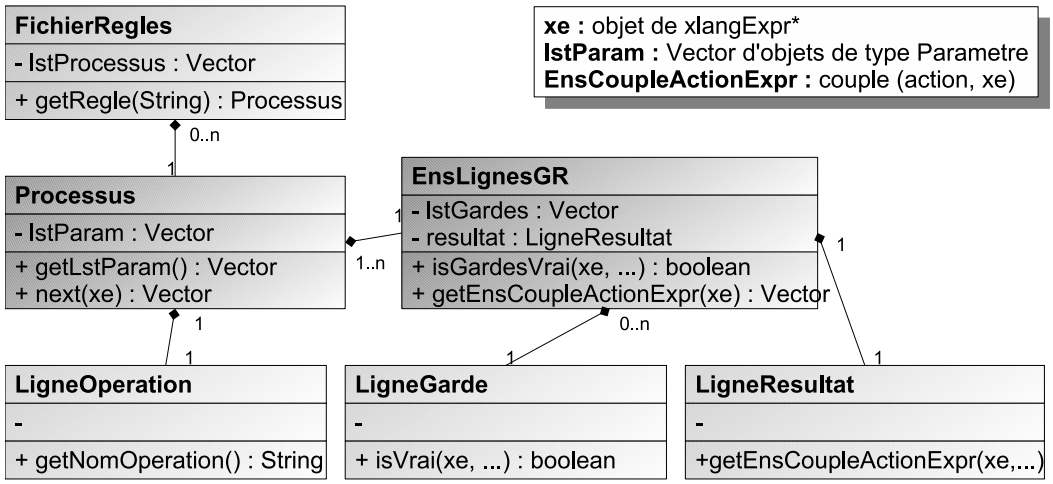


FIG. 7.3 – Règles et expressions.

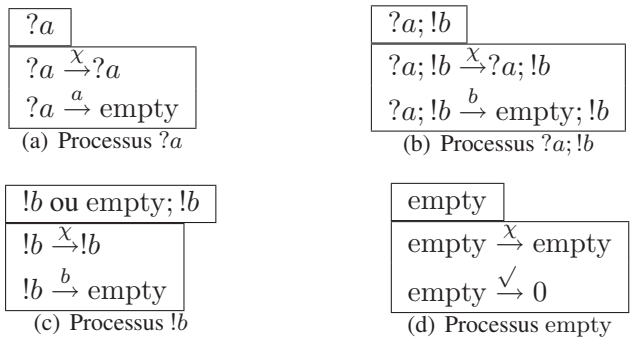
L'implémentation de l'analyse des règles et de génération du système de transitions de la partie service est une partie des plus difficiles (6500 lignes de code) et des plus conséquentes de notre implémentation (voir la figure 7.3 pour l'organisation des classes principales, dont nous n'aborderont pas tous les détails). Car non seulement les algorithmes doivent analyser le fichier texte, mais ils doivent également être capable d'analyser un processus et de donner les différentes possibilités d'actions qu'il peut réaliser, et ce récursivement. De plus, certains processus possèdent des paramètres de type liste (par exemple le processus flow) et dont l'évolution dépend d'un sous-ensemble de processus réalisant certaines actions.

Il est à noter que l'utilisation d'un analyseur syntaxique et grammaticale comme le couple *Flex* et *Yacc* pour le langage C ou pour le langage Java l'analyseur *JavaCC* n'apporterait rien de plus. En effet, analyser les règles doit être un procédé totalement générique et ne doit pas générer de code : l'outil WSMod ne doit pas être recompilé à chaque modification de la sémantique. De plus, la partie complexe de ce module ne réside pas au niveau de l'analyse du fichier dont la syntaxe est relativement simple, mais correspond à la génération des résultats.

La génération du système de transitions

La transformation en système de transitions se fait à partir des règles et de l'arbre syntaxique. Le noeud racine de l'arbre doit pouvoir fournir les différentes actions qu'il peut réaliser, ces actions seront ensuite traduites par des transitions dans le systèmes de transitions (TIOTS ou automate temporisé).

Pour déterminer l'ensemble des actions possibles, suivant le processus que représente un noeud de l'arbre syntaxique, il peut être nécessaire d'évaluer les actions possibles qu'un noeud fils peut réaliser, et ce récursivement.



(a) Processus $?a$ (b) Processus $?a; !b$ (c) Processus $!b$ (d) Processus empty

FIG. 7.4 – Exemple de génération d'ensembles d'actions pour le processus $?a; !b$, en temps discret.

Prenons, par exemple, dans le cas temps discret, le processus $?a; !b$, correspondant à recevoir le message a et à répondre par le message b. L'ensemble d'actions que peut réaliser le processus $\mathrm{sequence}$ dépend clairement du premier processus qu'il contient. Il est donc nécessaire d'analyser le processus $?a$, qui peut exécuter deux actions : écoulement de temps ou réellement recevoir le message a (voir figure 7.4(a)).

L'étape suivante consiste à vérifier la compatibilité des actions du processus $?a$ avec les différentes gardes du processus $\mathrm{sequence}$ et à déterminer l'ensemble d'actions que ce dernier peut réaliser (voir figure 7.4(b)). Ici, les deux ensembles sont identiques.

Le processus $\mathrm{empty}; !b$ a été créé lors des étapes précédentes et n'a pas encore été analysé. La règle de simplification permet de mettre en équivalence ce processus avec le processus $!b$, en effet, le processus empty se contente de laisser passer le temps (action χ) ou de terminer (action $\surd$). D'après la sémantique du processus $\mathrm{sequence}$, laisser passer le temps ne change pas l'expression, mais la possibilité d'exécuter l'action $\surd$ donne la main à l'exécution du deuxième processus, c'est-à-dire le processus $!b$. Ce qui revient à dire que le processus $\mathrm{empty}; !b$ est en tout point équivalent au processus $!b$, d'où la simplification. L'analyse du processus $\mathrm{empty}; !b$ se trouve à la figure 7.4(c).

Enfin, le processus empty est analysé (voir figure 7.4(d)).

Toutes ces étapes permettent de déterminer complètement le TIOTS du processus $?a; !b$, les différentes transitions et états sont indiqués dans la figure 7.4. L'état de départ est l'état $?a; !b$ et l'état de

terminaison est l'état 0. Bien évidement, les étapes décrites précédemment ne sont pas codées en dur pour un processus donné, mais dictées par le fichier de règles.

Il est donc nécessaire de vérifier, en continue, les types d'actions qu'un processus peut exécuter et leur compatibilité avec le processus de niveau supérieur le cas échéant, mais également d'évaluer les différents paramètres (de type simples –messages, exceptions– ou de type complexe –couples, listes–) d'un processus (comme par exemple la possibilité d'avoir un entier symbolisant la valeur de l'horloge dans le cas du processus scope), ou encore de créer de nouveaux processus suivant les actions réalisées.

Enfin, il est à noter que le code est commun entre la méthode temps discret et la méthode temps dense. Seul du code supplémentaire, activé en mode temps dense, est nécessaire pour l'ajout des gardes d'horloges et d'invariants comme expliqué dans les algorithmes présentés dans le chapitre 6 dans le cas temps dense.

Utilisation d'un système de cache

Pour être un minimum performantes, toutes les évaluations précédentes sont mises en cache. Ce système, si on le désire, peut être désactivé (attention à l'explosion mémoire dans ce cas !). Il existe deux niveaux de caches.

Les deux niveaux de caches. Un premier niveau concerne les processus : à chaque création d'un nouveau processus (et donc d'un arbre syntaxique), il est comparé avec l'ensemble des processus stockés en cache et si celui-ci existe déjà, il sera utilisé à la place du nouveau processus. Ceci évite par exemple de créer des nouveaux processus pour les actions faisant boucler sur l'état lui-même. Ce procédé de mise en cache de processus se fait également pour les sous-processus.

Le deuxième niveau de cache permet de stocker les différentes actions qu'un processus peut réaliser, associées au processus dérivé. Ainsi, dès que le processus a évalué les actions qu'il peut réaliser, celles-ci sont stockées dans ce processus. Ce mécanisme lié au mécanisme précédent de mise en cache des sous-processus permet par exemple de faire bénéficier à un processus de l'évaluation de sous-processus qui aurait été réalisée précédemment par un autre processus le contenant également (cas très fréquent avec le processus scope en temps discret : un nouveau processus est créé pour chaque transition de temps, ayant les mêmes ensembles d'évènements et d'exceptions à gérer), ces ensembles n'ont alors pas besoin d'être analysés à nouveau.

Statistiques liés au système de cache. Un compteur de performance permet d'évaluer le bénéfice des caches. En règle générale (cela peut être très variable dans des cas particuliers), en temps discret, le nombre d'utilisations du cache avec succès correspond au nombre d'éléments mis en cache (facteur de 0,8 à 1,2, avec un résultat légèrement meilleur pour le cache de processus que pour le cache d'actions). Pour le temps dense, le nombre d'utilisations du cache avec succès correspond à la moitié du nombre d'éléments qu'il contient à la fin de la génération pour le cache de processus, et pratiquement au nombre d'éléments mis en cache pour le cache d'actions. La différence s'explique par les nombreuses boucles sur l'action χ dans le cas temps discret qui n'existe plus dans le cas temps dense.

Implémentation des liens BPEL (processus flow)

Comme nous l'avons précisé dans les sections 4.3.4 et 6.3.4, la version actuelle de WSMod gère la mécanisme des *liens* (*links*).

Le mécanisme de liens. Le mécanisme de liens est un ajout au langage BPEL, par rapport à XLANG, permettant de gérer les synchronisations temporelles dans le processus flow(le but du processus flow est d'exécuter en parallèle l'ensemble de ses sous-processus). C'est-à-dire un moyen simple de permettre d'indiquer à une opération qu'elle doit attendre la fin d'une autre opération pour s'exécuter, ou qu'une condition, sur une variable interne au processus, soit vérifiée.

Un lien est représenté, techniquement, par un couple constitué d'une source (mot clé *source*) et d'une cible (mot clé *target*). Chaque opérateur BPEL peut se voir ajouté une source et une cible.

Chaque lien possède un nom, et il est possible de faire l'analogie de ce lien à une variable booléenne, initialisée à *faux* : l'affectation à vrai correspond à exécuter l'opérateur liée à une source, et l'exécution de l'opérateur liée à la cible n'est possible que si la variable booléenne est à *vrai*. L'utilisation, en grand nombre, de ce mécanisme casse fortement le parallélisme du processus flow.

De plus, chaque source peut être accompagnée d'une condition, qui sera évaluée au moment de la validation ou non de celle-ci. Par exemple, dans le cadre d'un échange d'argent, on peut imaginer une action en plus à réaliser dans le cas d'un montant supérieur à une certaine somme ; et donc, dans ce cas, valider un lien supplémentaire permettant d'exécuter une action gardée par une cible.

Le risque avec les liens est de rencontrer un chemin débouchant sur un blocage des processus (à cause d'un processus tombant en erreur par exemple). Cela est géré dans la spécification BPEL par le paramètre *suppressJoinFailure*. En activant ce paramètre à faux, cela désactive les actions gardées par une cible non encore validée en les remplaçant par le processus empty.

Pour un exemple d'utilisation des *links*, on pourra se rapporter à l'annexe B.

Processus non modélisés et les liens. Le processus invoke exprime l'invocation d'un service web externe au service web utilisant ce processus. Comme nous l'avons vu précédemment, les échanges de messages liés à son utilisation ne sont pas symbolisés dans l'interaction entre le service web principal et son client.

Or, les processus invoke peuvent activer également des sources utilisables dans d'autres processus intervenant, quand à eux, dans l'interaction entre le service web principal et son client. Donc ignorer simplement ces processus dans la construction des systèmes de transitions de la partie service peut entraîner des incohérences. La solution retenue et implémentée dans WSMod est de remplacer le processus invoke par un processus empty, ayant les sources et cibles définies dans le processus invoke.

L'implémentation des liens. Le code nécessaire à la gestion des liens est très similaire à celui de la remonté et l'ajout d'informations concernant les horloges sur le système de transitions. Il est cependant nécessaire de bien faire attention au moment de l'activation des sources et des cibles : les sources sont à vérifier dès le début d'exécution du processus associé, alors que les cibles ne sont à valider qu'à la fin de celui-ci.

