UNIVERSITÉ DU QUÉBEC À MONTRÉAL

TEST UNITAIRE DE PROCESSUS BPEL : GÉNÉRATION ORIENTÉE CHEMINS DE CAS DE TEST

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOLAY EL MEHDI ALAOUI SELSOULI

SEPTEMBRE 2010

REMERCIEMENTS

Cet espace ne peut me suffire à remercier tous les gens qui le méritent. Que les personnes qui n'y sont pas citées m'en excusent et acceptent mes remerciements les plus sincères.

Je souhaite tout d'abord adresser ma profonde gratitude à mon directeur de recherche, Monsieur ***Aziz Salah***, professeur à l'Université du Québec à Montréal, pour son appui et encouragement constant. Je ne le remercie jamais assez pour la rigueur intellectuelle, la disponibilité, et la patience dont il a fait preuve, ainsi que pour le support financier qu'il m'a accordé durant la réalisation de ce projet. Je garderai spécialement un doux souvenir de son approche amicale peu commune. Que ce mémoire lui soit le modeste témoignage de ma reconnaissance et de mon plus grand respect.

Je remercie aussi tous mes collègues du laboratoire LATECE ainsi que mes amis, en particulier Wassim, Mohamed, Abdelhalim, Gino, Houda, Ghizlane, Romdhane et Kadour, qui ont répondu à mes questions, supporté mes petites angoisses, mes remises en question et accepté mes petites excentricités.

Je tiens finalement à dédier ce travail à mon père My Ismaïl, ma mère Rachida, ma sœur Aziza, ainsi que mes frères Youssef, Jaâfar et Salah, qui m'ont toujours soutenu dans les moments les plus difficiles et sans qui je n'y serais jamais arrivé.

Merci à tous. Ce fut un long chemin.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Dans ce mémoire, nous proposons une méthode pour la génération des cas de test pour des processus métiers exprimés en langage BPEL. Cette méthode tient compte de la nature concurrente de ce langage ainsi que des caractéristiques qui lui sont propres. Elle adopte une approche *orientée chemin*. Ce faisant, ladite méthode définit une extension de la version usuelle du graphe de flot de contrôle (CFG) - baptisée *BPEL Control Flow Graph* (*B-CFG*) - pour la traduction du code BPEL en un modèle. En parcourant ce modèle, des chemins de test concurrents, qui sont à habiller par la suite par des données de test, sont générés. Ces chemins concurrents sont représentés d'une façon formelle et appropriée. La génération de ces chemins se fait selon différents critères de couverture. Ces critères, qui ont été à la base définis pour la programmation séquentielle, nous les avons adaptés de façon à pouvoir les appliquer pour le cas d'un langage concurrent comme le BPEL. Finalement des propositions de pistes de solution sont fournies pour la question de génération de données de test, ainsi que pour la question de spécification/exécution des cas de test.

INTRODUCTION

Motivation

Oeuvrant dans un décor où la concurrence est de plus en plus rude et où le client est roi, les entreprises qui espèrent fidéliser leur clientèle, en répondant à ses besoins changeants, se trouvent souvent contraintes à réorganiser et à repenser leurs processus métier. Or, on constate que plusieurs de ces entreprises disposent d'applications logicielles qui, étant développées en différentes technologies, sont de nature hétérogènes. Cette hétérogénéité des applications rend leur intégration difficile et entrave ainsi la mise en place de processus métier qui sont flexibles quant à leur réorganisation et maintenance.

Afin de remédier à cela, un grand nombre de telles entreprises penche pour l'*architecture orientée service* (*SOA*) et les séduisants concepts qui en découlent. Cette dernière, que le *Gartner Group* (17) prédit comme une dominante pratique pour les années à venir (44), permet à ce que les applications existantes soient consolidées tout en favorisant le dialogue entre elles (11). En d'autres mots, dans le cadre de cette architecture, les applications sont exposées comme des services qui sont de nature homogènes et qui peuvent être intégrés à des applications plus complexes jusqu'à arriver à supporter, dans sa globalité, le processus métier de l'entreprise.

L'une des implémentations les plus importantes et les plus répandues de l'architecture SOA est la pile des *services Web WS-**. Un service Web WS-* se définit comme une composante logicielle encapsulant une fonctionnalité qui peut être composée avec d'autres pour former une composante (et donc une fonctionnalité) de plus forte granularité. Cette composition des services Web WS-* est assurée par le langage *BPEL* (*Business Process Exécution Language*). À base d'une syntaxe XML, ce langage permet d'exprimer des processus métier qui sont flexibles quant à leur réorganisation et maintenance. Supporté par les vendeurs les plus importants du paysage informatique (e.g. IBM, Microsoft, Sun et Oracle), ce langage BPEL est « devenu le standard *de facto* dans cette arène de la composition de service »(64). En effet, on constate que de plus en plus de compositions de service sont réalisées à base du BPEL. Ce fait constaté fait jaillir la nécessité et la criticité de l'assurance d'une certaine qualité du code BPEL développé.

Parmi les démarches clés pour l'assurance d'un niveau de qualité figure le test logiciel. Ce dernier consiste, sous toutes ses formes, à détecter des anomalies restant dans l'application développée. En particulier, le test unitaire orienté boîte blanche (i.e., basé sur le code) est une forme de test qui a fait ses preuves pour une détection *précoce* des anomalies qui aide à l'amélioration de la qualité en pratique (13). Malheureusement, l'expérience nous a montré que, sous toutes ses formes, le test dans un cycle de développement normal représente 30% à 50% du temps de développement logiciel (75). Ainsi, il y a besoin de penser à la mise en place d'une méthode (d'un outil) qui facilite l'automatisation du test afin de réduire les coûts et les délais de développement. Il va sans dire que cela est aussi valable pour le cas du test unitaire appliqué aux processus BPEL, et surtout si ces processus sont réalisés dans le cadre d'un *développement piloté par les tests* ou selon la méthode *Extreme Programming*.

En revanche, il y a peu de travaux qui ont traité de cette question du test unitaire des compositions BPEL, en particulier sous sa forme orientée boîte blanche. Les travaux existants ainsi qu'une partie des éditeurs BPEL actuellement disponibles (e.g., *activeBPEL Designer*) se sont plutôt penchés sur le volet *exécution* des cas de test unitaires[1] que sur leur *génération*. Or, on sait à quel point les coûts et les délais peuvent être réduits si les cas de tests étaient générés automatiquement. Les travaux qui ont essayé d'aborder ce volet de génération automatique des cas de test se comptent sur le bout des doigts. Si une bonne partie de ces travaux n'a pas tenu compte de certaines caractéristiques importantes du BPEL, le reste de ces travaux l'a abordé d'une façon pas toujours appropriée.

Notre travail vient donc enrichir ces travaux en proposant une méthode pour une génération automatique des cas de test unitaires, selon une approche *orientée chemins*. Cette méthode traite les caractéristique les plus importantes et les plus particulières du langage BPEL d'une façon appropriée.

Problématique et objectifs

L'approche orientée chemins, que nous adoptons pour notre méthode de génération des cas de test, demande à ce que le programme (le processus BPEL) à tester soit traduit en un modèle. Dans notre cas, ce modèle correspondra à un graphe de flot de contrôle (i.e., à un CFG).

---

[1] Les cas de test unitaires sont composés de données de test ainsi que de composants logiciels légers simulant les parties connexes à la partie de code (unité de code) testée.

Par la suite, et en restant toujours fidèle à cette approche orientée chemin, notre méthode se basera sur le graphe représentant le processus à tester pour procéder en deux étapes : (a) une première étape qui consiste en la sélection des chemins à couvrir, à partir de ce graphe, et (b) une deuxième correspondant à la génération des données de test appropriées pour ces chemins. Finalement, les données et chemins de test ainsi générés seront combinés pour constituer des cas de test abstraits qui sont à transformer en des cas de test exécutables.

Cependant, il se trouve que le langage BPEL est un langage pouvant exprimer des comportements (i.e., scénarios d'exécution) concurrents complexes, d'autant plus qu'il a des caractéristiques assez particulières, par rapport même aux autres langages (e.g., Ada) de la programmation concurrente. Cela impose à ce que de nouvelles solutions soient proposées pour la prise en compte de ces caractéristiques et de cette nature concurrente propres à ce langage, tant lors de la traduction du code BPEL en CFG que lors de l'exécution des deux étapes précitées. Notre travail essaye donc d'apporter des (pistes de) réponses aux questions suivantes :

- Comment adapter le modèle CFG pour représenter du code écrit en un langage aussi complexe que le BPEL (un langage concurrent et ayant ses propres particularités comportementales), alors que l'on sait qu'un tel modèle a été à la base utilisé pour la représentation de programmes séquentiels relativement simples?
- Comment rendre la traduction du code BPEL en CFG quasi-systématique?
- Comment représenter, d'une façon formelle et appropriée, les chemins de test à générer; chemins qui pourraient représenter des scénarios d'exécution concurrents ou séquentiels?
- Comment parcourir le graphe CFG représentant le code BPEL à tester pour générer ces chemins de test?
- Selon quels critères de couverture ces chemins devraient-ils être sélectionnés?
- Comment adapter les critères de couverture, que l'on a définis pour la programmation séquentielle, de façon à pouvoir les appliquer à un langage concurrent comme le BPEL?
- Comment produire les données de test qui habilleront les chemins de test générés?
- Comment définir des cas de test exécutables à partir de cas de test abstraits?

Plan du mémoire

Le présent mémoire est organisé comme suit :

- Suite à notre chapitre introductif, un deuxième reprend le flambeau pour la présentation de l'architecture SOA, des services Web WS-* et du langage BPEL.
- Le troisième chapitre présente le test logiciel, de même que la relation qui lie certaines de ses variantes à la théorie des graphes.
- Le quatrième chapitre recense et présente les travaux qui ont abordé cette question de la génération des cas de test, d'autant qu'il présente le nôtre d'une façon sommaire.
- Le cinquième chapitre expose notre traduction du BPEL en graphe B-CFG (*BPEL Control Flow Graph*).
- Le sixième chapitre décrit notre représentation formelle des chemins concurrents de test, notre algorithme pour la génération de ces chemins, notre adaptation de certains critères de couverture que nous empruntons de la programmation séquentielle, ainsi que les pistes que nous proposons pour la génération des données et l'obtention de cas de test exécutables.
- Le septième et dernier chapitre présente les résultats d'application de notre méthode de génération de cas de test sur deux exemples différents de processus BPEL.

Finalement, nous clorons notre mémoire par une conclusion qui reprendra l'essentiel de notre contribution, tout en mentionnant les limites qui y incombent ainsi que les points restants à développer comme perspectives de travail.

CHAPITRE I

LA SOA ET LE BPEL

1.1 Introdution

Dans le présent chapitre, nous allons mettre en lumière le langage BPEL et sa place dans le processus de mise en œuvre d'une architecture orientée service (SOA). Pour cela, nous commençons d'abord par décrire la SOA et les concepts qui en découlent. Ensuite, nous fournirons une brève description de la plateforme des services Web WS-* qui est une des plus importantes solutions implémentant ladite architecture. Finalement, nous décrirons le BPEL qu'on considère comme l'un des composants pilier de cette plateforme, et sur lequel notre travail portera.

1.2 L'architecture SOA

L'*architecture orientée service* (i.e., *SOA*) est présentée par le fameux *Gartner Group* comme le modèle actuel le plus approprié pour la mise en oeuvre des systèmes d'information des entreprises (36). Cette architecture a été élaborée dans l'optique d'apporter de nouvelles solutions pour combler des besoins majeurs comme (19) :

- Permettre aux systèmes d'information d'absorber, d'une façon rapide et efficace, les changements métier dues aux exigences changeantes des clients et partenaires.
- Permettre à ces systèmes de s'adapter d'une façon flexible au développement rapide des nouvelles technologies.
- Permettre l'intégration des applications composant un système d'information, applications qui sont souvent développées en usant de différentes architectures et technologies. Cette intégration est nécessaire pour les systèmes d'information afin qu'ils puissent délivrer des valeurs métier, comme le support efficace à la prise de décision et l'intégrité des données.

La SOA n'a pas surgi du néant et ne peut être présentée comme une architecture radicalement nouvelle. La SOA est plutôt une évolution des méthodes d'intégration (i.e., *EAI*) et des architectures distribuées qui l'ont précédée. En les ayant améliorées et étendues, la SOA reprend le flambeau de ces dernières pour une réutilisabilité, une interopérabilité et une intégration meilleure et efficace des applications existantes (19) — applications que l'on enveloppe comme des services. De fait, l'un des apports majeurs de la SOA est cette possibilité qu'elle donne pour l'élaboration des *processus métiers* comme une composition de services (37).

La SOA ne se résume pas à une technologie donnée ou, autrement dit, n'est liée directement à aucune technologie. La SOA devrait être plutôt vue comme un ensemble de *concepts* à respecter — dire que l'implémentation d'un système d'information est conforme à SOA, c'est dire que cette implémentation respecte l'ensemble de ces concepts. Ci-dessous, nous dressons la liste des concepts SOA des plus importants (19) :

- ***Services.*** Un service doit fournir une fonctionnalité métier, et non pas une fonctionnalité orientée technologie comme la modification d'une table dans une base de données. De plus, un service doit cacher les détails de son implémentation, être autonome et faiblement couplé (i.e., réduire au minimum les dépendances vis-à-vis d'autres services).
- ***Interfaces.*** Une interface est un contrat liant le consommateur de service à son fournisseur. Chaque interface définit un ensemble d'opérations. Ces opérations sont définies comme un ensemble de messages. Ces messages spécifient le format et le type des données à échanger et ne doivent en aucun cas être un support à l'échange de comportements (code d'implémentation), comme c'est le cas pour l'approche composant ou orientée objet.
- ***Synchronicité.*** L'invocation des opérations exposées par les services pourrait être faite selon un mode de communication *synchrone* ou *asynchrone.*
- ***Enregistrement.*** Les services devraient être maintenus dans des registres électroniques de façon à automatiser et faciliter la recherche pour les consommateurs de services.
- ***Qualité de service*** *(i.e., QoS).* Du support devrait être fourni pour les attributs de QoS qu'on associe aux services — attributs comme la sécurité et la fiabilité.
- ***Composition de service.*** Afin de pouvoir incarner le processus métier de l'entreprise, les services devraient être composés selon un ordre particulier et en respect de certaines règles. Ce faisant, les processus métiers seront en mesure de s'adapter facilement et rapidement aux exigences changeantes du marché.

Plusieurs experts s'entendent pour dire que la composition de services en des processus métier est le concept SOA le plus important. On ne peut alors saisir l'importance de la SOA et ses bénéfices engendrés que si l'on atteint le niveau de composition de services (19). Notre travail s'inscrit autour de ce concept clé, et plus précisément dans le cadre du test des processus métiers qui en sont le fruit — processus qui sont exprimés dans un langage dédié qui ne sera autre que le langage BPEL que nous décrivons dans ce qui suit. Comme préalable à la description de ce langage, nous tâcheront d'abord de présenter, du moins d'une façon sommaire, la pile des *services Web WS-** dont BPEL fait partie.

1.3 Les services Web

L'architecture SOA suscite beaucoup d'intérêt vu les concepts attractifs qu'elle présente — concepts que nous avons tenus à rappeler à la section précédente. Néanmoins, il faut disposer d'une implémentation qui soit concrète, basée sur des standards et agissant de façon à permettre de tirer profit de tous les bénéfices promis par cette architecture (37). Une des plus importantes implémentations répondant à ces trois critères est l'implémentation basée sur les *services Web*. Principalement, ces services sont de deux types : les *services Web RESTful* (52) et les *services Web WS-**. Cela dit, dans le cadre d'une SOA, on s'intéresse beaucoup plus au service Web WS-* qu'à leurs congénères RESTful car ces derniers siéent plus à l'implémentation d'une architecture *orientée ressource* (52; 62) qu'à celle *orientée service*.

En fait, l'implémentation la plus appropriée et la plus répandue de SOA est basée sur les services Web WS-*(19). Tout service de ce type est défini par le consortium *W3C* comme une application qui peut être de toute taille, mais à condition (66) :

- D'être identifiée par une URI (i.e., *Uniform Resource Identifier*).
- D'avoir une interface pouvant être définie, décrite et découverte comme un artefact XML.
- D'être en mesure d'interagir directement avec des agents logiciels en échangeant des messages XML à travers des protocoles Internet.

Contrairement à cette première définition qui est relativement abstraite, il existe une deuxième (65) qui, étant fournie par le même organisme, le W3C, inclut déjà les plus importantes *spécifications* des services Web WS-*. Considérées dans leur intégralité, ces spécifications consistent en une panoplie de standards qui sont gérés par d'importants organismes de standardisation (e.g., W3C (68) et OASIS (45)), en plus d'être supportés par les principaux vendeurs de

logiciels dans le marché (e.g., IBM, Microsoft, Oracle et Sun). Cela permet donc de finalement concrétiser le souhait d'une interopérabilité universelle tant espérée. Dans leur début, ces spécifications ont commencé par définir le triangle de base de l'architecture des services Web WS-*. Ce triangle est constitué par les trois standards de base donnés ci-dessous :

- ***WSDL*** *(i.e., Web Services Description Language).* Permet de décrire l'interface publique d'un service à l'aide d'une syntaxe XML, *i.e.*, décrire en XML les opérations offertes par ce service de même que le format des messages qu'elles échangent.
- ***SOAP*** *(i.e., Simple Object Access Protocol).* Représente un cadre général permettant l'échange de données structurées au format XML, selon les deux mode de communication : synchrone et asynchrone. C'est un protocole de transport de données qui est basé sur des protocoles historiques tels que *HTTP* et *SMTP*.
- ***UDDI*** *(i.e., Universal Description Discovery and Integration).* Permet, du côté fournisseur, de publier, dans des annuaires distribués, les services à exposer, et du côté consommateur, de rechercher (découvrir), parmi ces services exposés, les services qui correspondent le mieux aux besoins énoncés, tout cela d'une façon normalisée.

Typiquement, un client, désirant faire appel à un service Web donné, commencera par une recherche par mot clé dans des annuaires UDDI[1]. Cette recherche résultera en une liste des prestataires habilités à répondre à sa requête. Une fois la réponse reçue (en XML), le client recherchera l'interface WSDL du service qui sied le mieux à son besoin — service fourni par l'un des prestataires considérés. Par la suite, le client examinera la description WSDL du service qu'il aura sélectionné afin de récupérer les informations nécessaires lui permettant de se connecter à son fournisseur et d'interagir avec le code l'implémentant. La connexion et l'interaction avec le service sélectionné se font à l'aide du protocole SOAP.

Le diagramme relaté par la figure 1.1 montre la disposition des standards faisant partie de la pile des spécifications des services Web WS-*, de même que leur connexion avec les concepts SOA énumérés à la section précédente.

Outre les positions occupées par les spécifications de base (i.e., WSDL, SOAP et UDDI), dans le diagramme ci-dessus, on note aussi la présence d'autres spécifications qui *étendent* ces

[1] Un annuaire UDDI se comporte lui aussi comme un service Web dont les méthodes sont appelées via le protocole SOAP.

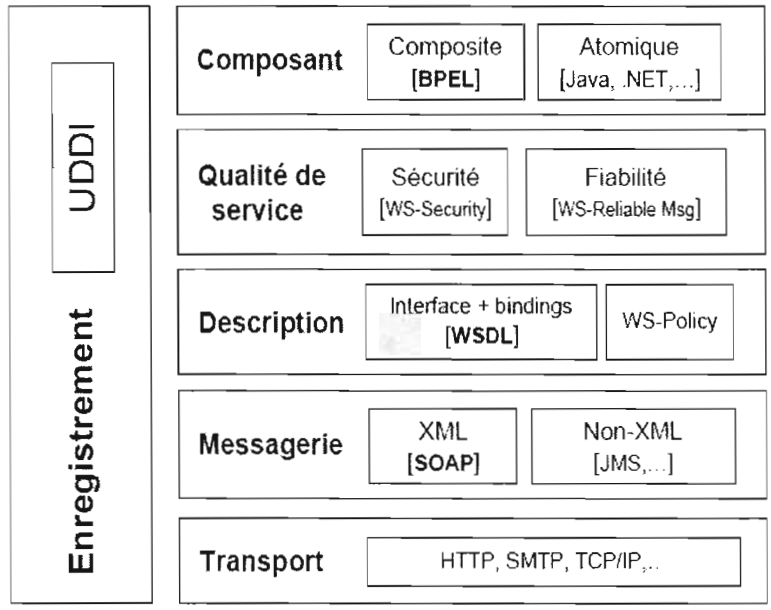


Figure 1.1 Les concepts SOA et la pile des spécifications des services Web WS-*(19).

dernières afin de couvrir tous les concepts dictés par SOA et ainsi permettre de bâtir des architectures qui lui (SOA) sont conformes. Parmi ces spécifications, on note le WS-BPEL et la position clé qu'il occupe. Ce dernier sera décrit dans la section qui suit.

1.4 Le langage BPEL

Comme il est souligné plus haut, si l'on veut profiter des bénéfices promis par SOA, il faut penser à ce que les services (fonctionnalités) exposés soient composés pour former de nouveaux services à granularité plus élevée, et cela jusqu'à arriver à supporter, dans sa globalité, le processus métier de l'entreprise. Cela signifie donc que les processus métier seront définis comme une collection d'activités servant à invoquer des services et à produire des résultats métier (19). Cependant, il faut disposer d'un langage qui, étant *spécialisé* et *standardisé*, permet une telle définition. Un des langages possédant ces propriétés est le langage BPEL que l'on considère comme le langage dominant dans sa catégorie vu sa large adoption dans l'industrie.

Le *Business Process Execution Language* (i.e., *BPEL*) est un langage utilisant une syntaxe XML pour la description de la logique interne des processus métier. Ce langage est basé sur le trio WSDL, XML Schema et XPATH. D'une part, BPEL repose sur le WSDL afin de pouvoir utiliser les services requis pour la composition à définir — composition qui sera elle aussi exposée

comme un service ayant sa propre interface WSDL. D'autre part, BPEL permet le traitement de données basées XML Schema et cela via des expressions XPath.

En 2002, la première version du langage BPEL a été développée par IBM, BEA et Microsoft avant qu'une deuxième version (9) ne voie le jour, en 2003, suite aux modifications et améliorations apportées par SAP et Siebel qui se sont joints aux trois initiateurs plus tard. Actuellement, on en est à la troisième version de ce langage, le BPEL2.0 (1). Au fait, le BPEL a été le fruit de la convergence des deux langages WSFL (25) et XLANG (61). Le premier (WSFL) a été conçu par IBM et vise à décrire les compositions de services sous forme de *graphes* d'activités de nature *orientée* et *acyclique*, alors que le deuxième (XLANG) a été conçu par Microsoft et vise à décrire de telles compositions selon une approche *structurée* basée sur des constructeurs représentant des flots séquentiels et parallèles (19; 72). Résultant de la convergence des deux langages WSFL et XLANG, BPEL combine ainsi les deux approches. À titre d'exemple, avec le BPEL on peut (19) :

- Invoquer séquentiellement ou parallèlement plusieurs opérations de service. L'invocation de ces opérations, qui sont souvent de longue durée, peut être synchrone ou asynchrone.
- Gérer les appels de retour qui ont lieu plus tard.
- Compenser d'une façon sélective les activités qui ont été exécutées dans un processus n'ayant pas pris fin avec succès.
- Corréler les requêtes avec et à travers les processus métiers.
- Structurer un processus métier en plusieurs unités logiques (i.e., *scopes*).
- Traiter les événements qui seraient déclenchés suite à la réception d'un message ou à l'expiration d'un délai.

Le BPEL est un langage de *Workflow* et de composition pour les services Web. Dans ce qui suit, nous mettons d'abord l'emphase sur le côté *Workflow* du BPEL puis sur son côté composition, et cela avant de commencer à décrire d'une façon sommaire sa spécification détaillée.

1.4.1 BPEL et le Workflow

Depuis son adoption, le terme *Workflow* (i.e., flux de travail) continue d'être utilisé dans son domaine d'application traditionnel couvrant, entre autres, la modélisation des processus métier et la gestion de documents. De nos jours, ce même terme est aussi utilisé dans les

domaines émergents des interactions *Business-to-Business* et *Business-to-Consumer* (22).

Dans tous ces domaines, le terme Workflow est utilisé pour décrire l'enchaînement du travail entre différents acteurs. Ces acteurs peuvent être des applications logicielles, locales ou distantes, et même des personnes ayant certaines activités à réaliser. Les spécifications de Workflow peuvent être considérées sous plusieurs perspectives, à savoir (63) :

- ***La perspective flot de contrôle***. Décrit les activités et leur ordre d'exécution à travers différents types de constructeurs. À titre d'exemple, ces constructeurs peuvent exprimer le contrôle d'exécution pour des flots séquentiels, parallèles ou conditionnels. Les activités peuvent être élémentaires ou composées (d'autres activités).
- ***La perspective flot de données***. Met l'emphase sur les données métier (e.g., des documents) et de traitement (e.g., gardées dans des variables locales) qui sont échangées entre activités. Ces données définissent les pré- et post-conditions pour l'exécution des activités.
- ***La perspective des ressources***. Attache au Workflow une structure organisationnelle définie sous forme de rôles (humains ou logiciels) responsables de l'exécution des activités.
- ***La perspective opérationnelle***. Décrit les actions élémentaires exécutées par les activités, de même que des interfaces, de type *activité-à-application*, permettant la manipulation des données entre et à travers des applications.

La perspective du flot de données repose sur celle du flot de contrôle. Quant à cette dernière, elle est considérée comme la perspective la plus pertinente pour juger l'efficacité d'un langage de Workflow (22). Parmi ces langages figure le langage BPEL. Ce dernier permet de définir un Workflow comme étant un processus métier consistant et en un nombre de tâches à accomplir et en une procédure de prise de décision régissant ces dernières, *i.e.*, procédure exprimée comme un ensemble de décisions déterminant l'ordre et la nature de ces tâches (55). Chacune de ces tâches constitue une unité logique de travail qui correspond à une opération de base de services Web.

1.4.2 BPEL et la composition

Suite à l'apparition des services Web et leur adoption, des termes tels que « *Web services composition* » ou « *Web services flow* » ont été utilisés pour décrire la notion de composition de services dans le flot d'un processus (50). Par la suite, ces termes ont été remplacés par les

termes *orchestration* et *chorégraphie*. Ces derniers correspondent à deux façons distinctes pour la description des services composites. Ces deux façons se présentent comme suit (19; 57) :

- ***L'orchestration.*** C'est une façon de composer qui exige à ce que le contrôle et la coordination, des opérations offertes par les services Web impliqués dans la composition à définir, soient centralisés. Ainsi, dans une telle composition, seul un coordinateur central (un processus qui peut être un service Web) se doit connaître les détails de la composition, tandis que les autres services impliqués ne savent pas, et n'ont pas besoin de savoir, s'ils font partie de ladite composition ou pas. Une orchestration est censée être exécutée.
- ***La chorégraphie.*** Toute chorégraphie est le résultat d'un effort de *collaboration* entre plusieurs services Web visant l'accomplissement d'un objectif métier bien précis. Cette deuxième façon de composer ne repose donc pas sur un coordinateur central comme c'est le cas pour l'orchestration, mais demande plutôt à ce que tous les services participants sachent, eux-mêmes, quelle opération exécuter, quel message échanger et quand et avec qui faut-il l'échanger. La chorégraphie est une définition abstraite donnant un aperçu général de la composition à définir.

Le langage BPEL supporte et l'orchestration et la chorégraphie[2], et cela même s'il se trouve que son acronyme contient le mot « *exécution* » — mot qui pourrait donner l'impression que le BPEL n'est fait que pour la variante exécutable de la composition (i.e., l'orchestration). En effet, le BPEL supporte les deux variantes de la composition via sa capacité à formuler les deux types de processus décrits ci-dessous (19) :

- ***Les processus métier exécutables.*** Spécifient les détails d'interactions entre les services qu'ils composent; spécifient les algorithmes exacts contrôlant les activités des compositions à définir, sans oublier de spécifier les messages entrants et sortants échangés par ces activités. La définition de tels processus suit le paradigme de l'orchestration et pourrait être exécutée par un moteur (un engin) BPEL. Dans la majorité des cas, c'est ce type de processus qui est spécifié par le BPEL.

---

[2]Les chorégraphies peuvent être définies selon deux modèles principaux : (a) le *modèle globale* selon lequel un protocole est défini de sorte à décrire, sous une perspective globale, les messages échangés entre toutes les composantes, et (b) le *modèle local* en respect duquel chaque service décrit ses dépendance temporelles et logiques vis-à-vis des messages qu'il échange. BPEL supporte le modèle local, alors qu'un langage comme le WS-CDL (67) supporte le modèle global (6).

- ***Les processus métier abstraits.*** Ne spécifient que les échanges publics entre les composants participants (que ce soit des processus ou de simples services), et leur manque de détails d'interaction les résume à des processus non-exécutables. La définition de tels processus suit le paradigme de la chorégraphie. Le cas d'utilisation le plus commun de ce type de processus correspond à leur usage comme gabarits (*templates*) aidant à la définition de futurs processus exécutables.

Pour le présent travail, nous ne nous intéresserons qu'aux processus exécutables et leur test. De plus, il est à noter que c'est la version 2.0 du BPEL (1) qui sera considérée et non ses versions antérieures.

1.4.3 La spécification WS-BPEL

Tout processus BPEL est décrit par un document XML ayant une structure de base qui est typiquement semblable à celle présentée ci-dessous. La racine d'un tel document correspond toujours à l'élément `<process>`. Cet élément pourrait disposer de plusieurs attributs. Parmi ces attributs, on cite l'attribut `abstractProcess` qui permet de spécifier si le processus est défini comme processus abstrait ou exécutable.

```
<process [attributs]>
    <partnerLinks>. . .</partnerLinks>
    <variables>. . .</variables>
    <correlationSets>. . .</correlationSets>
    <faultHandlers>. . .</faultHandlers>
    <eventHandlers>. . .</eventHandlers >
    { Activity }
</process>
```

Mis à part ses attributs, la racine `<process>` dispose aussi d'un ensemble d'*éléments enfants* (e.g., `<partnerLinks>` et `<variables>`) qui sont suivis du jeton `Activity`. Ces éléments sont optionnels et regroupent des constructeurs qui constituent la partie *déclaration* du processus. Quant au jeton `Activity`, il marque la présence des constructeurs qui constituent la partie *traitement* (Workflow) de ce dernier. Ainsi, dans tout processus, on distingue deux catégories de constructeurs : les constructeurs dédiés à sa partie déclaration, et ceux permettant de définir sa partie traitement. Les paragraphes suivants décrivent les constructeurs des deux catégories.

1.4.3.1 Les constructeurs de déclaration

Ces constructeurs instaurent une déclaration *globale* s'ils sont directement liés au processus. En revanche, si de tels constructeurs sont liés à une des unités logiques composant ce dernier (à un <scope>, voir paragraphe 1.4.3.2), dans ce cas on dit que ces constructeurs instaurent une déclaration *locale*. Dans les deux cas, ces constructeurs se définissent comme suit :

Les partnerLinks. Ils sont la modélisation des services avec lesquels le processus métier interagit. Leurs déclarations spécifient la forme *statique* des relations qu'aura le processus avec ses partenaires (i.e., d'autres processus ou services). Chaque <partnerLink> est typé par un <partnerLinkType>. Ce dernier détermine le rapport conversationnel entre deux services en spécifiant le rôle que joue chacun des deux dans une conversation.

```
<partnerLinks>
    <partnerLink name="ncname"
         partnerLinkType="qname" />+
</partnerLinks>
```

Les variables. Elles permettent de garder trace des données représentant l'état interne du processus. Dans la majorité des cas, ces données correspondent aux messages échangés entre le processus et ses partenaires. Une variable peut être déclarée comme un type de message WSDL (i.e., un messageType), un type XML Schema ou un élément XML Schema.

```
<variables>
    <variable name="ncname" messageType="qname"?
         type=''qname''? element=''qname''?/>+
</variables>
```

Les correlationSets. Ils permettent à ce que les messages envoyés soient livrés non pas juste au bon port de destination, mais aussi à la bonne instance de destination du processus considéré. Pour ce faire, chaque <correlationSet> définit un groupe de *propriétés* permettant d'identifier, d'une façon unique, chaque instance d'un même processus.

```
<correlationSets>
    <correlationSet name="ncname" properties="qname-list"/>+
</correlationSets>
```

Les faultHandlers. Ils permettent de traiter les erreurs qui seraient déclenchées au cours de l'exécution du processus. Généralement, ils sont conçus dans l'optique d'annuler le travail qui n'a pas pu être achevé dans un processus où une erreur a eu lieu.

Les eventHandlers. Ils permettent au processus, ou à une des unités logiques le composant (i.e., un `<scope>`), de pouvoir traiter les événements *normaux* qui seraient déclenchés au cours de son exécution. Ces événements sont de deux types : ceux qui ont lieu suite à la réception d'un message, et ceux qui se manifestent suite au déclenchement d'une alarme.

Plus de détails sur les `<faultHandlers>` et les `<eventHandlers>` sont fournis aux sous-sections 4.6.6 et 4.6.7.

1.4.3.2 Les constructeurs de définition du traitement

Tout processus BPEL dispose d'une seule *activité principale*. Cette activité pourrait correspondre à l'une des activités *structurées* ou *de base* qui sont offertes par le langage BPEL. Ci-dessous, nous exposons l'ensemble de ces activités.