Enfin, tout comme la gestion du cache, WSMod présente une option permettant de désactiver cette gestion des liens. Ceci ne produit pas un système de transitions faux pour autant, par contre, il comportera bien plus d'états que nécessaire et cela peut aboutir dans certains cas à des incohérences du système.

7.2.4 Génération du système de transitions client

Les différents algorithmes étudiés dans les chapitres précédent pour la génération du modèle client sont implémentés dans WSMod. Le modèle du client est généré seulement si l'algorithme de détection d'ambiguïté ne détecte aucun problème.

Ces algorithmes prennent en entrée le système de transitions de la partie serveur, et appliquent les différentes étapes des algorithmes sur ce dernier pour produire lui-même un système de transitions. Les données sont donc des sous-ensembles de l'ensemble d'états des systèmes de transitions, ainsi que les actions (ou transitions). Les algorithmes sont donc totalement indépendants des technologies nécessaire aux services web et représentent environ 1000 lignes de code.

Vers le module d'exécution client

Une fois le modèle client obtenu, il est alors possible de construire du code *proxy* permettant de lier les différentes opérations décrites dans le fichier WSDL aux actions du système de transitions. Cette partie est à l'heure actuelle encore en phase de développement.

Ce *proxy* a pour but de fournir un ensemble de classes permettant d'invoquer toutes les opérations présentes dans le système de transitions. En effet, ces invocations se font en utilisant des protocoles de communications tel SOAP, ou encore des transformations d'instructions WSDL en code Java. Pour permettre une exécution aisée du système de transitions, du code constituant ce *proxy* est généré, permettant alors lors de l'exécution guidée par l'automate de ne se concentrer que sur cette exécution et non pas toutes les couches basses liées à l'utilisation des technologies services web.

Par contre, cette étape est à réaliser avec le plus grand soin : en effet, le code produit ici sera destiné à être exécuté par un autre programme (sur une autre machine ou non). Ceci pose un problème notable de sécurité : il faut donc que le code généré présente le moins de risque possible concernant des attaques potentielles.

7.2.5 Utilisation de l'outil

WSMod peut être utilisé selon trois modes ou versions différents : en mode graphique, en mode ligne de commande, et en mode réseau.

Version graphique

La version graphique est une interface développée en Java Swing. Il suffit alors de charger les fichiers de descriptions WSDL et XLANG ou BPEL depuis cette interface puis de lancer la génération des modèles. Une boîte de dialogue indique la présence d'ambiguïté le cas échéant, et si tout se déroule correctement, les modèles serveurs et clients peuvent être visualisés par l'interface à l'aide de la bibliothèque JGrapht.

Cette version est totalement opérationnelle et permet de configurer facilement les différentes options accessibles (gestion des liens, conserver les processus invoke dans la modélisation, niveau de log, etc...). Lors de la génération des différents modèles, les statistiques concernant l'utilisation du système de cache y sont également affichés.

Le code de l'interface graphique est dissocié du code de la modélisation, pour un total d'environ 1000 lignes de code. Lors de la compilation, une archive au format JAR est généré.

Version ligne de commande

La version ligne de commande permet d'avoir des informations supplémentaires par des niveaux de *logs* configurable à l'exécution ou à la compilation. Dans cette version, les graphes sont visualisables par l'outil Dot : les versions PostScript sont automatiquement générées et le fichier source au format dot conservé, permettant alors à l'utilisateur de l'exploiter.

Les différents fichiers de règles (temps dense et temps discret), les fichiers de descriptions (interface et comportementale) sont à indiquer sur la ligne de commande.

Comme la version graphique, cette version est totalement opérationnelle, et lors de la compilation, une archive au format JAR est généré. Le code est commun à la version précédente, sauf pour l'interface utilisateur.

Version réseau

Enfin, la dernière version, encore en développement, permet l'utilisation de WSMod comme un démon et l'interaction se fait par les méthodes RMI. Cela est très utilisé pour le module d'exécution du client présenté dans la section suivante.

7.3 L'outil ExecClient

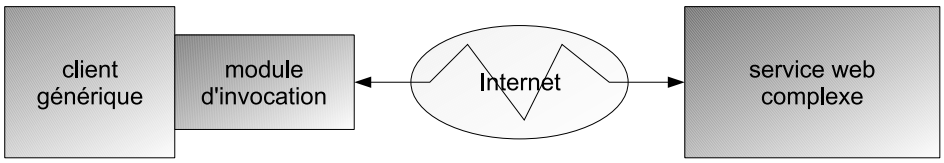


FIG. 7.5 – Architecture de l'outil ExecClient.

ExecClient est l'outil permettant d'utiliser le système de transitions généré par WSMod ainsi que les classes *proxy*, pour ensuite suivre le chemin dicté par les franchissements des différentes transitions, correspondant à l'exécution du client en interaction avec le service (voir figure 7.5). Son développement, à nouveau en Java et assuré par moi-même, est arrivé plus tard au cours de cette thèse et n'est pas encore terminé.

Ce module repose sur la gestion des évènements (arrivée de messages venant du service, intervention de l'utilisateur) et des horloges (gardes liées au processus métier, avec possibilités d'imbrications sur plusieurs niveaux), qui doivent être traitées, de façon non triviale, en suivant les indications de l'automate temporisé du client.

7.3.1 Interaction avec WSMod

L'appel de WSMod, pour être le plus automatique possible, se fait dès la saisie par l'utilisateur des informations sur le service web (fichiers de description). Ensuite, l'appel est réalisé de façon automatique par l'interface Java RMI mise à disposition par l'outil WSMod. Ainsi, ExecClient récupère le modèle du client et les classes *proxy* d'invocation, permettant de communiquer avec le service.

7.3.2 Exécution guidée par l'automate du client

Le principe de base de l'exécution du client repose sur le suivi des transitions empruntées dans le système de transitions et la réalisation des actions étiquetées sur celles-ci.

Il est alors nécessaire de gérer les types de transition que l'on peut rencontrer dans le client :
- l'écoulement de temps (χ), seulement dans le cas du temps discret, dans le cas du temps dense, il faut gérer des horloges ;
- l'envoi d'un message (!) ;
- la réception d'un message (?).

Les autres types de transitions ont été agrégées par l'algorithme de synthèse du client et la transition de terminaison ($\surd$) indique la fin de l'exécution du service et donc de l'interaction.

Écoulement du temps

L'écoulement de temps peut se traduire de deux façons différentes suivant son importance.

Si l'état ne possède aucun invariant et aucune transition gardée, alors, il ne faut pas en tenir compte : le temps peut évoluer indéfiniment, il n'influencera pas le changement d'état.

Par contre, si l'état actuel possède une transition sortante gardée ou présentant un *timeout*, alors il est important de gérer une horloge qui aura été correctement initialisée sur une des transitions franchies en amont et vérifier que les bornes de l'horloges correspondent aux bornes des gardes. Si ce n'est plus le cas, il faut adapter l'ensemble des informations demandées à l'utilisateur, par exemple en arrêtant la possibilité, à l'utilisateur, de remplir un formulaire, et afficher, le cas échéant, un message venant du service indiquant ce *timeout*. Il est donc indispensable de garder une liste des formulaires que l'utilisateur peut remplir, chaque élément de cette liste étant à mettre en correspondance avec l'action et la transition dans le modèle du client.

Envoi d'un message

Si une transition dont l'étiquette commençant par '!' est rencontrée, l'exécution reste en attente sur cet état. En effet, c'est à l'utilisateur de décider d'un envoi et donc du franchissement de cette transition.

Dans ce cas, une fenêtre est affichée pour permettre de remplir les différents champs du message à envoyer. Le moment où l'utilisateur décide de l'envoi du message est celui du franchissement de la transition correspondante.

Remarque : comme, pour un écoulement de temps, l'état courant peut changer pendant que l'utilisateur remplit les différents champs (gestion des *timeout*, évènements, etc...), il faut alors vérifier que sur chaque état, la transition sur le message à envoyer est toujours présente. Si ce n'est pas le cas, la fenêtre est simplement détruite et l'utilisateur ne pourra alors plus invoquer/franchir cette transition, puisque le service ne permet plus de l'invoquer.

Réception d'un message

À réception d'un message (transition dont l'étiquette commence par ' ?'), ExecClient affiche simplement le résultat et franchit la transition correspondante.

Pour permettre de visualiser la réponse assez longtemps, la fenêtre de réponse n'est détruite qu'au moment de la réception d'un nouveau message.

L'exécution continue ensuite, à travers le système de transitions. Et ce, tant que l'état courant possède au moins une transition sortante, le cas contraire correspond à l'état de terminaison.

7.3.3 Invocation des services

La partie invocation de services (voir figure 7.5) consiste à traiter les informations stockées sur l'étiquette de la transition, concernant le message, l'opération et l'URL du service. Pour réaliser cette invocation, l'outil ExecClient utilise les bibliothèques clientes fournies par Axis. Ces bibliothèques ont l'avantage d'être indépendantes du langage de description comportementale et ainsi nous laisser la maîtrise complète de l'exécution du client.

Il est suffisant d'avoir les informations contenues sur la transition du modèle à exécuter, correspondant, en fait, à un extrait de la description WSDL du service. Cet extrait contient le minimum d'information permettant l'invocation par l'API Axis d'une méthode du service web.

7.4 Synthèse

Ce chapitre montre l'implémentation et la mise en œuvre des algorithmes présentés dans les chapitres précédents. Ce développement personnel, réalisé dans le cadre de cette thèse, avait pour but d'obtenir un ensemble d'outils permettant la génération d'un modèle du service et d'un modèle du client (outil WSMod), puis ensuite de l'invocation du service à l'aide de l'exécution guidée par le modèle du client (outil ExecClient).

Il met également en évidence le fait que l'implémentation devait présenter deux facettes :

- premièrement, elle devait être le plus indépendante possible des technologies utilisées, et donc avoir le moins de code spécifique possible à ces technologies. Cela a été réalisé en utilisant des intermédiaires, comme la construction de l'arbre syntaxique lors de la modélisation du comportement du service ;
- deuxièmement, les règles issues de la formalisation d'un langage de description comportementale ne devaient pas être codé explicitement. Un langage a été créé pour l'occasion, entraînant la gestion complexe, à l'exécution, des types et des ensembles d'actions qu'un processus peut réaliser ; ces actions dépendant également des sous-processus.

L'outil WSMod est totalement opérationnel et permet de modéliser à la fois en temps discret et en temps dense des services web possédant une description comportementale dans les langages XLANG et BPEL. Une gestion d'un cache permet d'avoir des performances acceptables pour une interaction homme/machine. Les résultats peuvent être visuels grâce à l'utilisation de la bibliothèque *jgrapht* ou sous forme de fichier *dot*. C'est l'outil le plus conséquent développé dans le cadre de cette thèse, développement qui a du être adapté suivant l'évolution des algorithmes issus de notre recherche durant leurs établissements et également aux technologies très mouvantes des services web.

L'outil ExecClient est en stade de développement et de test. Le manque de service web disponible gratuitement et répondant à nos critères de sélection est un frein à ce développement. Cet outil met en évidence la difficulté de traiter les informations et les évènements venant à la fois du modèle client, du service en lui-même, et de l'utilisateur devant remplir les informations à envoyer au service.

Conclusions et Perspectives

Conclusions et Perspectives

Synthèse des travaux et résultats

Ce travail de thèse apporte une solution pour la modélisation formelle de service web, prenant en compte les aspects temporels. En effet, beaucoup d'approches concernant la formalisation et la modélisation des services web existent dans la littérature, mais très peu prennent en compte l'aspect temporel des différentes actions que peut réaliser un service web, pour se concentrer uniquement sur les échanges de messages. Ici, nous apportons deux solutions : une à base de formalisme temps discret et l'autre à base de formalisme temps dense. Le formalisme temps discret consiste à discrétiser le temps, c'est-à-dire considérer un écoulement de temps comme une suite de *tick* d'horloge correspondant eux-mêmes à une unité de temps. Par contre, le formalisme temps dense consiste à supposer l'écoulement du temps de façon continu en représentant, par exemple, une durée par un réel.