1.4.3.2.1 Les activités de base

Parmi les activités de base les plus importantes, on trouve celles qui sont liées à la réception et à l'envoi des messages, respectivement, de la part et à destination des partenaires. Ces activités sont :

- **L'activité <receive>.** Permet aux partenaires de faire appel à une des opérations exposées par le processus avec lequel ils interagissent. Elle bloque l'exécution séquentielle jusqu'à recevoir un message dont les propriétés concordent avec les valeurs des attributs `portType` et `operation`. Une fois reçu, ce message est sauvegardé par la variable qu'elle spécifie. Il faut aussi noter que cette activité joue un rôle clé dans le cycle de vie d'un processus puisqu'elle permet son instanciation si l'attribut `createInstance` est à `yes`.

```
<receive partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"? createInstance="yes|no"?
</receive>
```

- **L'activité <reply>.** Est utilisée pour l'envoi d'un message en réponse à un premier message ayant été reçu à travers une activité `<receive>` précédemment exécutée sur le

même `partnerLink`, `portType` et `operation`. Le résultat d'un `<reply>` peut prendre deux formes : la première forme correspond au cas où le `<reply>` résulte en une réponse normale qui sera portée par la variable qu'il spécifie, tandis que la deuxième correspond au cas où ce dernier résulte en une erreur dont le nom sera spécifié par l'attribut `faultName`.

```
<reply partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"? faultName="qname"?
</reply>
```

- **L'activité <invoke>**. Permet l'invocation d'une opération exposée par un service Web. Cette invocation peut être *synchrone* (sous forme d'une requête/réponse) ou *asynchrone* (sous forme d'une requête à sens unique). La déclaration d'une invocation synchrone requiert la présence et d'une `inputVariable` spécifiant le message à envoyer et d'une `outputVariable` pour la sauvegarde du message à recevoir. Contrairement à cela, la déclaration d'une invocation asynchrone n'exige la présence que d'une `inputVariable`. Une invocation synchrone peut aussi résulter en un message reportant une erreur.

```
<invoke partnerLink="ncname" portType="qname"
    operation="ncname" inputVariable="ncname"? outputVariable="ncname"?
</invoke>
```

En plus des trois activités décrites ci-dessus, le langage BPEL offre d'autres activités de base que nous présentons comme suit :

- **L'activité <assign>**. Permet de copier des données d'une variables à une autre, de même que de construire de nouvelles données à base d'expressions, *e.g.*, expressions XPath.
- **L'activité <throw>**. Permet de signaler une erreur interne d'une façon explicite.
- **L'activité <wait>**. Indique soit une *durée* soit une *date limite* pendant/avant laquelle l'exécution du processus est bloquée.
- **L'activité <empty>**. Indique au processus de ne rien faire.
- **L'activité <exit>**. Permet de mettre fin à l'exécution d'une instance d'un processus, d'une façon immédiate.
- **L'activité <rethrow>**. Permet de lever (à une autre portée) une erreur ayant été initialement capturée par le `<faultHandler>` englobant.

1.4.3.2.2 Les activités structurées

Les activités structurées dotent le langage BPEL de cette capacité à exprimer des compositions de services selon une approche structurée; approche héritée du langage XLANG. Ci-dessous, nous énumérons l'ensemble de ces activités :

- **L'activité <sequence>**. Contient (une à) plusieurs activités, structurées ou de base, qui devront être exécutées en séquence, suivant l'ordre selon lequel elles sont listées.
- **L'activité <if>**. Permet de sélectionner, à partir d'une liste ordonnée de choix, une et une seule activité à exécuter.
- **L'activité <pick>**. Attend l'occurrence d'un événement parmi plusieurs, pour n'activer qu'une seule branche parmi plusieurs, et par suite n'exécuter qu'une activité parmi plusieurs.
- **L'activité <while>**. Permet une exécution répétitive de l'activité qu'elle enveloppe, et cela tant que la condition qu'elle définit est maintenue à *vrai*.
- **L'activité <repeatUntil>**. Permet une exécution répétitive de l'activité qu'elle enveloppe, et cela tant que la condition qu'elle définit est maintenue à *faux*.
- **L'activité <forEach>**. Permet d'exécuter, d'une façon *parallèle* ou *séquentielle*, un certain nombre d'instances de l'activité `<scope>` qu'elle enveloppe.
- **L'activité <flow>**. Permet à ce que toutes les activités qui y sont directement enchâssées soient concurremment exécutées. À travers des liens de contrôle (i.e., des `<links>`) qu'elle déclare, elle arrive à définir des dépendances de contrôle entre les activités qu'elle enchâsse.
- **L'activité <scope>**. Définit un contexte d'exécution complet pour l'activité qu'elle enveloppe. Ce contexte a une structure similaire à celle d'un processus vu dans sa globalité. Outre les constructeurs de déclaration décrits au paragraphe 1.4.3.1, un `<scope>` pourrait aussi définir des constructeurs `<compensationHandlers>` et `<terminationHandlers>`. Les premiers permettent de défaire le travail exécuté par le `<scope>` auxquels ils sont attachés, alors que les deuxièmes permettent à ce dernier de contrôler, à un certain niveau, la sémantique de sa terminaison forcée.

Plus de détails, sur chacune des activités listées, seront fournis dans le cadre du chapitre portant sur la traduction du BPEL en **B**-CFG (*BPEL Control Flow graph*).

1.4.3.3 Les liens de contrôle et le Dead Path Elimination

Dans la sous-section précédente, nous avons vu que les activités qui sont directement enchâssées dans un `<flow>` s'exécutent concurremment. Nous avons aussi souligné qu'il est possible de définir des dépendances de contrôle entre ces activités à travers des liens de contrôle (i.e., des `<links>`). Chacun de ces liens définit une connexion dirigée entre une *activité source* et une *activité cible* (30). Une activité peut être la source de plusieurs liens *sortants* et/ou la cible de plusieurs liens *entrants*.

Dans le cas d'une exécution réussie d'une activité source, les états de ses liens de contrôle sortants sont déterminés dépendamment de leurs *transition Conditions*. Ces dernières sont des expressions booléennes retournant *vrai* ou *faux*. De son côté, une activité cible attend à ce que l'état, de chacun de ses liens de contrôle entrants, soit connu pour qu'elle puisse évaluer sa *join Condition*[3]. Cette dernière est une expression booléenne qui, dans le cas où elle est évaluée à vrai, donne le feu vert à cette activité cible pour s'exécuter. Cependant, si cette expression est évaluée à faux, l'activité cible en question ne sera pas exécutée, de même que tous ses liens de contrôle sortants (si elle en a) seront mis à faux. Ces derniers sont mis à telle valeur pour maintenir les dépendances de contrôle et éviter à ce que les activités, qu'ils ciblent, attendent indéfiniment l'évaluation de leurs états (à vrai ou à faux). Ce concept (ou processus) est appelé *Dead Path Elimination*(1), *i.e.*, DPE.

Afin d'illuster le concept du DPE, nous considérons le comportement concurrentiel décrit par le code exemple de la figure 1.2 — exemple tiré du travail de Lohmann (30). Selon le résultat d'évaluation de la condition définie par l'activité `<if>`, deux scénarios d'exécution sont possibles :

1. Si cette condition est évaluée à vrai, l'activité B sera exécutée après la fin d'exécution de l'activité A et l'évaluation positive du lien $AtoB$. Suite à la fin de son exécution, l'activité B évalue positivement son lien sortant $BtoC$ pour finalement permettre aux activités C et D d'être exécutées séquentiellement.
2. Si cette condition est évaluée à faux, l'activité E sera exécutée, tandis que, en respect du DPE, l'activité B sera sautée après avoir attendu jusqu'à ce que le lien $AtoB$ soit évalué suite à l'exécution de son activité source A. Ensuite, et toujours en respect du DPE, le

[3]Contraitement aux *transition Conditions* qui peuvent être exprimées à base de valeurs de variables arbitraires, les *join Conditions* ne peuvent être définies qu'à base des états des liens de contrôle entrants.

lien *BtoC* sortant de l'activité *B* est mis à faux. Par conséquent, l'action[4] de l'activité *C* ne sera pas exécutée, alors que celle de l'activité *D* le sera.

```
<flow>
  <links> <link name="AtoB"/> <link name="BtoC"/> </links>
  <activity name="A">
    <sources> <source linkName="AtoB"/> </sources>
  </activity>
  <if>
    <condition>...</condition>
    <activity name="B">
      <targets> <target linkName="AtoB"/> </targets>
      <sources> <source linkName="BtoC"/> </sources>
    </activity>
    <else> <activity name="E"/> </else>
  </if>
  <sequence>
    <activity name="C">
      <targets> <target linkName="BtoC"/> </targets>
    </activity>
    <activity name="D"/>
  </sequence>
</flow>
```

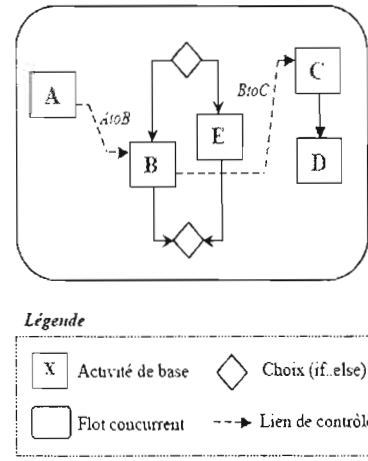


Figure 1.2 Exemple de code BPEL illustrant le concept de DPE.

1.4.4 Les serveurs d'exécution

Pour l'exécution des processus métier qui sont élaborés à base du BPEL, il faut disposer d'un serveur d'exécution adéquat pour cette fin. Un tel serveur doit définir un environnement d'exécution incluant ce qu'on appelle un *moteur* ou un *engin* d'exécution BPEL. Parmi les plus importants moteurs d'exécution BPEL, nous pouvons citer : le moteur *activeBPEL Engine* d'activeEndpoints (14), ainsi que le moteur *Collaxa* qui a été greffé au serveur d'application *10g* d'Oracle pour donner naissance au serveur *Oracle BPEL Process Manager* (47).

À propos du choix d'implémentation de certains points et règles, la spécification du BPEL ne s'est pas prononcée et a laissé libre-choix aux constructeurs des moteurs pour une prise en charge personnalisée de tels points et règles. Il est donc fort probable que l'implémentation du BPEL diffère d'un moteur à un autre. Pour avoir une idée sur le type de choix d'implémentation qu'on a pu envisager pour ces points et règles, nous avons étudié certain(e)s de ces dernier(e)s pour le moteur *activeBPEL Engine* d'activeEndpoints — voir annexe A.

---

[4] L'action d'une activité de base est atomique et en constitue le coeur, alors que celle d'une activité structurée consiste en l'exécution des activités de base ou structurées qui y sont enchassées.

1.4.5 Exemple

Comme exemple de processus, ci-dessous nous présentons le processus *loan Approval* qui est étudié à la section 6.3. Ce processus est une variante de celui évoqué par la norme BPEL (1). Il reçoit de la part de ses clients des requêtes de demande de prêts via l'activité *ReceiveLoanRequest*. Dépendamment du montant du prêt demandé et du risque que présente ce dernier, le processus fait appel à un service d'évaluation de risque (i.e., *assessor service*) ou d'approbation (i.e., *approval service*), sinon aux deux successivement. La partie *traitement* de ce processus est définie par une activité principale `<flow>`. Cette dernière englobe des activités de (r)envoi, de réception et d'affectation, en plus des six liens de contrôle liant lesdites activités.

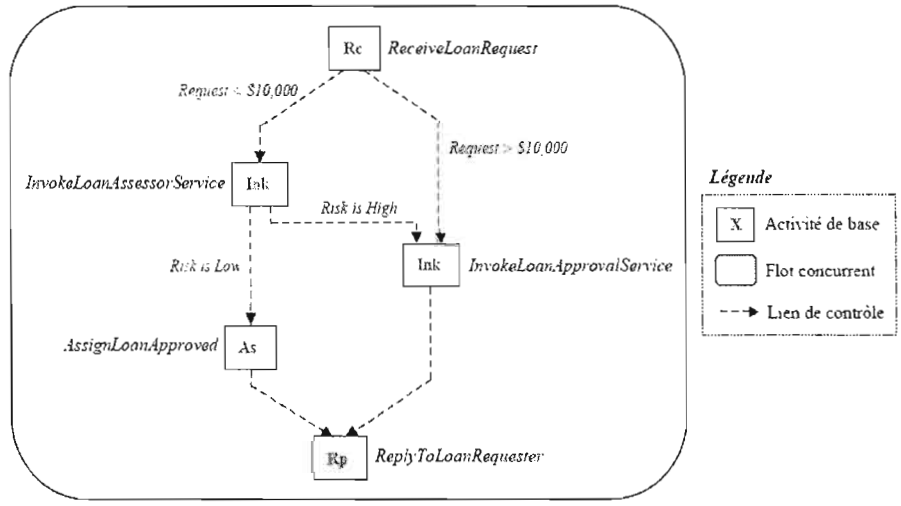


Figure 1.3 Un exemple de processus BPEL : le processus *loan Approval*.

La figure 1.3 est une représentation graphique du processus *loan Approval*. Cette représentation met en lumière l'autre approche (outre l'approche structurée) adoptée par le BPEL pour la composition de service, c'est-à-dire l'approche décrivant une telle composition sous forme d'un graphe d'activités orienté et acyclique.

1.5 Conclusion

À travers ce chapitre, nous avons donné une description sommaire de l'architecture SOA et des concepts qui en découlent, de même qu'une présentation du langage BPEL et de la place qu'il occupe dans cette architecture. Dans le chapitre suivant, nous présentons le test logiciel ainsi que la relation qui lie certaines de ses variantes à la théorie des graphes.

CHAPITRE II

LE TEST LOGICIEL

2.1 Introduction

Outre les phases de conceptualisation et de réalisation, l'activité de mise en oeuvre d'un programme informatique (un logiciel), comme toute autre activité de fabrication humaine (e.g., fabrication de voitures), nécessite l'entreprise d'une phase vitale qui est la *phase de test* (75). Cette phase, qui peut être manuelle ou automatique, a pour objectif de s'assurer que les résultats produits par le programme réalisé ne font pas fausse note par rapport à ceux qui sont attendus.

D'une façon plus concise, Myers(43) définit le test logiciel comme étant « *le processus consistant à exécuter un programme dans l'optique de détecter des anomalies* ». Ainsi, le test logiciel vise uniquement à prouver l'*existence* de défauts causant des anomalies lors de l'exécution du code à tester, et ne peut prouver leur *absence* dans ce code(10). De même, le test logiciel n'a pour objectif ni de localiser l'emplacement des défauts dans le code à tester, ni de corriger ces derniers.

2.2 Les classes de test

2.2.1 Vue d'ensemble

Il existe plusieurs variantes du test logiciel. Ces variantes peuvent être classées selon différentes perspectives. L'une de ces perspectives des plus usuelles est celle classant lesdites variantes selon l'étendue et la nature de l'entité testée, en adéquation avec la phase atteinte dans le cycle de développement. Ainsi, parmi les classes de tests (on parle aussi de niveaux de tests) attachées à cette première perspective, on peut citer :

- ***Les tests unitaires.*** Cette classe de tests consiste à tester les modules individuels de

l'application, *un par un*. À titre d'exemple, un module peut être une routine, une classe ou un service, et cela dépendemment du type de l'application développée.

- ***Les tests d'intégration.*** Cette classe de tests consiste à tester les modules d'une application qui ont été intégérés (ou combinés) ensemble. Elle vise à « valider l'intégration des modules, d'abord entre eux puis dans leur environnement d'exploitation définitif » (53).
- ***Les tests système.*** Cette classe de tests consiste à tester le système final (logiciel, matériel, documentation) dans son intégralité. Elle vise à « valider les spécifications techniques détaillées et les exigences fonctionnelles » (53).

La deuxième perspective classe les variantes de test selon le niveau de connaissance de la structure interne de l'entité testée (on parle aussi de niveau d'accessibilité). En effet, les classes de tests vues sous cette deuxième perspective sont au nombre de trois, à savoir :

- ***Les tests boîte noire*** *(i.e., tests fonctionnels).* Cette classe de tests considère le logiciel à tester comme une *boîte noire*. Dans ce cas, le testeur, n'ayant pas accès à (ou se passant de) la structure interne du logiciel, génère ses données de test en se basant sur les spécifications et cela dans le dessein de tester la *fonction* dudit logiciel.
- ***Les tests boîte blanche*** *(i.e., tests structurels).* Cette classe de tests donne au testeur le droit d'accès à la structure interne (au code source) du logiciel à tester et lui permet, entre autres, de générer ses données de test en se basant sur cette structure.
- ***Les tests boîte grise.*** Cette classe de tests est la combinaison des deux qui précèdent. Selon Rozenberg (53), « les tests boîte grise sont définis en deux phases distinctes : une *phase d'analyse* de type boîte blanche et une *phase de réalisation* de type boîte noire »[1].

Finalement, les variantes de test peuvent aussi être classées selon une troisième perspective qui repose sur le type des propriétés à tester pour un système donné — des propriétés comme la *robustesse*, la *performance* et la *vulnérabilité*. Néanmoins, on dénombre certaines variantes de test qu'aucune des trois perspectives évoquées ne permet de classer, comme les *tests de régression*. Ces derniers consistent à tester un logiciel, ayant été mis à jour, et cela en réutilisant les données de test qui ont été appliquées sur sa version précédente. Le but est de vérifier si aucune fonction du logiciel n'a été affectée suite à la mise à jour menée.

[1] « En pratique, le déroulement du test boîte grise se fait en deux étapes. En premier, les tests fonctionnels sont passés et le taux de couverture de la structure interne du logiciel testé est calculé. Après, seuls les tests nécessaires pour compléter cette couverture sont alors conçu et exécutés » (75).

Étant donné que notre travail entre dans le cadre des tests unitaires et boîte blanche, plus de détails sur ces derniers sont fournis dans ce qui suit.

2.2.2 Les tests unitaires

Comme nous l'avons souligné plus haut, les tests unitaires consistent à tester les modules individuels de l'application, *un par un*. Or, il se trouve qu'un module d'une application n'est souvent pas conçu pour fonctionner seul mais pour interagir avec d'autres. Ainsi, pour pouvoir tester un module donné, il est d'usage de procéder à la simulation des modules avec lesquels il interagit. En effet, les tests unitaires s'épaulent sur les deux concepts de *lanceurs de test* (i.e., *drivers*) et de *bouchons* (i.e., *stubs*); les premiers sont de petits programmes simulant les modules *appelants* du module testé, alors que les deuxièmes sont de petits programmes simulant ses modules *appelés*.

Le choix entre l'usage des lanceurs, des bouchons ou des deux à la fois, dépend de l'approche adoptée pour l'élaboration des tests unitaires. Les principales approches que ces tests pourraient adopter sont au nombre de trois et se présentent comme suit (53) :

1. ***L'approche descendante*** *(i.e., top/down).* Elle préconise à ce que le test de chaque module soit fait indépendamment des modules appelants, et cela en partant du haut de la hiérarchie[2]. En revanche, chaque module appelé sera simulé par un bouchon.
2. ***L'approche ascendante*** *(i.e., bottom/up).* Contrairement à la première, cette deuxième approche préconise à ce que les modules élémentaires soient testés avant que l'on passe au test de leurs congénères supérieurs dans la hiérarchie. Dans ce cas, on a recours à l'usage des lanceurs, et non plus aux bouchons.
3. ***L'approche modules isolés.*** Cette approche combine les deux qui précèdent. Elle préconise à ce que chaque module soit testé *en isolation* des autres. Ainsi, chaque module appelant sera simulé par un lanceur, et chaque module appelé par un bouchon.

Dans le présent travail, les tests unitaires adoptent l'approche *modules isolées*. Cela dit, dans le cas où l'on souhaite effectuer *simultanément* des tests unitaires et des tests d'intégration, il vaudrait mieux opter pour l'une des deux premières approches.

---

[2]Une hiérarchie qui correspond à un diagramme où les éventuels modules appelants du module à tester se trouvent en son dessus, alors que ses éventuels modules appelés se trouvent en son dessous.

Toutes approches confondues, les tests unitaires visent à améliorer et à valider la qualité du code, ainsi qu'à réduire les temps de développement. En effet, ce type de tests se distingue par sa détection précoce de la présence d'éventuels défauts dans l'application testée. Une telle détection rend *efficaces* et *économiques* la prévention et la correction des anomalies; un défaut détecté au cours des tests unitaires coûte **10** fois moins cher qu'un qui serait détecté lors des tests d'intégration, et **100** fois moins cher qu'un autre qui serait détecté lors des tests de recette[3] (53).

Le test unitaire est la variante de test la plus automatisée et la plus apte à l'être. Dans la littérature, on trouve plusieurs outils qui rendent cette automatisation possible. Ces outils sont communément appelés des *cadres de test* (i.e., *test frameworks*). La plus importante famille de ces cadres est la famille *xUnit* — le cadre *SUnit* étant le premier à avoir vu le jour, alors que *JUnit* est connu être le plus populaire de cette famille.

En particulier, tous les *frameworks* de la famille xUnit font appel aux concepts de *cas de test* et *suite de test*. Un cas de test sert pour enveloppe à la *logique de test*, *i.e.*, aux « instructions d'invocation de code dans le module à tester et de vérification des résultats » (37). Quant à une suite de test, elle est l'agrégation de plusieurs cas de test, et a pour vocation de permettre l'exécution de ces derniers, d'un trait.

Comme c'est le cas pour le présent travail, les tests unitaires sont généralement des tests orientés boîte blanche. La section suivante met la lumière sur les techniques associées à cette orientation, *i.e.*, les techniques de tests boîte blanche.

2.2.3 Les tests boîte blanche

Comme nous l'avons indiqué plus haut, les tests boîte blanche se caractérisent, entre autres, par la sélection des données de test à partir de la structure interne du programme à tester. La description d'une telle structure peut être basée soit sur le *flot de contrôle* soit sur le *flot de données*. Dans les des deux cas, on dénombre une panoplie de techniques encadrant le déroulement des tests boîte blanche à réaliser. Chacune de ces techniques a pour objectif de proposer un *critère de couverture* donné. Chaque critère correspond à une méthode de sélection qui nous fournit un ensemble fini de données de test permettant la couverture des sous-structures visées dans le programme à tester.

[3] « Ces tests sont exécutés par une équipe dédiée représentant la maîtrise d'ouvrage. Il s'agit de répéter les tests système et d'installation afin de se prononcer sur la qualité du développement »(53).

Dans le présent travail, seules les techniques de test basées sur le flot de contrôle sont considérées. Ainsi, dans ce qui suit, nous ne prenons note que des critères de couverture proposés par ces techniques. Parmi ces critères, on cite :

- **Le critère** « ***toutes-les-instructions*** ». Exige la sélection des données de test qui permettront l'exécution, au moins une fois, de chaque instruction du programme testé.
- **Le critère** « ***toutes-les-décisions*** ». Exige la sélection des données de test qui mèneront à la l'exploration de toutes les décisions dans le programme testé, et cela de façon à ce que leur valeur de vérité soit au moins une fois *vraie* et au moins une fois *fausse*.
- **Le critère** « ***chemins de base*** ». Exige la sélection des données de test qui permettront à ce qu'un nombre de scénarios d'exécution particuliers, à travers le programme testé, ait lieu. Ce nombre doit être égal au nombre de *McCabe* qu'on définit dans ce qui suit.
- **Le critère** « ***tous-les-scénarios*** ». Exige la sélection des données de test permettant à ce que tous les scénarios, qu'on peut imaginer à travers le programme testé, aient lieu.

Généralement, pour l'étude de ces critères, on se réfère à la *théorie des graphes*. Au fait, comme nous le dévoilerons dans la section suivante, pour l'étude de la structure interne d'un programme et des critères de sa couverture, on le représente sous forme d'un graphe. Ce dernier se définit comme le mariage de deux ensembles : un premier ensemble regroupant ses noeuds et un deuxième regroupant ses arcs. À travers un graphe, on distingue des chemins reliant son noeud d'entrée à son noeud de sortie; chacun de ces chemins représente un scénario à travers le programme représenté. Procédant à base d'une telle représentation, les critères cités ci-dessus deviennent les équivalents respectifs des critères de couverture : « *tous-les-noeuds* », « *tous-les-arcs* », « *tous-les-chemins-indépendants* » et « *tous-les-chemins* ». L'ordre dans lequel ces critères sont cités correspond à celui les classant du plus faible au plus fort; un critère est *plus fort* qu'un autre s'il est capable de mener à la détection de plus de défauts que son rival.

2.3 Le test et la théorie des graphes

Dans la présente section, nous expliquons comment on utilise la théorie des graphes pour mener une étude simple, mais formelle, de la structure interne d'un programme — structure sur laquelle les tests boîte blanche se basent pour la production des données de test selon un critère de couverture donné. L'exploitation de cette théorie se traduit par la représentation de ladite structure sous le format d'un graphe. Ce dernier est « un symbolisme mathématique particulièrement riche »(74). Ses propriétés seront elles aussi présentées dans la suite.

2.3.1 Généralités

Ci-dessous, sont données les définitions de base se rapportant à la notion de graphe :

Définition 1 Un graphe orienté G se compose d'un ensemble de noeuds N et d'un ensemble d'arcs E.

Définition 2 Le nombre d'arcs sortant d'un noeud x est appelé *degré extérieur* de ce noeud et est symbolisé par $d^+(x)$. De même, on définit le *degré intérieur* $d^-(x)$ d'un noeud x qui est égal au nombre d'arcs qui y entrent.

Définition 3 Un graphe est considéré comme *fortement connexe* lorsque, à partir de n'importe quel noeud, il est possible d'aller vers n'importe quel autre noeud (74).

Définition 4 Un *chemin* est une séquence finie et alternée de noeuds et d'arcs, débutant et finissant par des noeuds, telle que chaque arc est sortant d'un noeud et incident au noeud suivant dans la séquence (35).

Définition 5 Un *circuit* est un chemin dont les extrémités coïncident (35).

Dans le reste du mémoire, nous représentons un circuit (ou un chemin) en énumérant les arcs ou les noeuds en faisant partie. À titre d'exemple, dans le graphe G_1 de la figure 2.1, nous trouvons les circuits $\beta_1 = [u_1, u_2] = [c, a, c]$ et $\beta_2 = [u_4, u_5] = [b, c, b]$. Il est aussi à noter que tout circuit (et tout chemin) peut être représenté comme un *vecteur* où l'on marque la présence des arcs le constituant par "1", alors que l'absence du reste des arcs du graphe est marqué par "0". En l'occurrence, les arcs β_1 et β_2 peuvent être représentés comme suit :

	u_1	u_2	u_3	u_4	u_5
β_1	1	1	0	0	0
β_2	0	0	0	1	1

Par ailleurs, on constate qu'aucun des deux circuits $\beta_1 = [u_1, u_2] = (1,1,0,0,0)$ et $\beta_2 = [u_4, u_5] = (0,0,0,1,1)$ ne peut être exprimé en fonction de l'autre, ou proprement dit ne peut être une *combinaison linéaire* de l'autre. Dans ce cas, on dit que les deux circuits β_1 et β_2 sont *indépendants*. Par contre, le circuit $\mu = [u1, u2, u5, u4] = (1,1,0,1,1)$ peut être exprimé en fonction des circuits β_1 et β_2 : $\mu = \beta_1 + \beta_2$. Dans ce cas, on dit que le circuit μ est une *combinaison linéaire* des deux circuits indépendants β_1 et β_2.

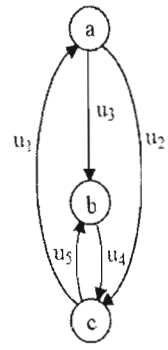


Figure 2.1 Graphe G_1.

Tout programme séquentiel, qu'il soit structuré ou non, peut être représenté par un graphe qu'on appelle communément *graphe de contrôle*. Ce dernier appartient à la classe particulière des graphes associés au test des programmes. Formellement :

Définition 6 « Un *graphe de contrôle* est un graphe qui n'a qu'un *seul* noeud d'***entrée*** et qu'un *seul* noeud de ***sortie***. Un nœud s est considéré comme nœud d'*entrée* d'un graphe si en entrant par celui-ci, nous pouvons visiter tous les autres, alors qu'un noeud f est considéré comme noeud de *sortie* d'un graphe, si quel que soit le noeud k de ce graphe, nous pouvons trouver un chemin qui nous conduit de k à f » (74).

Pour la construction d'un graphe de contrôle à partir d'un code source donné (74) :

- Chaque instruction est représenté par un noeud.
- Un arc est défini entre deux noeuds lorsqu'il se trouve qu'une exécution séquentielle entre ces noeuds est possible.
- Un noeud d'entrée (*resp.*, de sortie) fictif est ajouté pour marquer le début (*resp.*, la fin) du graphe.
- Chaque décision que le programme inclut est représentée par un noeud de décision — noeud dont sortira autant d'arcs qu'il y aurait de cheminements possibles à suivre, sur ce point, par le flot d'exécution du programme.

2.3.2 Propriété de la représentation vectorielle

Dans ce paragraphe, nous exposons une propriété qu'on associe à la représentation vectorielle des circuits d'un graphe. Cette propriété nous sera utile pour la re-définition du nombre

de McCabe au profit des programmes concurrents.

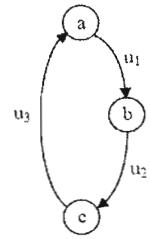


Figure 2.2 Graphe G_2 (74).

Dans le graphe G_2 de la figure 2.2 donnée ci-dessus, on considère les deux circuits μ_1 = [a, b, c, a] et μ_2 = [c, a, b, c]. Nous constatons que μ_1 = [u1, u2, u3] = (1,1,1) de même que μ_2 = [u3, u1, u2] = (1,1,1), donc *vectoriellement* parlant, les deux circuits μ_1 et μ_2 seraient égaux, alors qu'en fait, pour chacun des deux circuits, l'ordre selon lequel les nœuds ont été parcourus n'est pas le même. Cela est dû au fait que la représentation vectorielle est trop « ***grossière*** » et ne tient pas compte de l'ordre dans lequel les nœuds sont visités (74).

2.3.3 Le nombre de McCabe

Dans ce dernier paragraphe, nous rappelons les définitions et le théorème dont McCabe s'est servis comme jalon pour tracer l'alignement qui l'a mené à la définition de sa fameuse *complexité cyclomatique* pour les programmes séquentiels, *i.e.*, le *nombre de McCabe*. Après l'avoir défini, différentes façons de calculer ce nombre seront fournies.

2.3.3.1 Fondement

Définition 7 Le *nombre cyclomatique* d'un graphe G formé de n nœuds et a arcs est : V(G) = a − n + 1.

Théorème 1 Pour un graphe fortement connexe, le nombre cyclomatique est égal au nombre maximal des circuits *linéairement indépendants* (3){page 29}.

Définition 8 « Une *base de circuits* est un ensemble *minimal* de circuits indépendants tel que tout vecteur représentatif d'un circuit puisse s'exprimer comme combinaison linéaire des circuits de la base. La dimension de cette base est égale au nombre cyclomatique »(35).

À titre d'exemple, le nombre cyclomatique du graphe G_1 dépeint par la figure 2.1 vaut :

$V(G_1) = 5 - 3 + 1 = 3$. Dans ce cas, et d'après le théorème 1, le nombre maximal des circuits indépendants qu'on peut définir pour le graphe G_1 est égal à 3. Afin de définir une base de circuits pour le graphe G_1, on peut alors, en sus des deux circuits $\beta_1 = [u_1, u_2]$ et $\beta_2 = [u_4, u_5]$, considérer un troisième circuit $\beta_3 = [u1, u3, u4] = (1,0,1,1,0)$. Ce circuit est indépendant de β_1 et β_2 car il n'existe aucune combinaison linéaire de β_1 et β_2 qui pourrait exprimer β_3.

2.3.3.2 Définition

Comme nous l'avons souligné plus haut, tout programme séquentiel, qu'il soit structuré ou non, peut être représenté par un graphe de contrôle. En revanche, un tel graphe n'est pas de nature fortement connexe. Par conséquent, la propriété de l'existence d'une base indépendante n'est pas valable. Pour remédier à cela, McCabe propose, dans son article fondateur (39), l'ajout d'un arc de retour reliant le nœud de sortie du graphe à son nœud d'entrée. De cette façon, et de par la définition même de son nœud d'entrée et de sortie, tout graphe de contrôle calqué d'un programme séquentiel est rendu fortement connexe. Comme résultat, on peut dresser la définition suivante :

Définition 9 « Le *nombre de McCabe* d'un graphe de contrôle G correspond au nombre cyclomatique d'un graphe fortement connexe G' résultant de l'adjonction d'un arc au graphe G (allant de la sortie vers l'entrée) »(74).

Le nombre de McCabe *V(G)* d'un graphe de contrôle G formé de $\boldsymbol{n}$ nœuds et $\boldsymbol{a}$ arcs est donc : $V(G) = \boldsymbol{a} - \boldsymbol{n} + 2$.

2.3.3.3 Simplication du calcul

Afin de rendre le calcul de ce nombre plus facile, on a fait appel au théorème de *Mills* (39) pour offrir une alternative de calcul beaucoup plus simple. En effet, ledit théorème prouve que le nombre cyclomatique peut être défini comme égal au nombre π des *décisions simples* qui apparaissent dans un programme, augmenté d'une unité : $V(G) = \pi + 1$.

En définitive, nous déduisons que le nombre cyclomatique n'est fonction que du nombre de nœuds de décision figurant dans un graphe et non pas de sa taille (i.e., non pas du nombre d'arcs et de noeuds). Au fait, si l'on dote un graphe d'un nouveau nœud de décision δ (typiquement un tel nœud a deux arcs sortants et un seul entrant), on augmentera ainsi le nombre cyclomatique de : (3 arcs) – (1 nœud de décision) = 2 arcs. Proprement dit, l'augmentation du nombre

cyclomatique a été dû à l'*excédant* en arcs sortants ($d^+(\delta) - 1$) du nœud δ ajouté (74). Nous généralisons cela dans le cadre d'un corollaire que nous présentons comme suit :

Corollaire 1 Le nombre cyclomatique d'un graphe de contrôle G, calqué d'un programme séquentiel, est fonction de l'***excédant*** en arcs sortants engendré par chacun des nœuds de décision faisant partie de ce graphe. Plus précisément : $V(G) = \sum_{i=1}^{k}(d^+(\delta_i) - 1) = (\sum_{i=1}^{k} d^+(\delta_i)) - k$, avec k est le nombre des nœuds de décision δ présents dans le graphe G.

Combiné à la propriété de la représentation vectorielle décrite dans le paragraphe précédent, ce corollaire nous aidera à la re-définition du nombre de McCabe pour les programmes concurrents.

Une deuxième et dernière alternative de calcul du nombre de McCabe est celle basée sur l'*inspection visuelle* du graphe de contrôle. À la base de cette alternative de calcul, on évoque le théorème d'*Euler* (39). Ce théorème énonce une formule qui relie le nombre de sommets $\boldsymbol{n}$, de côtés $\boldsymbol{e}$, et de faces $\boldsymbol{f}$ d'un polyèdre, qui est une forme géométrique non plane. Cette formule s'écrit : $\boldsymbol{n} - \boldsymbol{e} + \boldsymbol{f} = 2$. Pour l'application de cette dernière au graphe de contrôle, qui est un graphe *planaire*, on considère plutôt ce que l'on a appelé les $\boldsymbol{r}$ régions figurant dans le plan découpé par un tel graphe — régions qui sont analogues aux f faces faisant partie d'un polyèdre. D'où le constat : $V(G) = e - \boldsymbol{n} + 2 = r$.