Sémantique en temps discret Dans le cadre de notre formalisme en temps discret, nous apportons une sémantique opérationnelle pour les langages de description comportementale de service web. Nos exemples d'applications sont XLANG et BPEL. Cependant, grâce à une approche générique, identifiant les éléments clés de ce type de langages, nous pouvons adaptater nos résultats à d'autres langages du même type. Par le biais de cette sémantique opérationnelle, nous obtenons un modèle TIOTS de la partie service. Ce TIOTS représente les états et l'évolution du service, c'est à dire les différentes actions que ce service peut réaliser.

Nous définissons également une *relation d'interaction* entre le TIOTS d'un client potentiel et celui du service qu'il invoque. Cette relation, proche d'une bisimulation classique entre TIOTS, en diffère au niveau des messages échangés. Elle doit en effet rendre compte du nécessaire déterminisme d'un programme client. La relation d'interaction nous permet de vérifier le comportement du service web avec un client adapté. Les propriétés vérifiées portent :

- sur l'exécution déterministe de la suite d'échanges de messages ;
- mais également sur la possibilité de la gestion du temps pour le client et le service ;
- et enfin, sur la détection de la terminaison par le client et le service.

Si l'ensemble des propriétés n'est pas vérifié, la relation d'interaction déclare le service web comme ambigu. Cette détection d'ambiguïté est indispensable et fortement liée au domaine des services web : l'interaction, entre un client et un service web déclaré ambigu, a de forte chance de terminer sur un inter-blocage ou une incompréhension de la part d'une des deux entités.

En parallèle à la vérification des propriétés issues de la relation d'interaction, nous avons développé un algorithme de génération du modèle client sous la forme d'un TIOTS. Ce modèle représente le comportement, en termes d'échanges de messages, que doit avoir un client pour interagir sans problème avec le service.

Nous obtenons ainsi, pour un service web possédant une description comportementale et non

détecté comme ambigu, la modélisation de son comportement par un TIOTS, et le modèle d'un client interagissant de façon adaptée avec ce service.

Vérification d'une chorégraphie Nous avons ensuite étendu nos recherche à la vérification d'une chorégraphie, et plus exactement de l'interaction interne entre les sous-services web, appelés partenaires, de la chorégraphie. Cette vérification est plus complexe que dans le cadre d'une orchestration. En effet, dans le cas d'une orchestration, le service complexe invoque des sous-services web sur lesquels il n'a aucun contrôle ; il devient donc client de ces services, qui peuvent même ne pas posséder de description comportementale. Par contre, dans une chorégraphie, chaque service connaît l'existence des autres services (ses partenaires). La vérification d'une chorégraphie repose donc sur la vérification de l'interaction de chaque partenaire avec les autres partenaires.

La modélisation des différents partenaires d'une chorégraphie, et la vérification de l'interaction interne à celle-ci exige une extension de notre sémantique en temps discret, aux opérations non visibles dans l'interaction entre le service et client. En effet, ces opérations sont liées aux interactions entre un service et ses sous-services, assimilés à des partenaires dans le cadre d'une chorégraphie. Elles doivent donc être modélisées dans ce cadre. Nous avons développé cette extension, également dans une approche générique : la sémantique proposée n'est pas liée à un langage de description comportementale donné. Nous avons appliqué notre approche aux chorégraphies dont les partenaires sont décrits en BPEL.

L'application des algorithmes provenant de la modélisation service/client, permet d'obtenir les modèles des partenaires. Cependant, pour la vérification, nous avons adapté notre relation d'interaction et la détection d'ambiguïté à deux partenaires. En effet, ici, le comportement des partenaires est connu, car décrit dans le cadre de la chorégraphie : il n'est pas nécessaire de générer un modèle du client, les partenaires sont à la fois service et client d'un autre partenaire. Il est nécessaire ensuite de vérifier les deux modèles dans leur ensemble, et non plus générer un modèle répondant à certains critères par rapport à un premier modèle.

Dans l'optique de vérifier la possible ambiguïté dans l'interaction interne à une chorégraphie, par rapport à notre relation d'interaction, nous étendons notre sémantique opérationnelle avec un opérateur d'agrégation. Cet opérateur décrit le comportement observable de plusieurs partenaires vus par un partenaire interagissant avec eux. Ainsi, en vérifiant la relation d'interaction sur l'ensemble des interactions entre chaque partenaire et ses partenaires agrégés, nous pouvons vérifier la possible ambiguïté de l'interaction des partenaires de cette chorégraphie, et déclarer si la chorégraphie présente une ambiguïté ou non. Il est possible d'effectuer cette vérification de façon parallèle : chaque partenaire vérifie son interaction avec la vue agrégée de l'ensemble des autres partenaires.

Sémantique temps dense La modélisation précédente en temps discret présente une possibilité d'explosion du nombre d'états du modèle représentant le service et le client. Cette explosion est intrinsèque au modèle utilisé et au système modélisé : des gardes de temps sur l'exécution peuvent être imbriquées, avec des échelles de temps différentes.

Dans l'approche temps dense, nous avons adapté notre sémantique opérationnelle issue du temps discret. Pour cela, nous avons ajouté une nouvelle action symbolisant le *timeout*. Nous avons développé un algorithme de transformation en automate temporisé qui exige une détection des horloges actives ainsi que la hiérarchisation de gardes d'horloges en cas de concurrence entre ces gardes et horloges au sein d'un même processus. Nous obtenons alors un automate temporisé modélisant l'évolution du comportement d'un service, par les différentes actions qu'il peut faire, incluant la représentation des aspects temporels par l'utilisation de gardes d'horloges.

Par la même approche qu'en temps discret, nous avons adapté notre relation d'interaction et notre détection d'ambiguïté pour prendre en compte les informations issues des gardes d'horloges. Concernant cette modélisation, certains processus sont déclarés ambigus alors qu'ils peuvent ne pas l'être. En effet, l'action d'initialisation d'une horloge sur un service, n'est pas détectable par un client, et inversement. Ceci entraîne donc le rejet de certains services, qui ont peut-être un comportement correct et non ambigu. Par contre, cette nouvelle modélisation est efficace en cas de nombreuses imbrications d'opérateurs temporels : le modèle obtenu du service web en version temps dense est optimal par rapport à la version temps discret, dont le nombre d'états, dans ce cas, a de fortes chances d'exploser.

Enfin, notre algorithme de vérification de la relation d'interaction et de l'ambiguïté du service, construit en même temps le modèle automate temporisé du comportement d'un client (si ce service est non ambigu). Ce modèle du client est en correspondance avec modèle du service par la relation d'interaction et par les horloges.

Implémentation et mise en œuvre Nous avons implémenté les différents algorithmes que nous avons développés dans un ensemble d'outils génériques. Les deux principaux outils sont WSMod et ExecClient. Leur utilisation est multiple : interface en ligne de commande et interface graphique, avec possibilité de sauvegarder les modèles générés au format *dot* par exemple, ou de les visualiser à l'exécution en utilisant la bibliothèque *jgrapht*.

L'outil WSMod est un outil de modélisation du comportement de la partie service et de la partie client, aussi bien en temps dense qu'en temps discret. Il implémente les algorithmes de détection d'ambiguïté et de vérification de la relation d'interaction. En cas d'absence d'ambiguïté sur le service, il génère le modèle client adapté au service.

La conception de ces implémentations met en œuvre une approche générique en ce qui concerne le langage de description utilisé. Ce choix de conception permet l'adaptation rapide des éléments clés en écrivant pas ou peu de code. Ainsi, la sémantique opérationnelle est fournie au programme par un fichier texte, permettant d'utiliser plusieurs sémantiques, sans même recompiler le programme. Cette approche s'avère très efficace lors de tests ou d'adaptations même majeures. Nous avons en particulier tiré partie de cette généricité lors du passage du temps discret au temps dense : seuls les algorithmes de gestions de gardes d'horloges ont dû être implémentés. De même, le code principal de modélisation est indépendant du langage de description comportementale, ainsi, lors du passage de XLANG à BPEL, seul le code construisant une représentation syntaxique de la description a du être réécrit.

Enfin, l'outil ExecClient, écrit en Java, permet de réaliser une exécution guidée d'un client, invoquant le service. L'exécution guidée se fait grâce aux modèles obtenus par l'outil WSMod, c'est-à-dire en suivant l'interaction dictée par le TIOTS (dans le formalisme temps discret) ou l'automate temporisé (dans le formalisme temps dense). Le but principal de ExecClient et WSmod est de fournir un ensemble d'outils permettant de générer et d'exécuter un client pour un service quelconque dont la seule connaissance serait le fichier de description comportementale récupéré chez le fournisseur de ce service. Plus globalement, ce développement s'inscrit dans le cadre d'une plateforme de services web développée au LAMSADE.

Perspectives à court terme

Tout au long de ce travail de thèse, des perspectives se sont dégagées ou des questions sont encore à ce jour, ouvertes et sans réponse. Nous présentons ci-dessous ces différents points en donnant certains

éléments de réponses. Nous présentons également les projets en cours de développement auxquels nous participons.

Temps dense et chorégraphie Nous allons étendre notre vérification d'un ensemble de services web évoluant dans le cadre d'une chorégraphie à la sémantique temps dense. Ce passage du temps discret au temps dense permettra d'éviter l'explosion du nombre d'états des modèles. En utilisant ce formalisme temps dense, nous devrons également étudier le problème des fausses détections d'ambiguïté, comme dans le cas de la génération du client en temps dense.

Une des premières pistes à explorer est d'employer une méthode analogue à celle développée lors du passage temps discret vers temps dense pour la modélisation du service et de la relation d'interaction. Il faut alors transposer le nouvel opérateur d'agrégation dans le formalisme temps dense, en proposant un algorithme de fusion des gardes d'horloges des différents modèles agrégés (éviter le chevauchement, etc...). Il faut ensuite découpler la relation d'interaction (approche temps discret) de la génération du modèle client et prendre en compte la vérification de la compatibilité des ensembles d'horloges pour les états en relation.

Plateforme de « services web » Dans le cadre de la plateforme de « services web », développée au LAMSADE, nous allons intégrer nos modules développés dans le cadre de cette thèse. Les outils WSMod et ExecClient peuvent ainsi être utilisés dans deux des principaux projets de cette plateforme.

L'équipe, composée de Céline Boutroos-Saab, Demba Coulibaly et Serge Haddad, travaille sur un méta-langage CWSL (*Complex Web Services Language*) proposant à l'utilisateur d'écrire facilement un service web *complexe*. Ce langage est très similaire à JAVA et permet de s'affranchir complètement des éléments spécifiques de la syntaxe liés aux services web : le code de création des objets et d'invocations des services web est généré automatiquement. Le langage permet donc d'utiliser directement les méthodes décrites dans le fichier WSDL, permettant ainsi, au développeur, de ne se préoccuper que de la composition des différents services et non plus de l'invocation des sous-services. L'outil propose de produire à la fois un code exécutable représentant le service web *complexe*, mais aussi les descriptions WSDL et BPEL. L'utilisation des outils WSMod et ExecClient, pour tester ce service web BPEL, ne nécessitera pas l'écriture d'un client spécifique.

Une autre équipe, composée de Serge Haddad, Mehdi Ben Hmida et Valérie Monfort [BM04, HHM06], propose de modifier le comportement des services web en utilisant la programmation par aspects. Ainsi, sans réécrire le service web, il sera possible de rajouter des fonctionnalités à ce service, comme par exemple une phase d'identification. Encore une fois, disposer d'un client générique tel que WSMod et ExecClient permet de tester ces services sans développer un client spécifique.

Transposition des résultats De part notre approche générique, il est possible d'adapter notre sémantique de formalisation à d'autres systèmes que les services web, ou encore à un autre langage de description de services web.

Pour l'adapter à un autre système, objectif à plus long terme, plusieurs éléments sont à prendre en compte : si les mêmes opérateurs sont présents ou identifiables, alors seule la partie analyse en entrée est à modifier, cela correspond à des aspects technologiques. Par contre, si de nouveaux opérateurs sont présents, alors, il faut étendre l'ensemble des opérateurs par un ensemble de règles algébriques définissant ces nouveaux opérateurs.

Pour l'adaptation à un autre langage, à plus court terme, nous pouvons appliquer la même méthode que lors de notre migration de XLANG à BPEL : identifier les éléments clés du nouveau langage et les mettre en relation avec le précédent. On pourrait imaginer adapter notre sémantique au langage

BPMN (*Business Process Modeling Notation*), développé par le groupe BPMI (*Business Process Management Initiative*) et l'OMG (*Object Management Group*). Les principaux opérateurs tels que la *séquence*, le *parallélisme* ou encore le *choix* y sont présents.