2.4 Conclusion

Dans ce deuxième chapitre, nous avons présenté le test logiciel et certaines de ses variantes, et cela tout en mettant l'emphase sur les deux variantes : test unitaire et test boîte blanche. En outre, nous avons aussi abordé la relation particulière qui lie ces deux variantes de test à la théorie des graphes. À travers le chapitre qui suit, nous recenserons et présenterons les travaux qui ont abordé cette question de la génération des cas de test pour ensuite présenter notre travail d'une façon sommaire.

CHAPITRE III

TEST DES PROCESSUS BPEL : LES TRAVAUX EXISTANTS ET LE NÔTRE

3.1 Introduction

Le test logiciel appliqué à la composition de services a été le sujet de plusieurs travaux de recherche. Ces travaux réutilisent, et si besoin est, adaptent les variantes de test, existantes pour les applications logicielles traditionnelles, de façon à pouvoir les appliquer pour des compositions de services. La majorité de ces travaux s'est particulièrement intéressée aux variantes de test *unitaire* (e.g., Mayer et Lübke (38)), d'*intégration* (e.g., Bertolino et Polini (5)) et de *régression* (e.g., Liu et *al.* (29)).

Vu que notre travail adopte la variante de test unitaire pour l'appliquer aux compositions de service décrites en BPEL, une attention particulière sera donc prêtée aux travaux qui ont porté sur l'application de cette variante pour les compositions de service. Néanmoins, pour le test de telles compositions, il faut d'emblée noter qu'une variante de test ne s'applique pas de la même façon pour une composition de type chorégraphie que pour une de type orchestration. En particulier, la variante du test unitaire, à laquelle nous nous intéressons, s'applique différemment à ces deux types de composition puisque (6) :

- Pour le cas d'une **composition de type orchestration**, l'unité de test est le processus orchestrateur de cette composition. Cet orchestrateur pourrait être testé soit selon une approche orientée boîte blanche (i.e. basée sur l'implémentation), soit selon une approche orientée boîte noire (i.e., basée sur les spécifications).
- Pour le cas d'une **composition de type chorégraphie**, l'unité de test est un partenaire particulier de la chorégraphie définie. Un tel partenaire est testé dans le but de vérifier s'il

suit un patron d'interaction qui est conforme au protocole spécifié par la chorégraphie en question. Cela revient donc à exercer un type de test orienté boîte noire que l'on appelle communément *test de conformité* (i.e., *conformance testing*).

Pour notre travail, nous nous sommes plutôt penchés sur les tests unitaires appliqués aux compositions de type orchestration (i.e., processus BPEL exécutables), et plus précisément sur ceux qui sont orientés boîte blanche. S'inscrivant dans le cadre de ce type de tests, notre travail concernera beaucoup plus la *génération* des cas de test que leur *exécution*.

À travers les années, plusieurs approches ont été proposées pour la génération des cas de test. En particulier, pour une génération basée sur le code (i.e., *program-based generation*), on dénombre principalement trois approches que l'on présente comme suit (12) :

- ***L'approche aléatoire.*** Est considéré comme l'approche de génération la plus simple. Elle pourrait être utilisée pour générer des données de test de tout type. Cependant, on lui reproche sa faible performance en termes de couverture. C'est d'ailleurs pour cela qu'elle est souvent utilisée comme point de référence (i.e., *benchmark*) pour l'évaluation des autres approches.
- ***L'approche orientée but.*** Définit une génération guidée qui permet d'identifier les cas de test qui assureront la couverture d'un but (e.g. d'une instruction ou branche) donné, sans pour autant imposer les chemins de test à suivre pour y arriver. Cette approche est beaucoup plus forte que sa rivale aléatoire. Cela dit, la prédiction de la couverture, pour un but donné, est délicate vu que l'approche adopte le concept « *find-any-path* ».
- ***L'approche orientée chemins.*** Permet la génération de données de test en respect d'un ensemble de chemins bien spécifiques. Elle a lieu en deux étapes : (a) une première étape qui consiste en la sélection des chemins à couvrir en faisant usage de l'information portant sur le flot de contrôle, et (b) une deuxième correspondant à la génération des données de test appropriées pour ces chemins. Elle est considérée comme l'approche la plus forte des trois mentionnées, et est réputée permettre une meilleure prédiction de la couverture.

Pour cette question de la génération des cas de test, nous avons opté pour l'approche *orientée chemins*. Dans ce qui suit nous allons présenter les travaux qui ont adopté cette même approche pour leur génération des cas de test pour les processus BPEL.

3.2 Travaux existants

Dans la littérature, on trouve quatre principaux travaux qui ont traité de cette problématique de génération des cas de test pour un test unitaire appliqué aux processus BPEL exécutables. Les points forts et faibles de chacun de ces travaux sont présentés comme suit :

Lertphumpanya et Senivongse (24). Proposent un outil qui permet la génération des cas test (qu'il regroupe en une suite de test) pour la couverture des chemins de base dans un processus BPEL donné. Pour réussir cela, cet outil crée le graphe de flot de contrôle du processus BPEL à tester pour, par la suite, générer les chemins de base en respect de la complexité cyclomatique de McCabe. En dernier, ledit outil génère les données de test qui habilleront ces chemins, ainsi que les lanceurs et bouchons simulant les partenaires du processus à tester et qui sont nécessaires à l'exécution des tests.

Cependant, l'outil ne considère que les constructeurs séquentiels, conditionnels et répétitifs du BPEL et exclut le traitement de toute structure à comportement concurrent (e.g., `<flow>`, `<eventHandler>`). Ainsi, les concepteurs de cet outil n'ont pas été confrontés aux problèmes de représentation des flots parallèles et d'adaptation de la complexité de McCabe pour un langage concurrent comme BPEL. De plus, l'outil ne traite pas le problème des chemins infaisables[1].

Yan et *al.* (76). Proposent une méthode de génération des cas de test qui prend en compte le côté concurrent du BPEL. Pour ce faire, la méthode s'est vue tailler un graphe de contrôle étendu, baptisé *XCFG* (*eXtended Control Flow Graph*). Cette version étendue, du graphe de contrôle usuel, a été définie pour représenter des processus BPEL de nature séquentielle ou concurrente. À travers un tel graphe, on génère les chemins séquentiels qu'on combine, le cas échéant, pour former des chemins de test concurrents. Par la suite, la méthode préconise l'usage d'un solveur de contraintes pour la résolution des contraintes présentes dans ces chemins et la génération de cas de test faisables. En sus de la couverture de « tous-les-chemins » (avec un critère *0-1 répétition*), cette méthode considère aussi la couverture des chemins de base.

En revanche, telle que présentée, la traduction d'un processus BPEL en XCFG est loin d'être intuitive car les constructeurs du BPEL n'ont pas été traduits en des représentations

[1] Chemins ne correspondant à aucun comportement effectif du code testé et qui ne seront donc jamais suivis lors de l'exécution de ce dernier.

graphiques suivant un modèle (template) uniforme qui aurait simplifié cette traduction en une opération d'assemblage de telles représentations. De plus, les auteurs de ladite méthode n'ont traité que superficiellement les `<eventHandlers>` et `<faultHandlers>`. Ces auteurs n'ont pas tenu compte non plus de cette question vitale de l'éventuelle présence du code mort (code non exécuté) et des règles de maintien des dépendances de contrôle qui s'ensuivent. À titre d'exemple, le deuxième scénario d'exécution du code donné au paragraphe 1.4.3.3 ne peut être reflété par une représentation de ce code en XCFG — scénario au cours duquel l'activité *B* ne sera pas exécutée et sera donc considérée comme du code mort, d'autant que la règle, exigeant à ce que le lien *BtoC* sortant de *B* soit mis à faux, sera à appliquer pour le maintien de la dépendance de contrôle liant les activités *B* et *C*.

Par ailleurs, Yan et *al.* ont proposé une méthode pour la génération des chemins de test qui est compliquée, et cela vu le nombre de vérifications à entreprendre avant chaque combinaison de chemins séquentiels pour la formation d'un chemin concurrent. Ils n'ont aussi proposé qu'une représentation informelle et limitée pour leurs chemins (séquentiels et concurrents). Certes, ces auteurs ont considéré le côté concurrent du BPEL, mais n'ont pas cherché à adapter la complexité cyclomatique de McCabe de façon appropriée pour ce côté concurrent.

Yuan et *al.* (81). Proposent une méthode de génération des cas de test qui, elle aussi, traite le côté concurrent du BPEL, mais en se basant sur une différente extension, du graphe de contrôle usuel, que l'on a baptisée *BFG* (*BPEL Flow Graph*). De plus, au lieu de générer des chemins séquentiels pour ensuite les combiner en des chemins concurrents, cette méthode permet de directement générer ces derniers, sans combinaison aucune. Finalement, la méthode emboîte le pas à celle proposée par Yan et *al.* en faisant appel à un solveur de contraintes pour la génération des données de test; données que l'on combine avec les chemins de test générés pour constituer des cas de test complets. La méthode ne considère qu'un seul type de couverture : la couverture de « toutes-les-branches », *i.e.*, couverture de « tous-les-arcs ».

Comme c'était le cas pour la méthode de Yan et *al.*, celle de Yuan et *al.* nous propose une traduction en BFG qui n'est pas intuitive et qui ne prend pas en compte la question de présence du code mort et des règles de maintien des dépendances de contrôle qui s'en suivent. De plus, si Yan et *al.* n'ont abordé que partiellement et superficiellement le traitement *exceptionnel* (i.e., `<event/faultHandlers>`) en BPEL, Yuan et *al.* ne l'ont du tout pas abordé. Enfin, l'algorithme de génération des chemins proposé par Yuan et *al.* se base sur une représentation matricielle de ces derniers qui n'est pas adéquate (voir sous-section 5.2.4).

Li et *al.* (26). Proposent une approche orientée boîte grise pour le test des solutions SOA qui se servent du BPEL comme élément clé pour leur implémentation. Pour la génération des cas de test, cette approche adopte et le modèle BFG et l'algorithme de génération des chemins de Yuan et *al.*. En fait, ladite approche étend le modèle BFG pour tenir compte des autres caractéristiques avancées du BPEL comme le traitement exceptionnel, de même qu'elle améliore l'algorithme de génération des chemins et données de test de façon à anticiper la découverte des chemins infaisables. En particulier, l'approche considère la couverture « toutes-les-branches » et la couverture « tous-les-chemins » avec quelques autres types de couvertures sous-jacents.

Présentant leur modèle BFG comme une extension de celui de Yuan et *al.*, Li et *al.* n'ont malheureusement, nulle part dans leur article, ni mentionné ni traité la question de présence du code mort et des règles de maintien des dépendances de contrôle qui s'en suivent. Outre cela, vu que leur traduction est basée sur un modèle BFG étendu mais non repensé, elle reste donc loin d'être intuitive. Enfin, leur algorithme de génération des chemins de test améliore celui de Yuan et *al.* sans pour autant se passer de la représentation matricielle des chemins qui aurait due être remplacée par une autre qui soit plus adéquate.

Les quatre travaux présentés ci-dessus portent tous sur la version antérieure de la spécification BPEL, *i.e.*, le BPEL 1.1 (9), et non pas sur la dernière, *i.e.*, le BPEL 2.0 (1).

Finalement, et concernant toujours ce test unitaire appliqué aux processus exécutables BPEL, d'autres travaux existent, mais aucun d'eux ne traite la question de génération des cas de test. Ces travaux, comme celui de Mayer et Lübke (38) ainsi que celui de Li et Sun (27), portent plutôt sur la question d'*exécution* des cas de test que sur leur génération.

3.3 Notre approche

Contrairement aux travaux décrits précédemment, le nôtre porte sur la dernière version du BPEL (i.e., le BPEL 2.0) et les nouveaux constructeurs et règles que cette version apporte par rapport à son antécédente. Notre travail adopte l'approche de génération des cas de test relatée par la figure 3.1. Cette approche est une adaptation de l'approche orientée chemins pour le cas du BPEL. Elle consiste en quatre principales étapes :

1. ***La traduction en B-CFG***. Cette première étape vise à traduire le processus correspondant à l'unité de test en un graphe **B**-CFG (*BPEL Control Flow Graph*). Pour que cette traduction soit intuitive, nous avons tenu à ce que les constructeurs du BPEL soient tous

traduits en des représentations graphiques (des patrons de traduction) suivant un modèle (template) *uniforme*. Ainsi, la traduction sera simplifiée en une opération d'assemblage des représentations graphiques images des constructeurs constituant le processus à tester. Cette façon de faire s'inspire de celle étant suivie par les équipes d'Ouyang(48) et de Lohmann (30; 56) pour leurs transformations respectives du BPEL en réseaux de Petri. Notre traduction tiendra particulièrement compte du côté concurrent du BPEL, ainsi que de la question de présence du code mort et des règles de maintien des dépendances de contrôle qui s'en suivent.

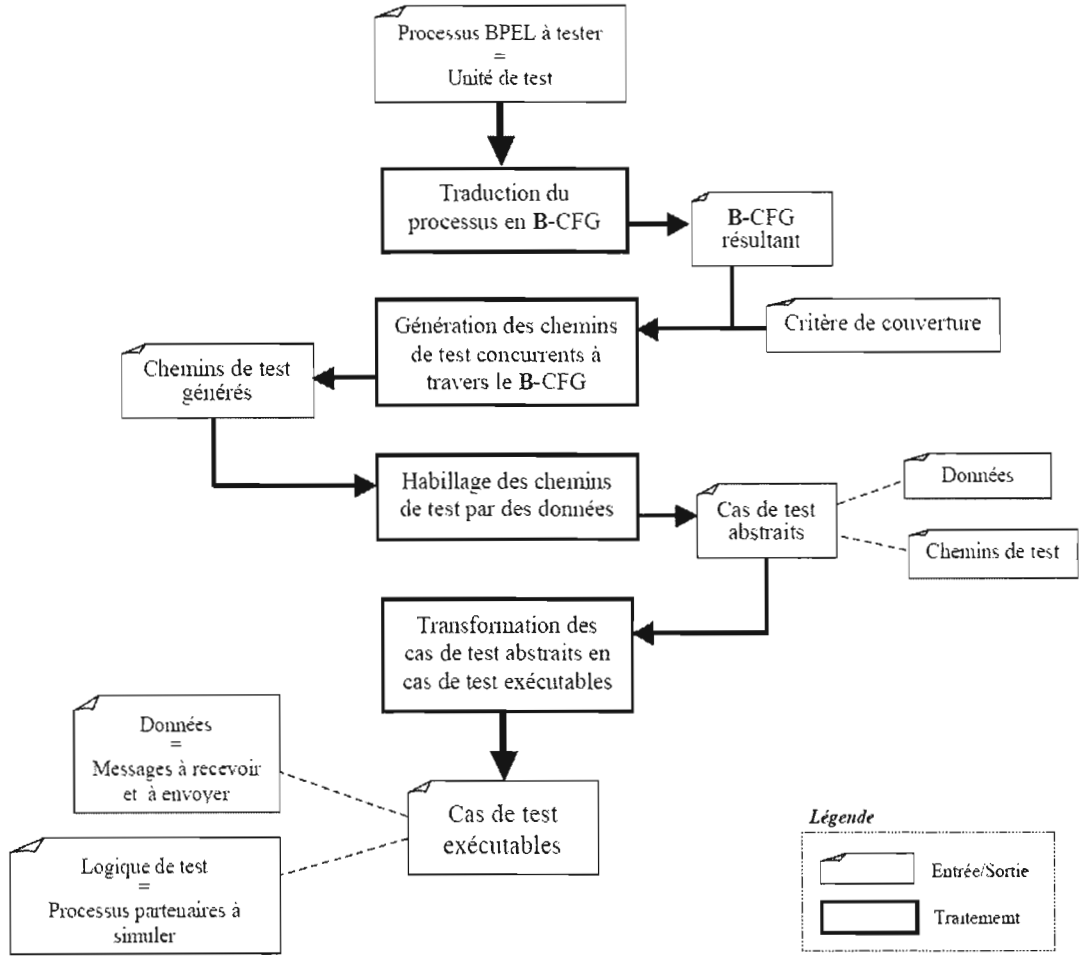


Figure 3.1 L'approche à suivre pour la génération des cas de test.

2. ***La génération des chemins***. Cette deuxième étape consiste à générer des chemins de test *concurrent* à travers le graphe B-CFG résultant de l'étape précédente, et cela en

adéquation à un critère de couverture donné. Pour nous, un chemin de test concurrent peut représenter un comportement concurrent ou séquentiel. Pour le cas concurrent, ce chemin correspondra à une liste partiellement ordonnée des actions définies par le comportement concurrent représenté. Cependant, dans le cas séquentiel, ce chemin correspondra à une liste totalement ordonnée des actions définies par le comportement séquentiel représenté[2]. Nous verrons, dans ce qui suit, comment représenter d'une façon formelle et adéquate de tels chemins. En ce qui concerne les critères de couverture qu'une génération de chemins aurait à satisfaire, nous considérons les critères : « tous-les-chemins » (avec le critère *0-1 répétition* pour les boucles), « toutes-les-branches » et « chemins de base » avec une adaptation de ce dernier pour le cas du BPEL. Lors de cette étape, certains types de chemins infaisables seront éliminés.

3. ***Habillage des chemins***. Cette avant-dernière étape vise à générer des données de test qui permettront l'exécution des chemins de test concurrents résultant de l'étape précédente. Les chemins, pour lesquels il serait impossible de trouver des données de test, sont des chemins infaisables que nous tâcherons à éliminer. Pour cette étape, nous présenterons les grandes lignes d'une solution de génération de données qui procède à la collection et la résolution des contraintes présentes dans un chemins de test — solution qui sera à mettre en oeuvre dans le cadre de futurs travaux. Combinées aux chemins de test, les données de test qui seront générées constitueront des cas de test abstraits.

4. ***La transformation des cas de test abstraits***. Cette dernière étape vise à transformer les cas de test abstraits préalablement générés en des cas de test exécutables. Cette étape n'est pas prise en compte par notre travail. Cela dit, des pistes pour la spécification (i.e., la définition de la logique de test et son enrichissement par les données à échanger) ainsi que l'exécution de tels cas de test seront fournies.

3.4 Conclusion

Au cours du présent chapitre, nous avons présenté les principaux travaux qui ont traité de cette problématique de génération des cas de test, tout en portant une attention particulière aux

[2]Le concept de « chemin de test concurrent » a été considéré pour le cas et séquentiel et concurrent car, au fond, le cas séquentiel n'est qu'un cas particulier du cas concurrent où le nombre, des actions qui ne sont pas ordonnées entre elles, est nul.

travaux s'inscrivant dans le cadre d'un test unitaire appliqué aux processus BPEL exécutables. Après énumération des points forts et des points faibles de chacun de ces travaux, nous sommes passés à présenter notre travail et ses nouveaux apports à l'égard de la problématique traitée. Comme suite logique au présent chapitre, nous exposerons dans le chapitre qui suit l'une des étapes centrales de notre approche qui est la traduction du code BPEL en graphe CFG. La traduction envisagée concernera aussi bien le comportement *normal* d'un processus BPEL qu'une partie de son comportement *exceptionnel*.

CHAPITRE IV

TEST DES PROCESSUS BPEL : LA TRADUCTION EN B-CFG

4.1 Introduction

Le présent chapitre a pour objectif d'exposer les *patrons* et *règles* qui seront adoptés pour la traduction d'un processus BPEL en un graphe de flot de contrôle (i.e., en un CFG). Pour que cette traduction puisse reproduire fidèlement les particularités comportementales du processus BPEL à tester, en particulier son comportement *concurrent*, nous avons été amenés à l'élaboration d'une version étendue du CFG. Cette version augmentée (syntaxiquement et sémantiquement) est baptisée ***B-CFG*** pour *BPEL Control Flow Graph.*

À l'image des travaux réalisés pour la transformation de la sémantique BPEL en réseau de Petri (48; 56) ainsi qu'en d'autres formalismes (15; 16), la traduction envisagée concerne aussi bien le comportement *normal* d'un processus BPEL qu'une partie de son comportement *exceptionnel* (i.e., *events, faults*). Il est aussi à noter que parmi les quatre perspectives décrites au paragraphe 1.4.1, nous ne retiendrons que deux pour notre traduction, à savoir la perspective *flot de contrôle* et la perspective *données.*

4.2 BPEL et le CFG

« Que l'orchestrateur (processus BPEL) à tester soit spécifié par des flots structurés ou non-structurés, sa structure interne (i.e., *computation structure*) peut être représentée comme un graphe de flot, comme c'est le cas pour tout programme ordinaire »(6). À la fois remué et motivé par cette affirmation, on pourrait a priori penser que la traduction du BPEL en CFG se ferait en appliquant les *mêmes* patrons et règles qui sont utilisés pour la construction des CFG de programmes adhérant à une programmation des plus usuelles : la programmation *séquentielle.*

Cependant, rien ne garantit qu'un flot BPEL, structuré ou non soit-il, serait forcément séquentiel mais pourrait aussi être de nature *concurrentielle*. Au fait, l'une des particularités principales du langage BPEL est qu'il donne cette possibilité de définir des comportements *concurrentiels* complexes (76). C'est cette particularité d'ailleurs qui le classe au rang des langages de programmation concurrente. À vrai dire, la concurrence dénote un point parmi d'autres qui définissent les particularités du BPEL dont il faut prendre compte et note lors de la traduction en **B**-CFG. Cela nous conduit donc à trouver de nouveaux patrons et règles qui siéent plus à la traduction de ces particularités en un graphe de flot.

4.2.1 Les particularités du BPEL

Un certain nombre des particularités du BPEL a été décortiqué par Bucchiarone et al (6). Outre la concurrence, les particularités identifiées embrassent la *synchronisation* (i.e., liens de contrôle), le couple *scopes/exceptions*, et en dernier les *événements* et les *timers*. Pour enfoncer le clou, nous n'omettrons surtout pas de mettre en lumière d'autres particularités aussi importantes que leurs congénères précités, et qu'il faut absolument prendre en considération lors de la traduction en **B**-CFG. Parmi ces particularités, mentionnons :

- Les *« initial start activities »* et leurs ordres d'exécution : contrairement à un programme ordinaire, un processus BPEL pourrait voir son exécution concurremment déclenchée, par différentes données d'entrée (i.e., messages entrants) destinées à *différents* points d'entrée (e.g., à différentes activités `<receive>`). Au fait, selon la norme BPEL (1), la définition d'un processus pourrait inclure plusieurs *« initial start activities »* (i.e., *ISA*) dont l'attribut `createInstance` est à 'yes'. Or, seule l'une de ces activités pourrait mener à la création d'une nouvelle instance, de même que l'exécution des autres *ISA* ne doit être permise qu'après la fin d'exécution de celle qui aurait déclenché la création de l'instance. Dans ce cas, il faut donc s'attendre à ce que la traduction en **B**-CFG, du processus incluant ces *ISA*, résulte en plusieurs graphes; chacun de ces graphes représentera une instance dont la création aurait à être déclenchée par l'une des *ISA* identifiées, et cela afin de pouvoir prendre compte des différents ordres d'exécution qui seront définis entre ces dernières.

- Les liens de contrôle et l'*acyclicité* : tout graphe orienté qui est l'image d'un processus BPEL défini à base de liens de contrôle, tel que celui de la figure 1.3, est censé être acyclique (64; 76). Cela est dû aux règles qui, dictées par la norme BPEL, interdisent aux liens de contrôle de former des cycles; seules les activités décrivant un comportement itératif (e.g.,

<while>) peuvent résulter en des cycles. Du coup, le graphe **B**-CFG à produire devra maintenir cette propriété d'acyclicité s'appliquant aux liens de contrôle. Pour ce faire, une bonne traduction de tels liens s'impose et cela en tenant compte de la sémantique associée à ces derniers (e.g., *Dead Path Elimination*).

- Les « *isolated scopes*[1] », les « *event Handlers* » et les variables partagées (76) : les avènements multiples du même événement déclencheur d'un `eventHandler`, de même que la sémantique liée à un « *isolated scopes* », font surgir à la surface le problème complexe de gestion du contrôle d'accès aux variables partagées.

4.2.2 États des activités et des liens de contrôle

Outre les particularités énumérées ci-dessus, il est aussi impératif de penser aux mesures à prendre pour représenter au mieux les différents états qu'une activité ou qu'un lien de contrôle pourrait prendre en cours d'exécution du `<flow>` les enchâssant. Une énumération exacte combinée à une traduction adéquate de ces états d'exécution sera bénéfique et aidera grandement à trouver les bons patrons de traduction à associer à chaque composante du langage BPEL. Dans la littérature, nous avons trouvé que plusieurs équipes ont essayé de définir ces états. Nous en avons retenues trois des plus importantes : l'équipe de Yuan (81), l'équipe de Yan (76) et celle d'Ouyang (48).

Pour l'équipe de Yuan, la sémantique du DPE contraint à prévoir trois états d'exécution autant pour les activités que pour les liens, à savoir : l'état *exécuté*, l'état *non-exécuté* et l'état *dead path*. En revanche, l'équipe de Yan distingue les états qu'une activité pourrait prendre de ceux qui seraient exclusivement attachés aux liens de contrôle. Pour les activités, cette deuxième équipe considère les deux états : *exécutée* et *sautée* (i.e., *skipped*), tandis qu'aux liens de contrôle, elle accorde deux autres états : l'état *normal* et l'état *DPE*. De même, quand l'équipe d'Ouyang parle du comportement normal (i.e. *normal processing*) et de saut d'activités (i.e. *skipped activities*) cela sous-entend qu'une activité ne pourrait être que sous deux états. Restée fidèle à une classification binaire, cette troisième équipe d'Ouyang accorde deux états aux liens de contrôle : *true* et *false*. Ces deux états sont respectivement identiques à l'état *normal* et *DPE* définis par la deuxième équipe, *i.e.*, l'équipe de Yan.

---

[1] Une *isolated scope* est une <scope> dont l'attribut *isolated* vaut "yes" La mise de cet attribut à telle valeur permet le contrôle de l'accès concurrent à des ressources partagées, en particulier à des variables qui sont partagées par des "tâches" concurrentes (voir (1, section 12.8)).

Pour notre traduction, la classification des états proposée par la deuxième et troisième équipes semble être plus appropriée car elle permet de distinguer entre les états d'exécution à associer aux activités, d'une part, et ceux à attacher aux liens de contrôle, d'autre part. Cela dit, ces états seront enrichis par d'autres et leur interprétation se verra réadaptée. En fait, nous avons scindé les états adoptés en deux groupes : un premier groupe de trois états à associer aux activités et un deuxième de deux autres à associer aux liens de contrôle (voir tableau 4.1).

	État	Description
ACTIVITÉ	*Non-exécutée*	Si la <joinCondition> de l'activité est évaluée à faux, son action[2] ne sera pas exécutée.
	Exécutée	Si la <joinCondition> de l'activité est évaluée à vrai et que cette dernière n'est pas à sauter[3], son action sera exécutée.
	Sautée	Si l'activité est à sauter, son action ne sera pas exécutée et cela que sa <joinCondition> soit évaluée à vrai ou à faux.
LIEN DE CONTRÔLE	*Exécuté*	Si la <transitionCondition> du lien de contrôle est évaluée à vrai après une exécution réussie de l'activité qui en est la source.
	Dead path	Si la <transitionCondition> du lien de contrôle est évaluée à faux, ou si ce lien fait partie des liens sortants d'une activité qui a été sautée ou non-exécutée[4].

Tableau 4.1 États des activités et des liens de contrôle dans un <flow>.

[2]Nous rappelons que l'action d'une activité de base est atomique et en constitue le coeur, alors que celle d'une activité structurée consiste en l'exécution des activités de base ou structurées qui y sont enchâssées.

[3]Une activité est susceptible d'être sautée si elle fait partie : (a) d'une branche non sélectionnée d'un <pick> ou d'un <if>, (b) de l'ensemble des activités enchâssées par une activité structurée non-exécutée ou ayant été sautée, (c) du reste des activités non amorcées d'un <scope> où une erreur a eu lieu, (d) d'un <fault/terminationHandler> n'ayant pas été exécuté.

[4]À l'état *non-exécutée* d'une activité ayant l'attribut <suppressJoinFailure> à yes, tous les liens de contrôle dont elle est source seront mis à l'état *dead path*. Cependant, si l'activité est à l'état *non-exécutée* et que son attribut <suppressJoinFailure> est mis à no, seuls les liens de contrôle dont elle est source, et dépassant les frontières du <scope> auquel elle appartient, seront mis à l'état *dead path*.

La sémantique que nous avons donnée à l'état *sautée* est plus proche du « *skipped processing* » défini par Ouyang (48) qu'elle ne l'est de la sémantique liée à l'état *sautée* (i.e., *skipped*) défini par Yan (76). L'état *sautée* de Yan est plutôt similaire à notre état *non-exécutée*.

En ce qui concerne l'état *dead path*, les deux cas de figures décrits dans le tableau 4.1 énumèrent deux situations des plus importantes dans lesquelles s'applique la mise des liens de contrôle à cet état. Cette mise à l'état *dead path* pourrait aussi s'appliquer dans le cas où une erreur est déclenchée suite à l'évaluation de la `<transitionCondition>` d'un lien de contrôle (voir 4.6.3.4). L'état *dead path* est similaire à l'état *false* que la norme BPEL a associé aux liens de contrôle, de même que l'état *true* qu'elle a défini pour ces derniers est similaire à l'état *exécuté*[5]. Enfin, il faut rappeler que la mise à l'état *dead path* est vitale pour le maintien des dépendances de contrôle entre activités, car en l'adoptant nous réussirons toujours à faire suite au flot d'exécution jusqu'à atteindre le point final du programme BPEL à tester.

4.3 D'ailleurs, pourquoi traduire?

Afin de répondre d'une façon objective à cette question, on n'a qu'à rappeler le dessein de ce travail qui consiste en la génération des cas de test pour un processus BPEL donné. Pour faciliter cette génération, il est d'usage de passer par un modèle intermédiaire représentant le code à tester sous un formalisme donné. En particulier, le graphe CFG est l'un des modèles les plus utilisés dans le domaine de génération de test. Contrairement à d'autres modèles qui sont, certes, bien adaptés à la représentation du flot de contrôle mais pas autant à celle du flot de données (e.g., réseaux de Petri (15; 48)), le graphe CFG, de par sa simplicité et son efficacité, est le modèle qui sied le plus à notre cas. En effet, ce dernier se distingue par sa capacité à combiner flot de contrôle et flot de données en un seul graphe — flot de contrôle à base duquel une génération future des chemins de test sera rendue possible, et flot de données que l'on juge incontournable pour la phase de génération de données[6].

---

[5] La norme BPEL définit aussi un troisième état : l'état *unset*. Un lien de contrôle est mis dans pareil état à chaque fois que le `<flow>` l'enveloppant est activé. Pour notre traduction, cet état est aussi considéré, mais juste comme valeur d'initialisation du lien, en attendant que ce dernier prenne l'état *exécuté* ou *dead path*.

[6] Le diagramme d'activité UML pourrait lui aussi être utilisé pour cette modélisation intermédiaire vu qu'il est capable de concilier flot de contrôle et flot de données par l'usage respectif de ses nœuds *objets* et *de contrôle*. Or, ce diagramme UML, comme le graphe CFG, ne supporte pas tous les patrons de workflow que le langage BPEL adopte (42). Ainsi, que l'on opte pour ce diagramme ou pour le CFG, on va devoir étendre celui qu'on a choisi afin d'aboutir à une modélisation complète du langage BPEL.

Qu'un programme structuré à tester soit de nature concurrente ou non, sa traduction en un graphe CFG permet, en premier lieu, d'éliminer toutes les structures de contrôle dictées par son code (74). Vu que les processus BPEL sont généralement de nature structurée, ils devraient donc, lors de leur traduction en **B**-CFG, bénéficier du même effet, c'est-à-dire, voir leurs structures de contrôle (e.g. `<flow>`, `<pick>`) éliminées. Toujours au profit de la phase de génération de tests, la traduction en **B**-CFG mènera à deux autres nouveaux acquis, à savoir (81) :

- Reproduire les flots de contrôle implicites et disjoints en des flots de contrôle explicites et connectés (e.g., traitement d'exception).
- Démêler les structures masquées du BPEL (e.g., DPE) à des structures qui pourraient être directement traversables lors du parcours du graphe construit.

Cette traduction se veut donc critique et nous épargnera les efforts d'une manipulation, éventuellement ardue, du graphe orienté BPEL qui est une visualisation directe du code à tester — le graphe de la figure 1.3 est un exemple d'une telle visualisation. « Ce graphe orienté reste très compact et n'exprime pas les alternatives dynamiques d'exécution du code BPEL »(76). Entre autres, ces alternatives seraient le résultat du dépistage des effets dynamiques de la *concurrence*, des *DPE* ou des *exceptions* (76).

Pour faire en sorte que les alternatives précitées soient représentées sous une forme appropriée et que la traduction escomptée soit la plus explicite possible, la syntaxe du **B**-CFG sera enrichie par de nouveaux éléments. Par ailleurs, le **B**-CFG résultant inclura trois classes d'information (81) dont deux sont directement liées aux perspectives du flot de contrôle et de données présentées plus haut. Ces trois classes d'information sont les suivantes :

- classe de l'information structurelle portant sur le flot de contrôle d'un programme BPEL;
- classe de l'information portant sur son flot de données;
- classe de l'information sémantique telle que celle émanant de la sémantique du DPE.

L'information *sémantique* qui serait exposée par le graphe **B**-CFG en décrit l'une des particularités les plus importantes, et cela même vis-à-vis des graphes CFG utilisés pour la représentation d'autres langages concurrents tels que le langage Ada (20; 79).