Problèmes ouverts

En plus des projets à court terme présentés ci-dessus, nous envisageons un ensemble de prolongements et d'extensions de nos travaux. Ces nouvelles approches présentent des points similaires : identification des éléments clés à modéliser, utilisation de l'aspect composition des services web, ou encore utilisation la description comportementale et sémantique des services web pour obtenir de l'information clé sur ceux-ci.

Temps dense et ambiguïté Comme nous l'avons souvent souligné, notre approche de formalisation des services web et nos algorithmes de détection d'ambiguïté partent sur le fait que les délais de communications sont nuls (ou égaux quelque soit la communication). En effet, nous ne prenons pas en compte ces délais de communications. Il serait donc indispensable d'étendre notre formalisation et nos algorithmes pour prendre en compte ces délais, et ainsi être fidèle au modèle asynchrone dont font partis les services web, et ne plus simplement s'en approcher.

Un autre aspect à développer peut également venir de la détection d'une fausse ambiguïté temporelle en temps dense pour certains cas de services web. Il est donc souhaitable de développer des recherches permettant de proposer une solution à ce problème, en travaillant, par exemple, sur les aspects liés aux horloges, ou encore liés au déterminisme du système.

Dynamicité Les services web sont accessibles à travers un réseau, généralement Internet. Ce réseau est souvent dynamique. Ainsi, fournir des approches permettant de répondre à cette dynamicité, en s'y adaptant, ouvre de grandes perspectives.

Par exemple, la vérification de l'interaction au sein d'une chorégraphie peut permettre une composition ou une adaptation automatique de la chorégraphie en cas de panne d'un des partenaires : plutôt que de déclarer la chorégraphie non fonctionnelle, les partenaires, détectant la panne, contacteraient un annuaire de services pour obtenir un remplaçant à ce service défectueux. Grâce aux algorithmes de vérifications présentés, le partenaire peut alors être certain qu'il pourra interagir correctement avec ce nouveau service.

L'utilisation des services web dans le monde de l'embarqué nécessite cette gestion de la dynamicité d'adaptation [SNMRRP06, BCPR04]. En effet, les ressources en termes de mémoire, énergie et bande passante sont très limitées dans ces environnements. En cas de changement de version d'un service auquel le terminal mobile accède, même mineure, il est alors impensable de faire une mise à jour de l'application cliente : cela nécessiterait trop de ressources. Plusieurs approches sont alors possibles :

- en cas d'injection de nouveaux services, le terminal mobile doit avoir un moyen de découvrir ces nouveautés. En plus de l'application des algorithmes de vérifications, l'aide des apports du web sémantique et des ontologies, permettant de décrire sémantiquement les informations, est très utile pour adapter son comportement par rapport au nouveau service ;
- en cas de modifications des services web, le terminal mobile doit s'en apercevoir, et cela, grâce à un système d'alerte par exemple ;
- enfin, les clients mobiles se connectent et se déconnectent fréquemment. Les services doivent donc gérer ces cycles de connexions et prévoir des reprises sur erreur par exemple.

Pour permettre la mise à jour du client, il est indispensable, en plus du module d'exécution guidée par le modèle client, que le terminal mobile possède un module permettant de mettre à jour ce modèle. Le terminal mobile doit alors être capable de récupérer le nouveau modèle client, suite à une mise à jour du service. La première application exécutera alors point à point les détails de la nouvelle description.

Transposition et extension de notre approche Les grilles de calculs, et notamment les projets GRID, s'orientent vers des systèmes de gestion proches sur plusieurs aspects des architectures orientées service et sur l'utilisation des approches du Web sémantique (ontologies). Nous relevons cette tendance par exemple dans le projet SOKU – *Service Oriented Knowledge Utilities* [JR06]. Nous lisons :

- l'utilisation de l'approche *orienté service* permet d'avoir une architecture dont les services peuvent être instanciés et composés dynamiquement ;
- des descriptions sémantiques sont indispensables pour permettre des compositions ou adaptations automatiques en cas de nouveaux services, de mises à jours ou encore de pannes ;

Ces évolutions probables des systèmes de grille ouvre des perspectives d'extension de nos travaux à ces systèmes pour ce qui concerne l'orientation service : dans quelle mesure, une modélisation des services et client de grilles peut s'appuyer sur nos résultats ?

Par ailleurs, les approches qui permettent un contrôle des applications à structure dynamique devront de plus en plus recourir à des méthodes de type Web sémantique (ontologies et langages associés). Des adaptations de ces approches au contexte des services web et de leur vérification ont déjà été engagées, comme par exemple le langage OWL-WS [BCMS06]. La question des liaisons entre ce niveau et celui de nos travaux est donc une voie à explorer.

Liaison avec les approches MDE/MDA Les approches (MDE – *Model Driven Engineering*) et MDA (*Model Driven Architecture*) sont actuellement un des moyens utilisés en génie logiciel pour élever le niveau conceptuel de développement des applications informatiques.

Dans le prolongement du module de composition de services web de la plateforme développée au LAMSADE, il serait intéressant d'étudier dans quelle mesure, notre approche peut s'intégrer aux approches MDE/MDA. Ainsi, plutôt que d'écrire directement du code JAVA indépendant des services web, il serait possible d'utiliser des techniques de compositions automatiques à base de modèles pour représenter la composition de services. Le développeur utilisant ce type d'outil pourrait alors se concentrer, dans un premier temps seulement, sur l'architecture avant même de prendre en compte l'écriture du code de la composition. Dans [Bar06] par exemple, il est possible de définir des patrons de compositions de services web. Ces patrons aident au développement et au déploiement d'une architecture entièrement basée sur les services web.

À un niveau supérieur, ces différents modèles peuvent eux-mêmes être décrits dans un méta-modèle. Par exemple, [SWK06] établit un méta-modèle pour le langage de description de site web WebML. Les résultats attendus de cette méta-modélisation sont l'indépendance vis à vis du code et la possibilité de définir des transformations entre différents modèles d'un système. Si de telles abstractions s'avèrent pertinentes dans le contexte des services web, l'adaptation de nos travaux à de tels méta-modèles pourrait compléter la démarche architecturale dans le domaine de la vérification.

Bibliographie

[AA02] Intalio Assaf Arkin. Business Process Modeling Language, 13 Novembre 2002. http ://www.bpmi.org/bpml-spec.esp.

[AAA+06] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling Commerce, Canyang Kevin Liu, SAP, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web Services Business Process Execution Language Version 2.0 - Committee Draft, 17 may 2006.

[AAF+02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal Takacsi-Nagy, Ivana Trickovic, and Sinisa Zimek. Web Service Choreography Interface 1.0, 2002.

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services (BPEL4WS), Version 1.1, may 2003.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[AFH99] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata : a determinizable class of timed automata. *Theoretical Computer Science*, 211(1–2) :253–273, 1999.

[Alu98] Rajeev Alur. Timed automata. *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998.

[AP04] Rajeev Alur and Madhusudan Parthasarathy. Decision problems for timed automata : A survey. In *Lecture Notes in Computer Science, Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems : Real Time*, volume 3185, pages 1–24, 2004.

[Bar06] Ronan Barrett. Model driven design of distribution patterns for web service compositions. In *In Proceedings of Model-Driven Web Engineering (MDWE 2006)*, Standford Linear Accelerator Center, July 11, 2006.

[BBC+05] Ruslan Bilorusets, Don Box, Luis Felipe Cabrera, Doug Davis, Donald Ferguson, Christopher Ferris, Tom Freund, Mary Ann Hondo, John Ibbotson, Lei Jin, Chris Kaler, David Langworthy, Amelia Lewis, Rodney Limprecht, Steve Lucco, Don Mullen, Anthony Nadalin, Mark Nottingham, David Orchard, Jamie Roots, Shivajee Samdarshi, John Shewchuk, and Tony Storey. Web Services Reliable Messaging, Février 2005.

[BCC+04] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugène Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. Web Services Addressing, August 2004.

[BCH05] Dirk Beyer, Arindam Chakrabarti, and Thomas A. Henzinger. Web service interfaces. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005, Chiba, May 10-14)*, pages 148–159. ACM Press, 2005.

[BCMS06] Stefano Beco, Barbara Cantalupo, Nikolaos Matskanis, and Mike Surridge. Putting semantics in grid workflow management : the OWL-WS approach, 2006.

[BCPR04] M. Brambilla, S. Ceri, M. Passamani, and A. Riccio. Managing asynchronous web services interactions. In *IEEE International Conference on Web Services, ICWS 2004*, San Diego, CA, USA, 2004.

[BGM01] Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the IF environment, 2001.

[BLFIM98] T. Berners-Lee, R. Fielding, U.C. Irvine, and L. Masinter. Uniform Resource Identifiers (URI) : Generic Syntax, August 1998. http ://www.ietf.org/rfc/rfc2396.txt ?number=2396.

[BLL+98] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of uppaal, 1998.

[BM04] Fabien Baligand and Valérie Monfort. A concrete solution for web services adaptability using policies and aspects. In *ICSOC '04 : Proceedings of the 2nd international conference on Service oriented computing*, pages 134–142, New York, NY, USA, 2004. ACM Press.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[CCF+05a] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Jim Johnson, Sean Joyce, Chris Kaler, Johannes Klein, David Langworthy, Mark Little, Anthony Nadalin, Eric Newcomer, David Orchard, Ian Robinson, John Shewchuk, and Tony Storey. Web Services Coordination, August 2005.

[CCF+05b] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Jim Johnson, Sean Joyce, Chris Kaler, Johannes Klein, David Langworthy, Mark Little, Anthony Nadalin, Eric Newcomer, David Orchard, Ian Robinson, Tony Storey, and Satish Thatte. Web Services Atomic Transaction , August 2005.

[CCF+05c] Luis Felipe Cabrera, George Copeland, Max Feingold, Robert W Freund, Tom Freund, Sean Joyce, Johannes Klein, David Langworthy, Mark Little, Frank Leymann, Eric Newcomer, David Orchard, Ian Robinson, Tony Storey, and Satish Thatte. Web Services Business Activity Framework, August 2005.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1, march 2001. http ://www.w3.org/TR/wsdl.

[CD99] James Clark and Steve DeRose. XML Path Language (XPath) – Version 1.0, 16 Novembre 1999. http ://www.w3.org/TR/1999/REC-xpath-19991116.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications : a practical approach. In *POPL '83 : Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[CFN+02] Michael Champion, Chris Ferris, Eric Newcomer, Iona, and David Orchard. Web Services Architecture, 14 Novembre 2002. http ://www.w3.org/TR/2002/WD-ws-arch-20021114/.

[CGK+02] Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business Process Execution Language for Web Services (BPEL4WS), Version 1.0, 2002. http ://www-106.ibm.com/developerworks/library/ws-bpel/.

[Cha02] Jean-Marie Chauvet. *Services Web avec SOAP, WSDL, UDDI, ebXML...* Eyrolles, Paris, 2002.

[DF02] Traduction d'Alexandre Gachet David Flanagan. *Java In a Nutshell, Manuel de Référence, 4ème édition*. O'Reilly, Paris, 2002.

[Dia01] Michel Diaz. *Les réseaux de Petri - Modèles fondamentaux*. Hermes, Paris, 2001.

[DKK05] Raymond Devillers, Hanna Klaudel, and Maciej Koutny. A petri net semantics of a simple process algebra for mobility. In *Proceedings of the 12th International Workshop on Expressiveness in Concurrency, EXPRESS'05, satellite workshop of the CONCUR'2005*, pages 69 – 80. Technical Report CS-05-20, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2005.

[EH85] E. A. Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1) :1–24, 1985.

[Eme90] E. Allen Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 995–1072, Cambridge, MA, USA, 1990. MIT Press.

[FBS04a] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. of the 13th International World Wide Web Conference (WWW'04)*, USA, 2004. ACM Press.

[FBS04b] X. Fu, T. Bultan, and J. Su. Wsat : A tool for formal analysis of web services. In *Proc. of the 16th International Conference on Computer Aided Verification (CAV'04)*, 2004.

[Fer04] Andrea Ferrara. Web services : a process algebra approach. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *ICSOC*, pages 242–251. ACM, 2004.