4.4 Démarche générale

La démarche générale à suivre, pour la traduction de tout processus BPEL en **B**-CFG, sera une variante de celle définie pour la transformation de la sémantique BPEL en réseau de Petri (32; 56). La traduction escomptée sera guidée par la syntaxe BPEL. En effet, vu que la construction de tout processus BPEL se fait en *connectant*, ou en organisant hiérarchiquement, les constructeurs du langage de façon à aboutir au code désiré, il va de soi donc que chacun de ces constructeurs soit, dans un premier temps, traduit en un *sous-graphe* **B**-CFG pour être par la suite assemblés et résulter ainsi en un graphe représentant le processus à tester dans son intégralité. En définitive, de tels sous-graphes seront considérés comme des patrons de traduction dont nous ferons usage pour pouvoir passer du code BPEL en graphe **B**-CFG.

Une collection de ces patrons représentera l'incarnation de la sémantique BPEL en **B**-CFG. Il est à noter que tout l'effort sera déployé afin que ces patrons fassent eux-mêmes ressortir les trois classes d'information précitées plus haut et qu'un **B**-CFG est censé inclure.

4.5 Le graphe B-CFG

La définition structurelle et formelle du **B**-CFG est donnée par le quintuplet *(**s**, **f**, **N**, **E**, **V**)* dont chacun des éléments est décrit comme suit :

- ***s*** et ***f*** représentent respectivement le nœud d'*entrée* et de *sortie* du graphe.
- ***N*** représente l'ensemble des *nœuds* du graphe. Chacun de ces nœuds appartient à l'une des classes de nœuds suivantes : la classe des nœuds normaux, des nœuds de décision, des nœuds de fusion, des nœuds de bifurcation ou celle des nœuds de jointure.
- ***E*** représente l'ensemble des *arcs* du graphe. À chacun de ces arcs, on associe un quadruplet *<sn, tn, pr, ac>* qui, en partie, s'inspire de celui proposé par Yan (76), avec :
  - **sn** (i.e., *source node*) : dénote le nœud source de l'arc.
  - **tn** (i.e., *target node*) : dénote le nœud destination de l'arc.
  - **pr** (i.e., *predicate*) : enveloppe l'expression conditionnelle contrôlant l'exécution de l'action qui est associée à l'arc. La nature de cette expression est calquée sur celle des « expressions booléennes» du langage BPEL.
  - **ac** (i.e., *action*) : enveloppe l'action à exécuter suite à une évaluation positive de la condition contenue dans l'élément *pr* de l'arc. Entre autres, cette action pourrait

consister en l'enregistrement du message reçu (e.g., message reçu d'un `<receive>`) ou en une affectation de la valeur *true* ou *false* à la variable globale décrivant l'état d'un lien de contrôle (voir 4.6.3).

Par défaut, les valeurs des éléments *pr* et *ac* sont mis à *Null* (i.e., aucune condition à vérifier et aucune action à exécuter). Autrement dit, s'il se trouve, lors de la description des patrons de traduction, que les éléments *ac* et *pr* d'un arc n'ont pas été spécifiés, cela sous-entend que ces deux éléments gardent la valeur *Null*.

- V représente l'ensemble des *variables globales* du graphe, ensemble que nous avons scindé en deux sous-ensembles. Un premier sous-ensemble où sont regroupées les variables déclarées explicitement dans la définition du processus BPEL à tester. Un deuxième sous-ensemble où sont regroupées les variables reflétant l'ensemble des sémantiques qui feraient surface suite à une traduction du processus BPEL à tester en graphe **B**-CFG (e.g., pour chaque lien de contrôle traduit en **B**-CFG, nous associons une variable globale décrivant son état, en particulier, son état *dead path* qui est le reflet d'une sémantique de DPE).

D'après Xanthakis (74) : « un graphe CFG n'a qu'une *seule* entrée et une *seule* sortie ». Cela s'applique aussi pour le graphe **B**-CFG qui n'est rien d'autre qu'une variante du graphe CFG usuel. Au fait, pour le test unitaire des processus BPEL, l'unité de test considérée est le processus lui même, privé de ses partenaires. Outre cela, pour son test, un processus BPEL ne pourra être subdivisable en des composantes indépendantes, mais il devra toujours être pris comme un tout. Par conséquent, le graphe de contrôle **B**-CFG, correspondant au processus BPEL qui est l'unité de test considérée, ne sera en aucun cas donné comme un ensemble de plusieurs graphes de contrôle qui seraient images de composantes indépendantes, avec éventuellement une multitude de nœuds d'entrée et de sortie. Par contre, la traduction du processus considéré résultera en un seul graphe **B**-CFG qui sera doté d'un *seul* nœud d'entrée et un *seul* nœud de sortie.

En revanche, on trouve que Yuan (81) et Li (26) ont utilisé pour leur traduction un graphe de contrôle, baptisé BFG, qui ne respecte pas cette contrainte. Ces deux auteurs ont défini leur graphe BFG comme un quadruplet dont le nœud de sortie n'est pas supposé être unique, mais plutôt augmenté à plusieurs instances (à un ensemble de nœuds de sortie, éventuellement multiples).

En ce qui concerne l'ensemble N, nous constatons que deux écoles se partagent le terrain pour décider de la *nature* et de la *sémantique* à associer aux éléments de cet ensemble (i.e., aux nœuds). La première école, à laquelle Yan (76) adhère, stipule que tous les nœuds devront être homogènes de nature et que leurs sémantiques respectives devront refléter les différents états du programme. Cependant, sa rivale, *i.e.*, la deuxième école dont Yuan (81) et Li (26) sont des partisans, différencie bel et bien entre les nœuds dits *de contrôle* (e.g., nœud de bifurcation, nœud de décision) et les nœuds dits *normaux*.

A priori, il paraît que l'approche de la première école mènera à des graphes relativement simples, qui dévoilent l'essentiel du flot de contrôle, mais qui seraient surtout riches de par la quantité de données représentée. Cependant, la deuxième école s'intéresse, certes, au flot de données, mais met plus l'accent sur le flot de contrôle qui serait démêlé d'une manière plus explicite (graphiquement parlant). Cette dernière façon de faire risque donc d'aboutir à des graphes qui seraient relativement de grande taille.

Pour notre traduction, nous avons opté pour la deuxième école pour éviter au maximum d'avoir à analyser les données présentes dans le graphe, lors de la génération des chemins de test. Autrement dit, nous avons voulu réduire au minimum la quantité de données présente dans le graphe, et ainsi mettre l'emphase sur le flot de contrôle en dévoilant au maximum la structure du code à traduire et en distinguant les nœuds de contrôle des nœuds normaux. Au fait, en plus des nœuds de contrôle qui font d'office partie de la syntaxe d'un CFG usuel, à savoir les nœuds de décision et de fusion, nous avons rajouté deux autres nœuds de contrôle : les nœuds de bifurcation et de jointure, afin de rendre le flot de contrôle démêlé par les graphes B-CFG plus explicite. Le tableau 4.2 relate la façon dont chacune des classes de nœuds sera représentée.

Légende	Description
	Représente les nœuds normaux.
	Représente les nœuds de décision et de fusion.
	Représente les nœuds de bifurcation et de jointure.

Tableau 4.2 Représentation des classes de nœuds.

En fait, nous nous sommes épaulés sur la variété des nœuds de contrôle adoptés pour modéliser au mieux les patrons de *workflow* définis par Kiepuszewski et *al.* (22) — patrons qui

ont été élaborés pour une modélisation formelle des constructeurs de base de disjonction (i.e., AND/OR-split) et de conjonction (i.e., AND/OR-join) lié à un langage de *workflow* tel que le langage BPEL. Finalement, il faut signaler que pour des raisons de simplification[7], un nœud de décision *ND* pourrait être fusionné avec un nœud de fusion *NF*, comme un nœud de bifurcation *NB* pourrait l'être avec un nœud de jointure *NJ*. Ce mariage éventuel des nœuds est dépeint par la figure 4.1.

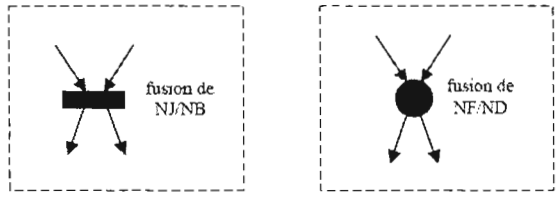


Figure 4.1 Fusion des nœuds de contrôle.

Quant à l'ensemble $\boldsymbol{E}$, chaque arc $< sn, tn, pr, ac >$ en faisant partie est un arc *dirigé* qui relie le nœud *sn* à son successeur *tn*. La définition d'un tel arc matérialise le fait que le nœud *tn* ne pourra être atteint via son prédécesseur *sn* que si la condition enveloppée par l'élément *pr* de l'arc, dans le cas où elle est spécifié (i.e., pr $\neq$ Null), est évaluée à vrai et que son action enveloppée par l'élément *ac*, si elle est spécifiée (i.e., ac $\neq$ Null), est exécutée suite à cette évaluation. Si le prédicat et l'action de l'arc ne sont pas spécifiés (i.e., pr = Null et ac = Null), le passage du nœud *sn* à son successeur *tn* est systématique.

Toutes écoles confondues, on s'entend à ce que l'ensemble $\boldsymbol{E}$ regroupe des arcs de natures différentes. Au fait, pour pouvoir démêler les structures masquées du BPEL, en particulier l'information sémantique qui y est encastrée (e.g., DPE), à des structures qui pourraient être directement traversables lors du parcours du graphe construit, nous avons pensé à trois types d'arc : *arcs normaux*, *arcs morts* (i.e., *dead path*) et *arcs de saut*. Ces trois types d'arcs vont nous permettre de représenter en **B**-CFG et les liens de contrôle et les liens que nous avons dénommés *liens implicites*[8].

---

[7] À l'image d'ailleurs de ce qui s'applique pour les nœuds de contrôle d'un diagramme d'activité UML.

[8] « À l'absence des liens de contrôle, ce sont les <scopes>, la sémantique BPEL, et les activités structurées qui déterminent quand est ce qu'une activité est prête à être exécutée » (1). Autrement dit, c'est la catégorie des *liens implicites* qui est dans ce cas sollicitée pour la définition de l'ordre d'exécution des activités.

Par ailleurs, les différents états, qu'une activité ou qu'un lien pourrait prendre, conduisent à classer les (sous) chemins qu'un graphe **B**-CFG pourrait inclure en différents types. Les types de (sous) chemins à distinguer sont au nombre de trois : le chemin *normal*, le chemin *mort* (i.e., *dead path*) et le chemin de *saut* (i.e., *skip path*). Ces trois types de chemins reflètent les différents types des scénarios d'exécution qu'un processus BPEL pourrait incorporer, alors que chacun d'eux est représenté en s'appuyant sur l'un des trois types d'arcs cités plus-haut. Le tableau 4.3 expose la représentation graphique associée à chacun de ces trois types d'arcs et résume la relation qui relie ces derniers aux trois types de chemins considérés.

Légende	Nom	Description
⟶	*Arc normal*	Faisant partie d'un (sous) chemin normal, ce type d'arc dépeint le comportement normal du processus traduit. Il pourrait correspondre à un lien implicite comme il pourrait modéliser l'état *exécuté* d'un lien de contrôle.
⋯⋯▸	*Arc mort*	Faisant partie d'un (sous) chemin mort, ce type d'arc dépeint la propagation de l'effet d'un DPE. Il modélise l'état *dead path* d'un lien de contrôle.
----▸	*Arc de saut*	Faisant partie d'un (sous) chemin de saut, ce type d'arc dépeint la propagation de l'effet de saut d'activités. Il ne pourrait modéliser que les liens implicites.

Tableau 4.3 Représentation des arcs.

Finalement, il est à noter que l'ensemble V des variables globales du graphe **B**-CFG a été adopté pour deux principales raisons. Premièrement, c'est parce que nous avons voulu éviter un éventuel retour au code BPEL à tester une fois que ce dernier est traduit en **B**-CFG, et cela en calquant et regroupant toutes les variables explicites qui y sont déclarées dans un premier sous-ensemble de l'ensemble V. Les données qui sont enveloppées par ce premier sous-ensemble seront utiles pour la phase de génération de données de test. Deuxièmement, c'est parce que nous avons voulu dévoiler les sémantiques cachées du BPEL et garder l'information s'y rapportant. Cette information (e.g., état d'un lien de contrôle) sera utile pour la phase de génération des chemins de test, et sera maintenue en des variables qui seront regroupées dans un deuxième sous-ensemble de l'ensemble V.

4.6 Les patrons et les règles

Dans cette section, nous présentons les *patrons* qui traduisent les constructeurs BPEL en des sous-graphes B-CFG, de même que les *règles* que nous adoptons afin d'aboutir à des graphes B-CFG simples et fidèles aux processus qu'ils modélisent.

4.6.1 Activités

Optionnellement, chaque activité BPEL, qu'elle soit de base ou structurée, pourrait mettre à l'affiche et ses attributs et ses éléments standard. En particulier, pour notre traduction en B-CFG, nous tenons compte de l'attribut *<supressJoinFailure>* et des deux conteneurs `<sources>` et `<targets>`. Respectivement, ces deux conteneurs enveloppent les éléments standard `<source>` et `<target>` de l'activité à traduire. La traduction de ces éléments, en considération de l'attribut `<supressJoinFailure>`, sera abordée dans la section 4.6.3.

4.6.1.1 Activités de base

La figure 4.2 illustre la traduction en B-CFG d'une activité de base X. Cette première traduction annonce d'emblée les couleurs de l'approche à suivre pour le reste à traduire. Toute activité, qu'elle soit de base ou structurée, sera traduite en un patron qui sera doté de deux nœuds : un premier nœud de début (i.e., s_X pour *start*) et un deuxième de fin (i.e., f_X pour *final*). Au niveau de chacun de ces patrons, nous veillerons à faire surgir et le comportement *normal* et, le cas échéant, le chemin de *saut* de l'activité à traduire. À vrai dire, contrairement à la traduction d'Ouyang (48), notre traduction adopte une règle dictant que seules les activités étant cibles ou sources de liens de contrôle seraient susceptibles d'être sautées et, par conséquent, seules de telles activités verront leurs patrons de traduction inclure des chemins de saut venant s'adhérer à un comportement normal omniprésent.

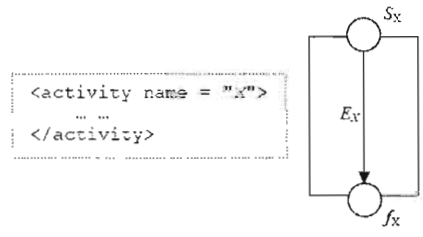


Figure 4.2 Activité de base.

Par application directe de cette règle, et comme le montre la figure 4.2, le patron de traduction d'une activité de base qui n'est ni cible ni source de liens de contrôle ne peut inclure de chemins de saut. L'arc E_X, combiné aux nœuds s_X et f_X, représente ainsi un comportement normal dépourvu de toute sémantique de saut. Cet arc est défini comme suit : $E_X = <s_X, f_X, pr$ = Null, $ac \in \{Enreg, Affect, Empty\}>$ avec $\{Enreg, Affect, Empty\}$ est l'ensemble d'actions dont une et une seule pourrait constituer le cœur de l'activité de base traduite. Respectivement, ces actions correspondraient à l'*enregistrement* d'un message reçu (e.g., cas d'un `<receive>` ou de la partie réceptionniste d'un `<invoke>` de type *requête-réponse*), à une *affectation* de valeurs (e.g., cas d'un `<assign>`) ou tout simplement à *ne rien faire*[9] (e.g., cas d'un `<empty>`).

4.6.1.2 Activités structurées

Les activités structurées seront traduites en s'appropriant l'approche décrite plus haut, à savoir en des patrons dotés de nœuds de début/fin, qui seraient aussi enrichis par d'autres types de nœuds et d'arcs. Dans un premier temps, c'est le comportement normal de ces activités[10] qui sera dénudé pour être par la suite enrichi, le cas échéant, par des chemins de saut.

4.6.1.2.1 L'activité <sequence>

Cette activité enveloppe une ou plusieurs sous-activités[11] qui sont exécutées séquentiellement, à l'image de l'exécution d'une suite d'instructions faisant partie d'un programme séquentiel classique. Le patron illustré par la figure 4.3(b) reproduit le flot de contrôle implicite, incarné par l'activité `<sequence>`, en un flots de contrôle *explicite*. Autrement dit, les liens implicites, que la sémantique du `<sequence>` décline, sont traduits explicitement en des arcs normaux. Ces arcs garantissent le fait qu'une sous-activité A_i ne sera exécutée qu'une fois l'activité A_{i-1} aurait terminé son exécution.

Pour un éventuel allègement de la représentation en graphe B-CFG, le patron de traduction du `<sequence>` pourrait être élagué pour résulter en un sous-graphe semblable à celui

---

[9] Vu que l'aspect temporel ne peut être représenté dans un graphe CFG qui est une représentation *statique* du code, le prédicat de l'arc E_X qui représentera une activité `<wait>` gardera la valeur *Null*, et il en sera de même pour son élément *ac*.

[10] À l'exception des autres activités structurées, l'activité `<scope>`, qui est étroitement liée aux comportements *exceptionnels*, verra son patron de traduction donné à la section 4.6.5.

[11] L'appelation *sous-activité* dénomme toute activité étant enchâssée dans une activité structurée.