[FUJ+03] H. Foster, S. Uchitel, J.Magee, , and J.Kramer. Model-based verification of web service compositions. In *Proc. of. the 18th Int. Conf. on Automated Software Eng.*, 2003.

[Gar90] Hubert Garavel. Introduction au langage LOTOS, 1990. Revue de l'Association des Anciens Elèves de l'ENSIMAG.

[Ger02] Marie-Pierre Gervais. Towards an MDA-Oriented methodology. In *COMPSAC '02 : Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life : Development and Redevelopment*, pages 265–270, Washington, DC, USA, August 2002. IEEE Computer Society.

[GGM99] J.-M. Geib, C. Gransart, and P. Merle. *CORBA : des concepts à la pratique, 2ème édition*. DUNOD, 1999.

[GHM+02a] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1 – Messaging Framework, 19 Décembre 2002. http ://www.w3.org/TR/2002/CR-soap12-part1-20021219.

[GHM+02b] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 2 – Adjuncts, 19 Décembre 2002. http ://www.w3.org/TR/2002/CR-soap12-part2-20021219.

[Gro01] William Grosso. *Java RMI*. Robert Eckstein - O'Reilly & Associates, Inc., 2001.

[HHM06] Mehdi Ben Hmida, Serge Haddad, and Valerie Monfort. Dynamically adapting clients to web services changing. In *WEWST'06 : Porceedings of Workshop on Emerging Web Services Technology – à paraître –*, Zurich, Switzerland, December 4–6 2006.

[HKPQ02] Jérôme Hugues, Fabrice Kordon, Laurent Pautet, and Thomas Quinot. A case study of middleware to middleware : Mom and orb interoperability. In *4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[HMMR04a] Serge Haddad, Tarak Melliti, Patrice Moreaux, and Sylvain Rampacek. A dense time semantics for web services specifications languages. In *Proc. of the 1st Int. Conf. on Information & Communication Technologies : from Theory to Applications (ICTTA'04)*, pages 647–648, Damascus, Syria, April 19–23 2004. IEEE France.

[HMMR04b] Serge Haddad, Tarak Melliti, Patrice Moreaux, and Sylvain Rampacek. Modelling web services interoperability. In *Proc. of the 6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, volume 4, pages 287–295, Porto, Portugal, April 14–17 2004.

[HMR06] Serge Haddad, Patrice Moreaux, and Sylvain Rampacek. Client synthesis for web services by way of a timed semantics. In *Proc. of the 8th Int. Conf. on Enterprise Information Systems (ICEIS06)*, Paphos, Cyprus, May 23–27 2006.

[Hol03] Gerard J. Holzmann. *The SPIN MODEL CHECKER - Primer and Reference Manual*. Addison-Wesley, Pearson Education, september 2003.

[HPK03] Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Contributions to middleware architectures to prototype distribution infrastructures. In *14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, pages 124–131. IEEE Computer Society, 2003.

[JHA+96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP : a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[JR06] K. Jeffery and D. De Roure. Future for european grids : Grids and service oriented knowledge utilities. next generation grids expert group report 3, 2006.

[JSM05] M. Juric, P. Sarang, and B. Mathew. *Business Process Execution Language for Web Services*. Packt Publishing, 2005.

[Jur05] M. Juric. BPEL and Java. *On line journal theserverside.com*, 2005. `http://www.theserverside.com/articles/article.tss?l=BPELJava`.

[KBR+05] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto and. Web Services Choreography Description Language Version 1.0, Novembre 2005. W3C Candidate Recommendation.

[KM03] Hubert Kadima and Valérie Monfort. *Les Web Services – Techniques, démarches et outils – XML, WSDL, SOAP, UDDI, Rosetta, UML*. Dunod, mars 2003.

[Ley01] Frank Leymann. Web Services Flow Language (WSFL 1.0), Mai 2001. http ://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2) :134–152, 1997.

[MBD+03] David Martin, Mark Burstein, Grit Denker, Jerry Hobbs, Lalana Kagal, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. DAML-S (and OWL-S) 0.9 draft release, 2003.

[MBG03] Libero Maesano, Christian Bernard, and Xavier Le Galles. *Services Web avec J2EE et .NET : Conception et Implémentation*. Eyrolles, 2003.

[MBSR06] Tarak Melliti, Celine Boutrous-Saab, and Sylvain Rampacek. Verifying correctness of web services choreography. In *Proc. Forth IEEE European Conference on Web Services (ECOWS06) – à paraître –*, Zurich, Switzerland, December 4–6 2006. IEEE Computer Society Press.

[Mel04] Tarak Melliti. *Interopérabilité des services web complexes. Application aux systèmes multi-agents*. PhD thesis, Université Paris IX Dauphine, décembre 2004.

[MG04] Valérie Monfort and Stéphane Goudeau. *Web services et l' interopérabilité des SI : WS-I, WSAD / J2EE, Visual Studio .NET et BizTalk*. Dunod, 2004.

[MH03] Tarak Melliti and Serge Haddad. Synthesis of agents for web services interaction. In *Workshop Semantic Web Services for Enterprise Application Integration and E-Commerce of the Fifth International Conference on Electronic Commerce*, Pittsburgh, USA, Sept 2003.

[Mit02] Nilo Mitra. SOAP Version 1.2 Part 0 – Primer, 19 Décembre 2002. http ://www.w3.org/TR/2002/CR-soap12-part0-20021219.

[MM05] Nikola Milanovic and Miroslaw Malek. Architectural support for automatic service composition. In *SCC '05 : Proceedings of the 2005 IEEE International Conference on Services Computing*, pages 133–140, Washington, DC, USA, 2005. IEEE Computer Society.

[NM03] Srini Narayanan and Sheila McIlraith. Analysis and simulation of web services. *Comput. Networks*, 42(5) :675–693, 2003.

[NS94] X. Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP : Theory and Application, 1994.

[PB05] Frédéric Peschanski and Jean-Pierre Briot. Architectures de composants répartis. In Mourad Oussalah, editor, *Composants : concepts, techniques et outils*. Vuibert, 2005. Chapter 9. (in french).

[Pnu79] Amir Pnueli. The temporal semantics of concurrent programs. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 1–20, London, UK, 1979. Springer-Verlag.

[Ram03] Sylvain Rampacek. Modélisation et validation d'applications basées sur les services web. Master's thesis, Université de Pierre et Maris Curie - Paris 6, 2002-2003. DEA Systèmes Informatiques Répartis. Sous la direction de Claude Dutheillet et Patrice Moreaux.

[SBA+02] Poornachandra Sarang, Christopher Browne, Dietrich Ayala, Vivek Chopra, Kapil Apshankar, and Time McAllister. *Professional Open Source Web Services*. Wrox Press, 2002.

[SBB+99] Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking*. Vuibert, April 1999.

[SBS04] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *ICWS*, pages 43–. IEEE Computer Society, 2004.

[Sma03] Patrick Smacchia. *Pratique de .NET et C#.* O'Reilly, Juin 2003.

[SNMRRP06] Elena Sanchez-Nielsen, Sandra Martin-Ruiz, and Jorge Rodriguez-Pedrianes. An open and dynamical service oriented architecture for supporting mobile services. In *ICWE 2006 – Standford Linear Accelerator Center*, Palo Alto, California, July 12-14, 2006.

[SWK06] Schauerhuber, Wimmer, and Kapsammer. Bridging existing web modeling languages to model- driven engineering : A metamodel for WebML. In *Model-Driven Web Engineering (MDWE 2006)*, Standford Linear Accelerator Center, July 11, 2006.

[TBLCDE+02] IBM Tom Bellwood, Microsoft Luc Clément, IBM David Ehnebuske, IBM Andrew Hately, IBM Maryann Hondo, HP Yin Leng Husband, Microsoft Karsten Januszewski, Oracle Sam Lee, IBM Barbara McKee, Intel Joel Munter, and SAP Claus von Riegen. UDDI Version 3, 19 Juillet 2002. http ://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[Tha01] Satich Thatte. XLANG - Web Services For Business Process Design, 2001. http ://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm.

[Tur05] Kenneth J. Turner. Formalising web services. In *Proc. of Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume LNCS 3731, pages 473–488, Taipei, Taiwan, october 2005. Springer.

[Whi75] J. E. White. RFC 707 : High-level framework for network-based resource sharing, December 1975. http ://www.ietf.org/rfc/rfc707.txt ?number=707.

[YCB+04] François Yergeau, John Cowan, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.1, 4th February 2004. http ://www.w3.org/TR/2004/REC-xml11-20040204/.

[Yov97] Sergio Yovine. Kronos : A verification tool for real-time systems. (kronos user's manual release 2.2). *Software Tools for Technology Tranfer*, 1 :123–133, 1997.

Annexes

Annexe A

Fichiers de règles

Les fichiers de règles permettent d'indiquer, indépendamment du code, comment évolue un processus donné en fonction des règles issues de notre algèbre de processus présentée lors des chapitres 4 et 6. La première partie de ce chapitre présente la syntaxe utilisée par ces fichiers, et les deuxième et troisième parties présentent le fichier correspondant aux deux sémantiques (temps discret et temps dense).

A.1 Syntaxe

Il existe cinq types de lignes (le type est déterminé par le premier caractère de la ligne) :
- *ligne* `T` : ce type de ligne ne doit être rencontré qu'une seule fois dans un fichier de règles. Il permet de définir si les règles utilisent le temps discret (`T LTS`) ou le temps dense (`T TA`).
- *ligne* `O` : cette ligne permet de définir un opérateur. Par exemple `O sequence [(p) (p)]` définit l'opérateur `sequence` comme acceptant deux paramètres de types `P`. Ce type de ligne est suivi par des ensembles de lignes de types `R` et `G`, mais également de types `S`.
- *ligne* `R` : cette ligne indique l'action possible, que le processus définit par la ligne `O`, peut réaliser. Ainsi, `R DONE [0]` indique que le processus peut exécuter l'action `DONE` (correspondant à l'action $\surd$) et devenir le processus `0` (correspondant au processus *null*). Cette ligne peut être gardée par une ou plusieurs lignes de type `G`.
- *ligne* `G` : cette ligne indique une garde concernant l'activation des lignes de types `R`. Ainsi, `G #1 A` indique : si le paramètre 1 (qui doit alors être de type `p`) peut réaliser une action de type `A`, alors la garde est vérifiée. De même, `G #1 not DONE` indique : si le paramètre 1 ne peut réaliser l'action $\surd$, alors la garde est vérifiée.
- *ligne* `S` : cette ligne indique comment simplifier un processus. Par exemple, `S #1 empty / #2` indique : dans le cas où le paramètre 1 correspond au processus `empty`, alors le processus est simplifiable et se résume au paramètre 2.

Dès qu'une ligne comporte le caractère %, la suite est ignorée (principe des commentaires).

L'opérateur `#` permet d'accéder à un paramètre précis. Ainsi `#1` permet d'accéder au premier paramètre de l'opération.

Utiliser `#1'` dans une ligne de type `R` consiste à prendre le paramètre dérivé après avoir exécuté l'action précisée sur la même ligne.

Voici les différents types de paramètres :

- type `b` : correspond à un booléen ;
- type `i` : correspond à un entier (l'utilisation de `#i--` indique la création d'un nouvel entier décrémenté de 1) ;
- type `m` : correspond à un message ;
- type `e` : correspond à une exception ;
- type `p` : correspond à un processus ;
- type `s` : correspond à un ensemble. Ainsi, `(s (p))` indique un ensemble dont les éléments sont de types processus. Voici comment accéder aux différents éléments :
  - `#i` : permet d'accéder à tous les éléments de l'ensemble ;
  - `#i.s1` : permet d'accéder à un élément de l'ensemble répondant à un certain critère ;
  - `#i.ss` : permet d'accéder au plus grand sous-ensemble correspondant à un certain critère ;
  - `#i.s1c` : permet d'accéder à tous les éléments de l'ensemble privé de l'élément `#i.s1` sélectionné plus haut ;
  - `#i.ssc` : permet d'accéder à tous les éléments de l'ensemble privé des éléments `#i.ss` sélectionnés plus haut ;
- type `c` : correspond au type couple. Ainsi, `(c (m p))` correspond au type couple dont le premier élément est de type message et le deuxième de type processus. `#i.1` et `#i.2` permettent d'accéder aux deux sous-éléments.

Voici la liste des différentes actions (correspondant à la version ASCII des actions qu'un service web peut réaliser) :

- action `A` : correspond à toutes les actions possibles ;
- action `EX` : correspond à une exception ;
- action `TAU` : correspond à l'action silencieuse ou interne τ ;
- action `DONE` : correspond à l'action terminaison $\surd$;
- action `MSG` : correspond à un message $!m$ ou $?m$;
- action `TIME` : correspond à l'action écoulement de temps χ ;
- action `TO` : correspond à l'action *timeout*.

A.2 Version temps discret

Voici maintenant le code correspondant à la traduction des différentes règles de la sémantique temps discret présentées lors de la section 4.3.

En-tête

La partie suivante permet d'activer le temps discret :

```
% active le mode temps discret
T LTS
```

Le processus time

Le processus time ne peut que réaliser l'action χ pour rester lui-même.

```
% règles pour le processus time
O time []
R TIME [time []]
```

Le processus empty

Le processus empty ne peut que réaliser l'action $\surd$ pour devenir le processus 0 ou *null*.

```
% règles pour le processus empty
O empty []
R DONE [0]
```

Le processus $o[*m]$

Le processus $o[*m]$ peut :
- soit laisser écouler le temps (action `TIME`) et rester lui-même (`#0`).
- soit envoyer ou recevoir un message (`#1`) et devenir le processus empty.

```
% règles pour le processus message
O message [(m)]
R TIME [#0]
R #1 [empty []]
```

Le processus throw

Le processus throw en exécutant une exception, devient le processus *null*.

```
% règles pour le processus raise/throw
O raise [(e)]
R #1 [0]
```

Le processus sequence

Le processus sequence est composé de plusieurs éléments :
- soit le premier processus peut réaliser une action différente de la terminaison (premier ensemble `G`/`R`) ;
- soit le premier processus se termine, alors il laisse la main au deuxième processus (deuxième ensemble `G`/`R`) ;
- enfin, si le premier processus correspond au processus empty, alors la séquence peut se réécrire simplement en considérant le deuxième processus (ligne `S`).

```
% règles pour le processus sequence
O sequence [(p) (p)]
G #1 A
G #1 not DONE
R A [sequence [#1', #2]]
G #1 DONE
G #2 A
R A [#2']
S #1 empty / #2
```

Le processus switch

Le processus switch est le premier processus utilisant un ensemble pour paramètre. Ainsi, la règle indique qu'il faut, pour chaque élément e de l'ensemble, produire une action τ dont la cible est l'élément e.

```
% règles pour le processus switch
O switch [(s (p))]
R TAU [#1]
```

Le processus while

Le processus while consiste à la possibilité d'exécuter deux actions τ. Dans le premier cas, le processus devient le processus empty. Dans le deuxième cas, il est « transformé » en un processus exécutant séquentiellement le corps de la boucle et à nouveau la boucle *while* elle-même.

```
% règles pour le processus while
O while [(p)]
R TAU [empty []]
R TAU [sequence [#1, while[#1]]]
```

Le processus flow

Le processus flow présente bien les différentes règles définies par notre algèbre :

- la première partie consiste à n'exécuter que les actions silencieuses, les autres processus n'évoluant pas. L'utilisation de `s1` à la place de `ss` permet ici d'éviter d'agréger les actions silencieuses par la même transition.
- la deuxième partie concerne les exceptions ;
- la troisième partie concerne l'exécution des évéments pouvant survenir durant l'exécution (si aucun des processus ne peut exécuter une action silencieuse ou une exception) ;
- la quatrième partie concerne l'agrégation des différentes actions liées au passage du temps (petite précision : soit l'ensemble des processus est capable de réaliser cet écoulement de temps, soit seulement un sous-ensemble peut réellement l'exécuter, mais les autres doivent avoir terminé leur exécution) ;
- enfin, la cinquième partie concerne l'agrégation des actions liées à la terminaison des processus.

```
% règles pour le processus all/flow
O all [(s (p))]
G #1.s1 TAU
R TAU [all [#1.s1', #1.s1c]]

G #1.s1 EX
R EX [0]

G #1.s1 MSG
G #1.s1c not TAU
G #1.s1c not EX
R MSG [all [#1.s1', #1.s1c]]

G #1.ss TIME
G #1.ssc DONE
R TIME [all [#1.ss', #1.ssc]]

G #1 DONE % tous
R DONE [0]
```

Le processus scope

Les paramètres du processus scope sont les suivants :

1. un processus principal ;
2. un ensemble des évènements à traiter (un message associé à un processus à exécuter) ;
3. un temps maximal d'exécution (valeur entière) ;
4. un processus à exécuter en cas de *timeout* ;
5. un ensemble des exceptions à intercepter.

Les différentes règles correspondent à (numérotées par groupe G/R) :

1. l'interception des évènements prévus ;
2. l'interception des exceptions prévues ;
3. l'interception d'une exception non prévue (dérivation sur le processus *null*) ;
4. l'écoulement de temps si possible (*timeout* non atteint, création d'un nouveau processus scope dont le corps a évolué et la valeur d'horloge est décrémentée) ;
5. l'exécution de l'écoulement de temps si le processus associé au *timeout* est *null* ;
6. l'exécution du processus associé au *timeout* si nécessaire ;
7. l'exécution de l'action de terminaison du processus principal ;
8. l'exécution de la réception ou l'envoi d'un message par le processus principal ;
9. l'exécution d'une action silencieuse par le processus principal ;
10. la simplification au processus empty si le processus principal est empty.

% règles pour le processus context/scope

```
O context [(p) (s (c (m p))) (i) (p) (s (c (e p)))]

G #1 not EX
G #1 not DONE
G #1 not TAU
R #2.1 [#2.2] % pour tous les couples

G #1 #5.s1.1
R TAU [#5.s1.2]

G #1 EX
G #1 not #5.s1.1
R EX [0]

G #4 not null
G #3 not 0
G #1 TIME
R TIME [context [#1', #2, #3--, #4, #5]]

G #4 null
G #1 TIME
R TIME [context [#1', #2, #3, #4, #5]]
```

```
G #4 not null
G #3 0
G #1 TIME
R TIME [#4]

G #1 DONE
R DONE [0]

G #1 MSG
G #1 not #2.1
R MSG [context [#1', #2, #3, #4, #5]]

G #1 TAU
G #1 not #2.1
R TAU [context [#1', #2, #3, #4, #5]]

S #1 empty / #1
```

A.3 Version temps dense

La version temps dense du fichier de règles est très similaire à la version temps discret, nous ne traiterons que les éléments ayant changés.

En-tête

La partie suivante permet d'activer le temps dense :

```
% active le mode temps dense
T TA
```

Le processus time

Le processus time, bien que ne réalisant aucune action dans le cas temps dense, est indispensable.

```
% règles pour le processus time
O time []
```

Le processus empty

```
% règles pour le processus empty
O empty []
R DONE [0]
```

Le processus $o[*m]$

```
% règles pour le processus message
O message [(m)]
R #1 [empty []]
```

Le processus throw

```
% règles pour le processus raise/throw
O raise [(e)]
R #1 [0]
```

Le processus séquence

```
% règles pour le processus sequence
O sequence [(p) (p)]
G #1 A
G #1 not DONE
R A [sequence [#1', #2]]
G #1 DONE
G #2 A
R A [#2']
S #1 empty / #2
```

Le processus switch

```
% règles pour le processus switch
O switch [(s (p))]
R TAU [#1]
```

Le processus while

```
% règles pour le processus while
O while [(p)]
R TAU [empty []]
R TAU [sequence [#1, while[#1]]]
```

Le processus flow

Deux éléments importants pour le processus flow : la gestion du *timeout* doit se faire en deux actions pour autoriser ou non l'agrégation du *timeout* en une seule transition (correspondant à des *timeout* se déclenchant en même temps).

```
% règles pour le processus all/flow
O all [(s (p))]

G #1.s1 TAU
R TAU [all [#1.s1', #1.s1c]]

G #1.s1 TO % timeout indépendant
R TO [all [#1.s1', #1.s1c]]

G #1.ss TO % le timeout peut se déclencher en même temps !
R TO [all [#1.ss', #1.ssc]]
```

```
G #1.s1 EX
R EX [0]

G #1.s1 MSG
G #1.s1c not TAU
G #1.s1c not EX
R MSG [all [#1.s1', #1.s1c]]

G #1 DONE % tous
R DONE [0]
```

Le processus scope

Le nouveau paramètre du processus scope correspond à l'indication concernant l'activation de l'horloge ou non. Le reste est inchangé, sauf pour les lignes de type `R` :

- `-[#6, #3, -]` : indique qu'aucune garde de type « *timeout* » ne doit être créée sur cette transition ;
- `-[#6, #3, G]` : indique qu'une garde de type « *timeout* » doit être créée sur cette transition, la valeur maximale de l'horloge est indiquée par le paramètre 3 et l'information quant à l'activation ou non est contenue dans le paramètre 6.

```
% règles pour le processus context
O context [(p) (s (c (m p))) (i) (p) (s (c (e p))) (b)]

G #1 not EX
G #1 not DONE
G #1 not TAU
R #2.1 [#2.2]-[#6, #3, -] % pour tous les couples

G #1 #5.s1.1
%R #5.s1.1 [#5.s1.2]-[#6, #3, -]
R TAU [#5.s1.2]-[#6, #3, -]

G #1 EX
G #1 not #5.s1.1
R EX [0]

G #4 not null
G #1 not DONE
G #1 not TAU
G #1 not EX
R TO [#4]-[#6, #3, G]
G #1 DONE
R DONE [0]

G #1 MSG
G #1 not #2.1
G #1 not TAU
G #1 not EX
R MSG [context [#1', #2, #3, #4, #5, TRUE]]-[#6, #3, -]
```

```
G #1 TAU
G #1 not #2.1
R TAU [context [#1', #2, #3, #4, #5, TRUE]]-[#6,#3, -]

G #1 TO
G #1 not DONE
G #1 not TAU
G #1 not EX
R TO [context [#1', #2, #3, #4, #5, TRUE]]-[#6, #3, -]

S #1 empty / #1
```

Annexe B

Exemples de service web BPEL

B.1 Service web *loanApproval*

Cette section traite un exemple complet de modélisation d'un service web et la génération du modèle de son client. L'exemple est celui fourni de base avec de nombreux serveurs BPEL et couvre la plupart des opérateurs BPEL. De plus, il a l'avantage d'être relativement petit et fournit donc des modèles de petites tailles, facilement compréhensibles.

B.1.1 Description de l'exemple

Le service web étudié ici se nomme *loanApproval*. Son but est d'accepter (ou non) un emprunt d'un certain montant. Suivant le montant, une deuxième confirmation est demandée.

Le processus principal est un flow. Les différentes actions que le service doit réaliser sont les suivantes :

- un emprunteur envoie la demande avec le montant (opération receive)
- suivant le montant, une demande est envoyée à un service (opération invoke)
- si ce montant excède une certaine valeur, la demande est alors envoyée à un deuxième service (opération invoke)
- enfin, la réponse est envoyée à l'emprunteur (opération reply)

L'exemple peut paraître déroutant : comment est-il possible d'envoyer en parallèle la demande de validation du prêt aux deux sous-services et envoyer en même temps la réponse à l'emprunteur ? Cela est possible par une utilisation (que l'on peut considérer comme abusive) des *links* (voir section 7.2.3 pour une explication des *links*), ceci à un tel point que les opérations sont alors exécutées séquentiellement !

B.1.2 Fichier de description BPEL de l'exemple

Cette section présente la description BPEL du service, tel qu'il est possible de l'obtenir lorsque l'on se connecte au serveur proposant ce service. Pour faciliter la lecture, il a été coupé en deux morceaux : l'en-tête, définissant les différentes liaisons entre les partenaires (sous-service), et le processus en lui-même.

```
<!-- BPEL Process Definition                                          -->
<!-- Edited using ActiveWebflow(tm) Designer version 1.0.0           -->
<!-- (http://www.active-endpoints.com)                               -->
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:apns="http://tempuri.org/services/loanapprover"
         xmlns:asns="http://tempuri.org/services/loanassessor"
         xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:lns="http://loans.org/wsdl/loan-approval"
         xmlns:loandef="http://tempuri.org/services/loandefinitions"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         name="loanApprovalProcess" suppressJoinFailure="yes"
         targetNamespace="http://acme.com/loanprocessing">
   <partnerLinks>
      <partnerLink myRole="approver" name="customer"
                   partnerLinkType="lns:loanApprovalLinkType"/>
      <partnerLink name="approver" partnerLinkType="lns:loanApprovalLinkType"
                   partnerRole="approver"/>
      <partnerLink name="assessor" partnerLinkType="lns:riskAssessmentLinkType"
                   partnerRole="assessor"/>
   </partnerLinks>
   <variables>
      <variable messageType="loandef:creditInformationMessage" name="request"/>
      <variable messageType="asns:riskAssessmentMessage" name="riskAssessment"/>
      <variable messageType="apns:approvalMessage" name="approvalInfo"/>
      <variable messageType="loandef:loanRequestErrorMessage" name="error"/>
   </variables>
   <faultHandlers>
      <catch faultName="apns:loanProcessFault" faultVariable="error">
         <reply faultName="apns:loanProcessFault" operation="approve"
                partnerLink="customer" portType="apns:loanApprovalPT"
                variable="error"/>
      </catch>
   </faultHandlers>
```

La suite du code présente la description du processus métier du service *loanApproval* en lui-même. Pour faciliter la lecture les éléments clés par rapport à la description des différentes actions expliquées plus haut ont été mis en gras.

```
   <flow>
      <links>
         <link name="receive-to-approval"/>
         <link name="receive-to-assess"/>
         <link name="approval-to-reply"/>
         <link name="assess-to-setMessage"/>
         <link name="assess-to-approval"/>
         <link name="setMessage-to-reply"/>
      </links>
      <receive createInstance="yes" name="receive1" operation="approve"
               partnerLink="customer" portType="apns:loanApprovalPT"
               variable="request">
         <source linkName="receive-to-approval"
                 transitionCondition="bpws:getVariableData('request',
                                                           'amount')&gt;=10000"/>
         <source linkName="receive-to-assess"
                 transitionCondition="bpws:getVariableData('request',
                                                           'amount')&lt;10000"/>
      </receive>
      <invoke inputVariable="request" name="invokeapprover" operation="approve"
              outputVariable="approvalInfo" partnerLink="approver"
              portType="apns:loanApprovalPT">
         <target linkName="receive-to-approval"/>
         <target linkName="assess-to-approval"/>
         <source linkName="approval-to-reply"/>
      </invoke>
```

```
      <invoke inputVariable="request" name="invokeAssessor" operation="check"
              outputVariable="riskAssessment" partnerLink="assessor"
              portType="asns:riskAssessmentPT">
         <target linkName="receive-to-assess"/>
         <source linkName="assess-to-setMessage"
                 transitionCondition="bpws:getVariableData('riskAssessment',
                                                           'risk')='low'"/>
         <source linkName="assess-to-approval"
                 transitionCondition="bpws:getVariableData('riskAssessment',
                                                           'risk')!='low'"/>
      </invoke>
      <reply name="reply" operation="approve" partnerLink="customer"
             portType="apns:loanApprovalPT" variable="approvalInfo">
         <target linkName="approval-to-reply"/>
         <target linkName="setMessage-to-reply"/>
      </reply>
      <assign name="assign">
         <target linkName="assess-to-setMessage"/>
         <source linkName="setMessage-to-reply"/>
         <copy>
            <from expression="'approved'"/>
            <to part="accept" variable="approvalInfo"/>
         </copy>
      </assign>
   </flow>
</process>
```

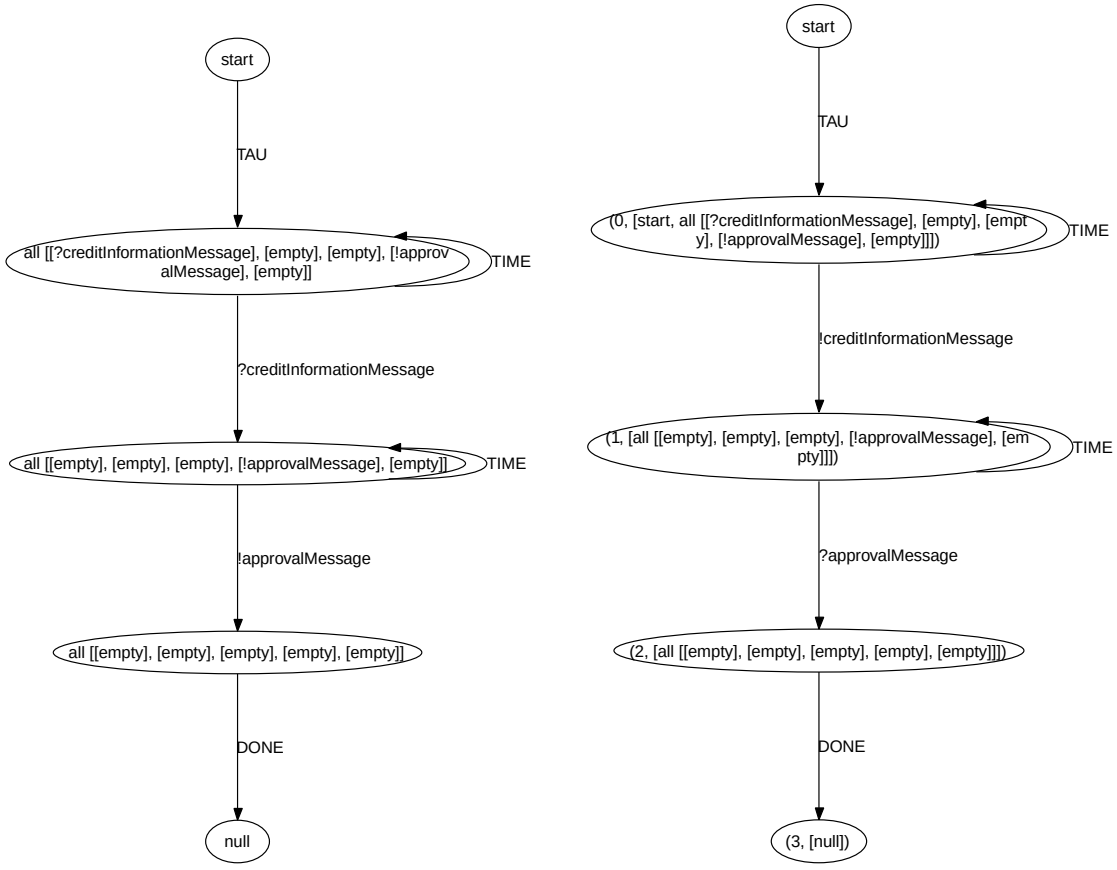
B.1.3 Approche temps discret

La figure B.1 représente la modélisation en temps discret du processus *loanApproval*, c'est-à-dire le TIOTS de la partie service, généré à partir du processus BPEL et l'application des règles sémantiques, ainsi que le TIOTS de la partie cliente, généré à partir de la relation d'interaction.

B.1.4 Approche temps dense

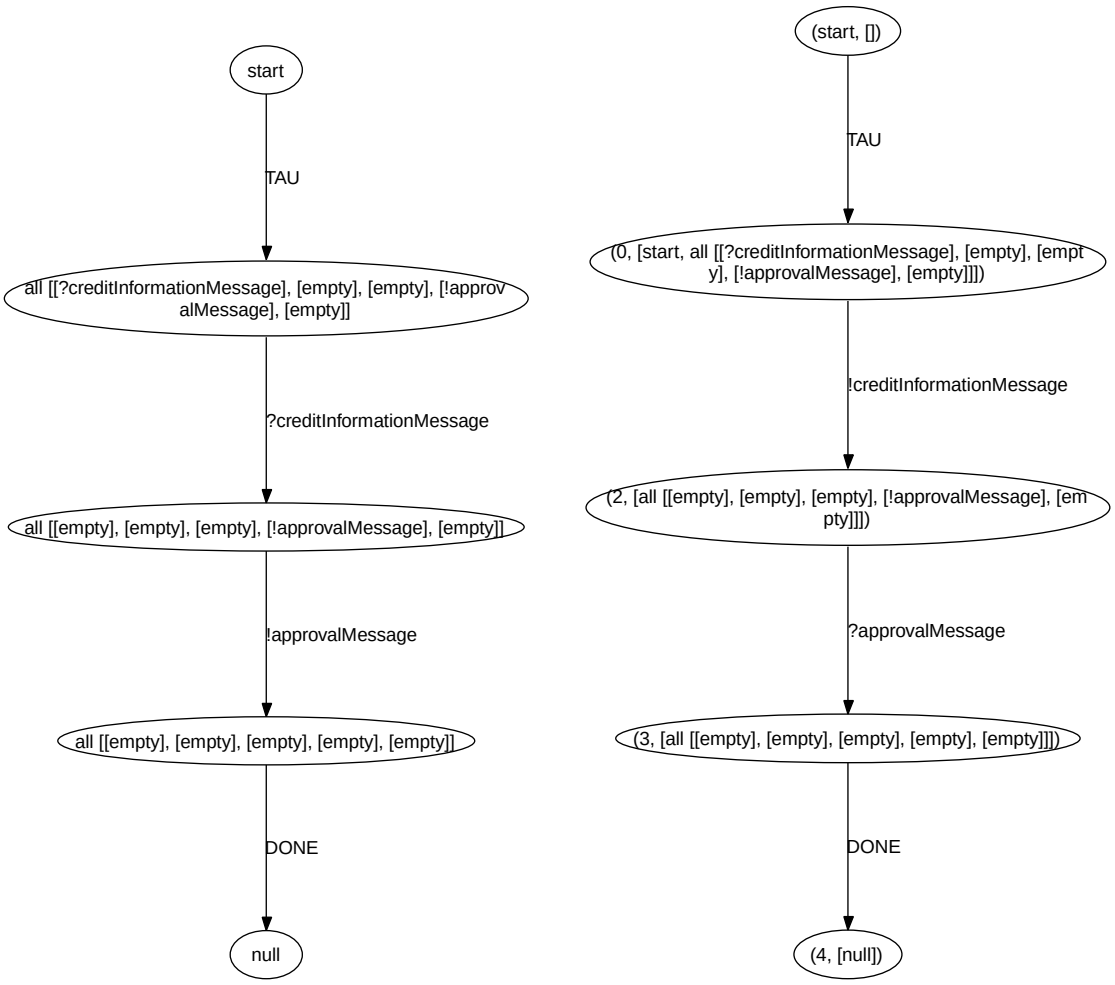
De la même manière, la figure B.2 représente la modélisation en temps dense du processus *loanApproval*, c'est-à-dire l'automate temporisé de la partie service, généré à partir du processus BPEL et l'application des règles sémantiques, ainsi que l'automate temporisé de la partie cliente, généré à partir de la relation d'interaction.

Enfin, la figure B.3 représente la modélisation de la partie service du processus *loanApproval* en modélisant les échanges entre le service et les sous-services. Ce modèle sera celui utilisé par un algorithme visant à vérifier l'interaction entre les différents partenaires d'une chorégraphie.



(a) Modèle de la partie service (b) Modèle de la partie cliente

FIG. B.1 – Modélisation temps discret du processus *loanApprocal*



(a) Modèle de la partie service (b) Modèle de la partie cliente

FIG. B.2 – Modélisation temps dense du processus *loanApprocal*

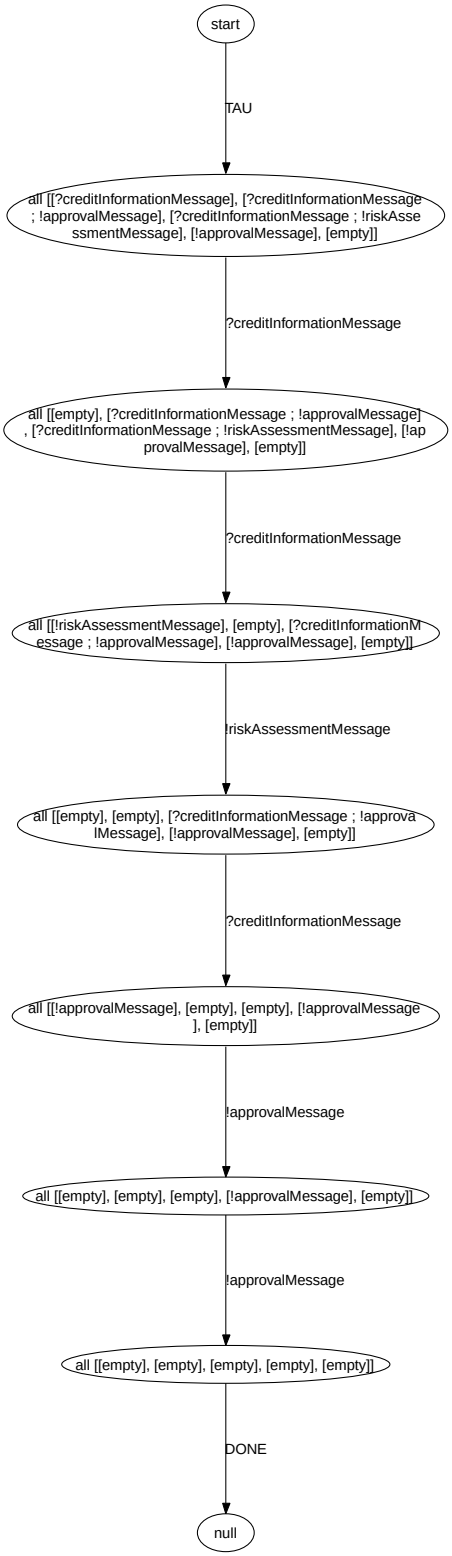


FIG. B.3 – Modélisation temps dense du processus *loanApprocal*, partie service, avec modélisation des échanges liés au processus invoke

B.2 Exemple d'utilisation de la hiérarchie d'horloges

Cet exemple propose une modélisation d'un service mettant en évidence la hiérarchie d'horloges vu lors de la section 6.3.5. La modélisation ne se fera qu'en temps dense pour permettre d'avoir les horloges.

Le processus métier est le suivant (l'écriture H1-2 indique que l'horloge associée au processus contenant cette déclaration se nomme H1 et expire après 2 unités de temps) :

```
flow (
    (
        scope (
            scope ( !impl :StartCtxtSub, {} (H1-2, !im :TOSub) {} )
            {} (H2-2, !im :TO1) {} )
    ),
    (
        scope ( !impl :StartCtxt2, {} (H3-2, !im :TO2) {} )
    )
)
```

Le modèle obtenu pour la partie service se trouve figure B.4 et le modèle obtenu pour le client se trouve figure B.5. Les transitions issues des deux premiers états mettent bien en évidence les initialisations d'horloges et les différents cas rencontrés dans la hiérarchisation de ces horloges.

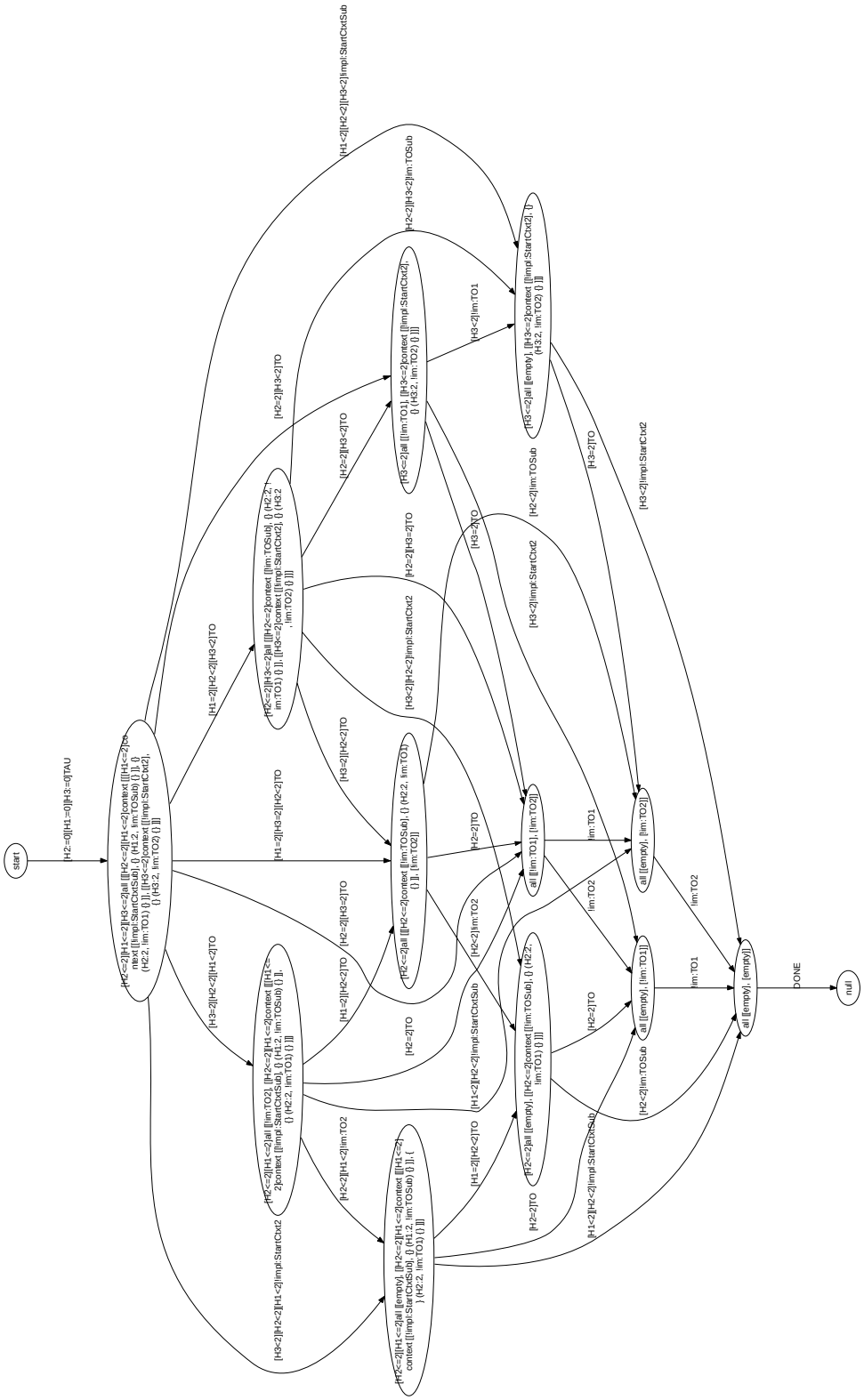


FIG. B.4 – Exemple d'automate temporisé représentant un service avec une hiérarchie d'horloge

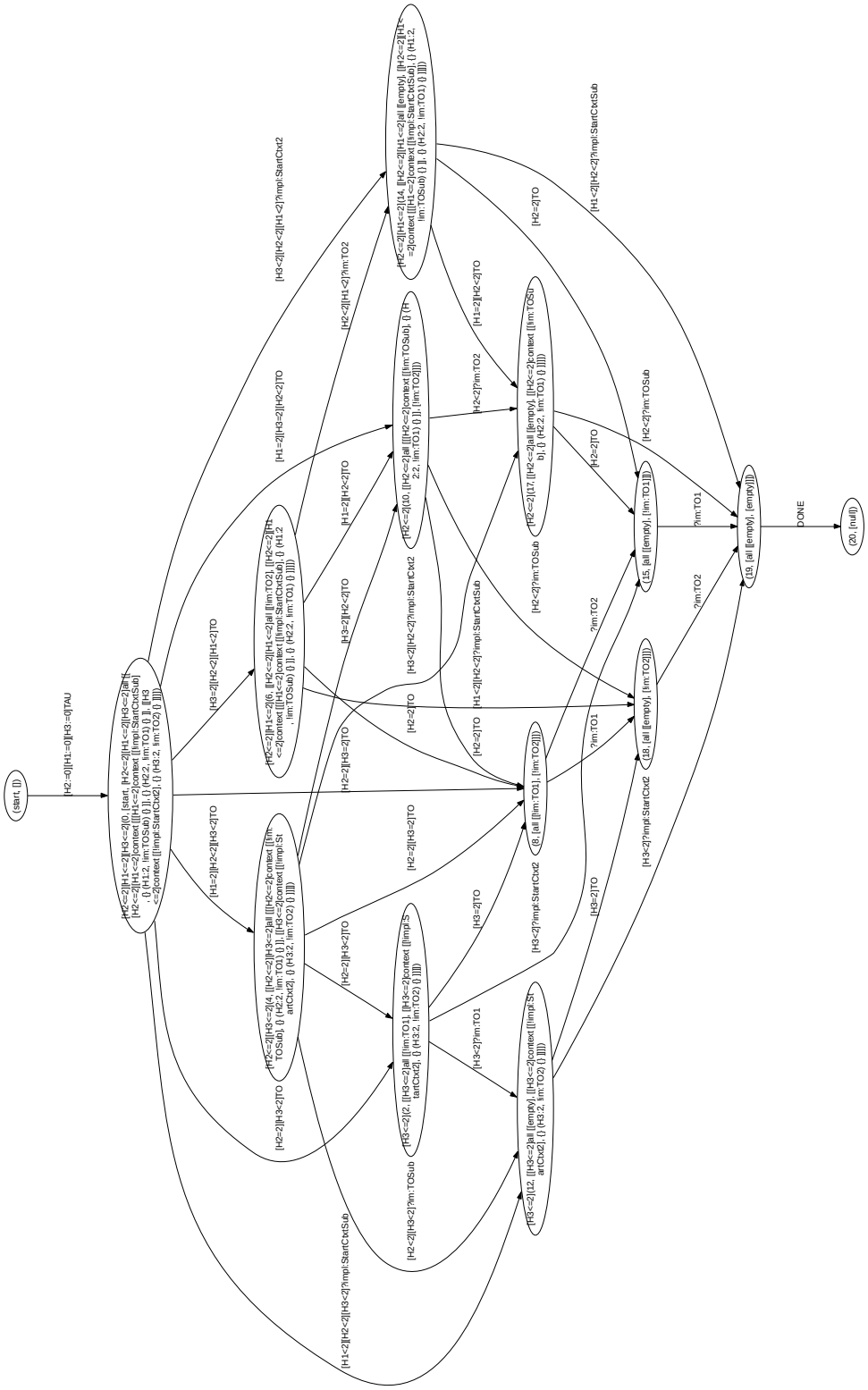


FIG. B.5 – Automate temporisé représentant le client du service de la figure B.4

Index

Sémantique, interactions et langages de description des services web complexes

Résumé :

Cette thèse de Doctorat s'articule autour des services web dont le processus métier est décrit en utilisant un langage de description comportementale tel que XLANG ou BPEL. Nous fournissons une méthode, des algorithmes et leurs implémentations permettant de modéliser sous forme de systèmes de transitions temporisées (TIOTS en temps discret et automates temporisés en temps dense) le comportement de ces services web. Pour cela, nous avons fixé une sémantique à l'aide des algèbres de processus temporisés nous permettant de décrire formellement le comportement, en terme d'actions, d'exceptions et d'évènements temporels, de chaque opérateur de ces langages de descriptions comportementales. Ces descriptions sont alors directement traduisibles dans le système de transitions temporisées choisi. Par la suite, grâce à notre relation d'interaction, nous pouvons détecter la non-ambiguïté d'un tel service ainsi modélisé, assurant l'interaction possible avec un client adapté dont nous fournissons alors le modèle. Nous avons également étendu notre approche à la possibilité de vérifier l'interaction des différents partenaires d'une chorégraphe, permettant ainsi de détecter une possible ambiguïté dans cette interaction.

Mots-clés : service web, modèle, comportement, vérification, sémantique, interaction, chorégraphie, TIOTS, automate temporisé, algèbre de processus temporisés.

Semantics, interactions, and description languages of complex web services

Abstract:

This PhD thesis is about web services, in which business process is described by using a behaviour description language, such as XLANG or BPEL. We give a method, algorithms and their implementations enable us to modelling in timed transition systems (TIOTS in discrete time and timed automaton in dense time) the behaviour of these web services. We give semantics with algebra of timed processes, enabling us to describe formally the behaviour, concerning actions, exceptions and timed events, of each operator of these behaviour description languages. These descriptions are directly translating in the given timed transition systems. In the next, with our interaction relation, we can detect the non ambiguity of this service, providing possible interaction with an adapted client that we produce the model. We extend our approach to the possibility to verify the interaction of different partners of a given choreography, enable us to detect a possible ambiguity in this interaction.

Key words : web service, modelling, behavior, verification, semantics, interaction, choreography, TIOTS, timed automata, algebra of timed processes.