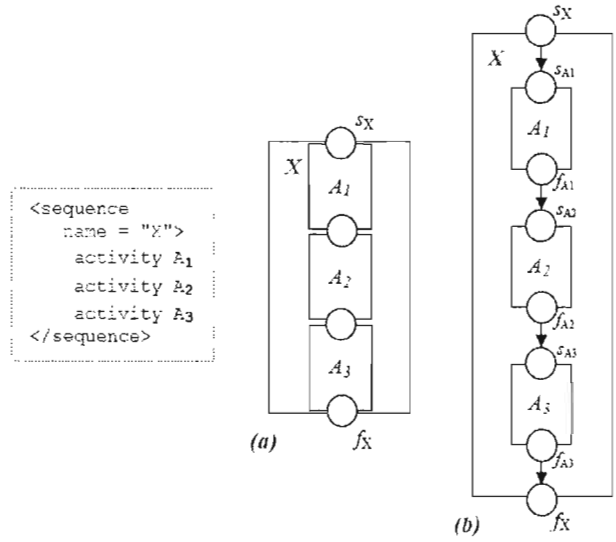


Figure 4.3 L'activité <sequence>.

dépeint par la figure 4.3 (a). Pour passer à un tel sous-graphe, les deux règles données ci-dessous devront être appliquées sur le sous-graphe initial dépeint par la figure 4.3(b) :

- Respectivement, les nœuds de début et de fin du <sequence> (i.e., s_X et f_X) sont à fusionner avec le nœud de début s_{A_1} de sa première sous-activité A_1 et le nœud de fin f_{A_n} de sa dernière sous-activité A_n, avec n est le nombre des sous-activités enveloppées par le <sequence> traduit (pour le cas de la figure 4.3, $n = 3$).
- Pour toute activité A_i, où i = 2...n, son nœud de début s_{A_i} sera fusionné avec le nœud de fin de l'activité A_{i-1} la précédant (i.e. $f_{A_{i-1}}$).

4.6.1.2.2 Les activités <while> et <repeatUntil>

L'activité <while> permet une exécution itérative de sa sous-activité A (figure 4.4(a)), et cela tant que sa condition C est évaluée à vrai au début de chaque itération. Cependant, une fois que cette condition ne tient plus (i.e. $!C$ = vrai), la sous-activité A est détournée et le <while> prend fin. La figure 4.4(a) représente le patron traduisant le <while> en B-CFG. En particulier, ce patron met en jeu les deux arcs E_t et E_f qui résument en grande partie le fonctionnement du <while>. L'arc E_t, dont le prédicat $pr = C$, conditionne le passage au nœud s_A et donc l'exécution de la sous-activité A, tandis que l'arc E_f, dont le prédicat $pr = !C$, conditionne le passage au nœud f_X et donc la fin d'exécution du <while>.

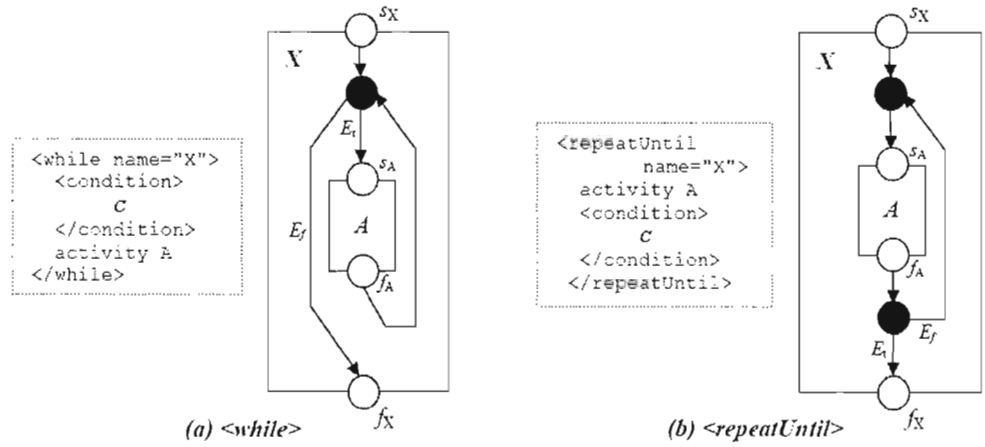


Figure 4.4 Les activités `<while>` et `<repeatUntil>`.

L'activité <repeatUntil> permet une exécution itérative de sa sous-activité A, et cela tant que sa condition C est évaluée à faux à la fin de chaque itération. Cependant, une fois cette condition bascule à vrai (i.e. C = vrai), la sous-activité A ne doit plus s'exécuter et le `<repeatUntil>` prend fin. La figure 4.4(b) représente le patron traduisant le `<repeatUntil>` en B-CFG. En particulier, ce patron enveloppe les deux arcs E_f et E_t : le premier, dont le $pr = !C$, conditionne l'exécution de la sous-activité A, alors que le deuxième, dont le $pr = C$, donne le feu vert pour mettre plutôt fin à l'exécution du `<repeatUntil>`. Il faut noter que la sous-activité d'un `<repeatUntil>` est exécutée *au moins* une fois, contrairement à celle d'un `<while>` qui pourrait ne l'être même pas une fois (i.e., sa condition C est fausse dès la première évaluation).

Il est d'usage, dans le domaine de génération de test, d'adopter le critère *0-1 répétition* pour une performance répétitive d'un flot de contrôle (e.g. performance répétitive incarnée par un `<while>`). Pour le présent travail, nous avons prévu des patrons adaptés à ce critère qui seront donnés au chapitre suivant (voir sous-section 5.3.4).

4.6.1.2.3 L'activité <flow>

L'activité `<flow>` exprime de la concurrence et de la synchronisation (1). Toutes les activités qui y sont directement enchâssées peuvent être concurremment exécutées, et une fois que leur exécution s'achève, elles sont reconduites à un point de synchronisation. De plus, à travers les liens de contrôle qui y sont déclarés, le `<flow>` permet aussi de définir des dépendances de synchronisation entre les sous-activités qui y sont enchâssées à différents niveaux de profondeur.

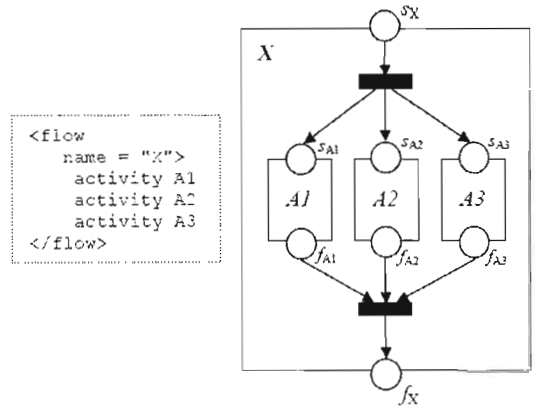


Figure 4.5 L'activité `<flow>`.

La figure 4.5 représente le patron de traduction du <flow> en B-CFG. Dans ce patron, la concurrence est représentée par un nœud de bifurcation, alors que le point de synchronisation est représenté par un nœud de jointure. Les liens implicites que la sémantique du <flow> implique sont rendus explicites en faisant usage des arcs normaux. Cependant, la traduction des liens de contrôle qui seraient déclarés dans un <flow> (ce qui n'est pas le cas pour le code de la figure 4.5) sera plutôt abordée à la section 4.6.3. Des règles de traduction, qui sont liées rien qu'à la présence de ces liens de contrôles, seront néanmoins évoquées dans la présente section 4.6.1 ainsi que dans celle lui succédant, la section 4.6.2.

4.6.1.2.4 L'activité <forEach>

Elle permet d'exécuter l'activité `<scope>` qui y est enchâssée pour un nombre de fois n, avec $n = (fcvExp - scvExp) + 1$ et où *fcvExp* et *scvExp* correspondent respectivement aux expressions désignées par les éléments `<finalCounterValue>` et `<startCounterValue>`. Ces deux expressions sont évaluées lorsque l'exécution du `<forEach>` est déclenchée, et demeurent constantes pour toute la durée de cette exécution. Au fait, le `<forEach>` est une activité au visage double : elle pourrait définir un cadre pour une exécution *séquentielle* des n éventuelles instances de son activité `<scope>`, comme elle pourrait aussi définir un cadre pour une exécution *concurrentielle* de ces n instances, et cela dépendamment de la valeur de son attribut *parallel*.

Ci-dessous (figure 4.6), sont fournis les deux patrons de traduction du <forEach>, pour les cas séquentiel et parallèle. Les deux patrons sont dotés des arcs E_X et E'_X. L'arc E_X, dont le prédicat $pr = (fcvExp \geq scvExp)$, donnerait éventuellement le feu vert à l'exécution de

l'activité <scope> pour un certain nombre de fois, tandis que l'arc E'_X, dont le prédicat $pr = (fcvExp < scvExp)$, conduit au nœud final f_X et conditionne ainsi la fin du <forEach>.

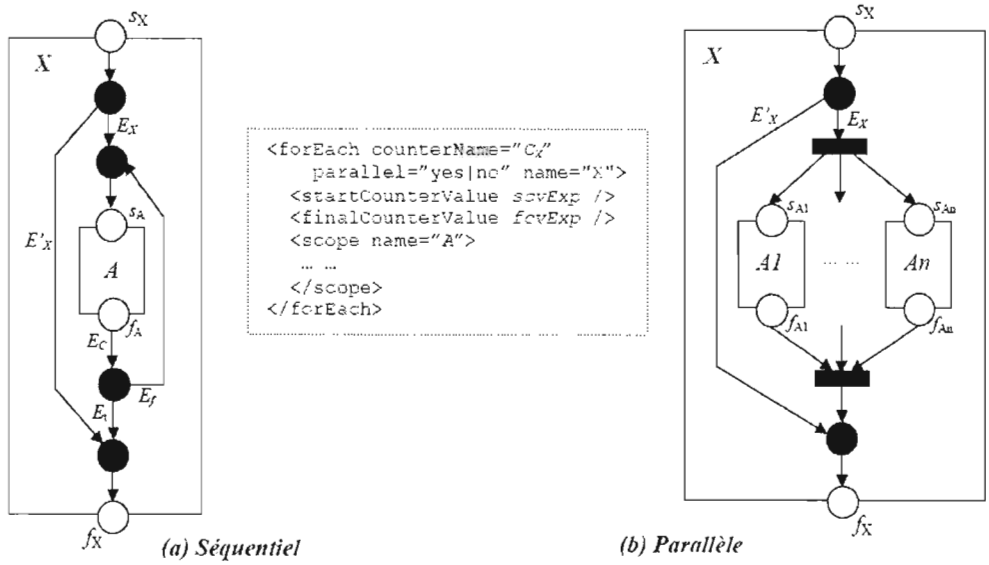


Figure 4.6 L'activité `<forEach>`.

Pour un `<forEach>` séquentiel (figure 4.6(a)), nous avons rendu explicite la déclaration implicite de la variable qui, se trouvant définie au sein de l'activité `<scope>` dudit `<forEach>`, est dédiée au comptage du nombre d'itération atteint. Cette variable, portant le nom spécifié par l'attribut *counterName* (i.e. C_X), fera partie des variables globales de l'ensemble $\boldsymbol{V}$ du graphe **B**-CFG résultant. Elle est initialisée à la valeur `<startCounterValue>` et est incrémentée à chaque fin d'itération, suite à une exécution de l'action ac = $(C_X{+}{+})$ attachée à l'arc E_C. Une fois l'incrémentation est faite, le prédicat de l'arc E_t (i.e. $pr = (C_X > n)$) ainsi que celui de l'arc E_f (i.e. $pr = (C_X \leq n)$) sont testés pour décider si, respectivement, l'activité `<forEach>` prendrait fin ou bien si une autre itération serait allouée.

Pour un `<forEach>` parallèle (figure 4.6(b)), l'un des points majeurs qui pourraient brouiller sa traduction en **B**-CFG est l'introduction du *parallélisme dynamique*. Autrement dit, la norme BPEL précise que le nombre des branches parallèles à exécuter dans un tel `<forEach>` n'est pas connu à *l'avance*, alors que paradoxalement pour pouvoir définir le patron de traduction dudit `<forEach>`, nous avons besoin de connaître ce nombre à l'avance.

Dans le cas où les deux expressions *fcvExp* et *scvExp* correspondent à deux constantes ou que leur évaluation consiste à décortiquer des expressions *XPath* simples, il serait possible

qu'une analyse statique soit appliquée sur le processus à tester afin de calculer ce nombre, et cela à l'image de ce que propose Lohmann (30) pour sa traduction du BPEL en réseau de Petri. Cependant, Lohmann fait savoir que s'il s'agit d'expressions *XPath* complexes, cette analyse risque d'être indécidable, et que dans ce cas il faut définir une borne supérieure pour le nombre de branches (i.e. instances) à exécuter. Pour nous, définir cette borne sera équivalent à définir le nouveau critère de couverture *0-n instances* que nous avons considéré pour le test des processus BPEL incluant des <forEach> parallèles. À vrai dire, c'est la variante *0-2 instances* de ce critère qui sera décrite dans ce qui suit (voir 5.3.4). Finalement, pour le cas où le nombre des branches parallèles à exécuter est reçu comme un message, Lohmann (30) conseille l'adoption de quelques travaux existants (40) pour le calcul de ce nombre.

L'activité `<forEach>`, qu'elle soit séquentielle ou parallèle, pourrait être munie de son élément optionnel `<completionCondition>`. Pour le cas d'un <forEach> séquentiel qui serait muni de cet élément, le patron de traduction lui correspondant est fourni à la section B.1 (Annexe).

4.6.1.2.5 Les activités <pick> et <if>

L'activité <pick> attend l'occurrence d'un événement parmi plusieurs, pour n'activer qu'une seule branche parmi plusieurs. Chaque branche est une paire d'événement-activité. Les événements attendus pourraient provenir aussi bien de l'extérieur (i.e. des *message events* pour les branches `<onMessage>`) que du moteur d'exécution lui-même (i.e. des *timing events* pour les branches `<onAlarm>`). Si une branche est sélectionnée suite à l'avènement de son événement déclencheur, l'activité en faisant patrie est exécutée.

La figure 4.7(a) expose le patron de traduction du `<pick>` en **B**-CFG. En particulier, ce patron est doté d'un nœud de décision incarnant un choix exclusif de la branche à exécuter. Ce noeud dispose d'arcs sortants (e.g. E_1, E_2 et E_3) dont les éléments *ac* et *pr* seront définis dépendamment à ce que ces arcs seraient associés à un `<onMessage>` ou à un `<onAlarm>`. Un arc conduisant à l'exécution de l'activité incorporée par un `<onAlarm>` (e.g. E_2 à A2) gardera la valeur par défaut *Null* pour ses deux éléments *ac* et *pr*. Cependant, un arc conduisant à l'exécution de l'activité incorporée par un `<onMessage>` (i.e. E_1 à A1) gardera, certes, la valeur *Null* pour son élément *pr*, mais verra son action correspondre à un enregistrement de valeurs (i.e., $ac = Enreg$), valeurs transmises par le message reçu. Les prédicats des arcs E_i ($i = 1, \ldots, 3$) sont mis à *Null* car la sélection d'une branche d'un <pick · ne dépend pas du résultat d'évaluation

d'une expression booléenne explicite, mais de l'occurrence ou non d'un événement[12]. Dans notre cas, cela correspond donc à un choix *non-déterministe* de la branche à exécuter.

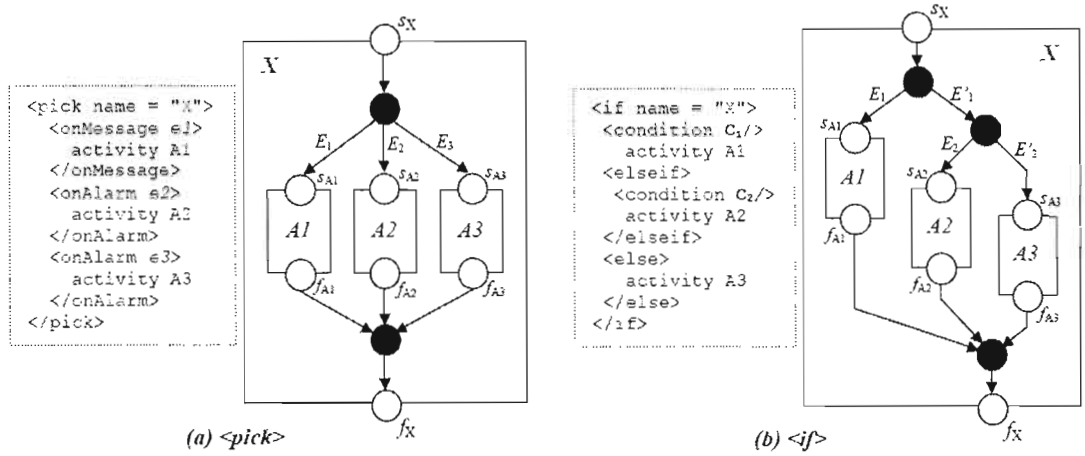


Figure 4.7 Les activités `<pick>` et `<if>`.

L'activité <if> consiste en une liste *ordonnée* d'une ou plusieurs branches conditionnelles. La branche considérée comme pièce maîtresse de cette liste est définie par l'élément `<if>`. Cette branche pourrait, éventuellement, être suivie par d'autres branches qui seraient définies par d'optionnels éléments `<elseif>`, accompagnés, le cas échéant, de l'élément optionnel `<else>`. Seule l'activité de la branche dont la condition tient en *premier* sera allouée à s'exécuter. Or, si aucune condition d'aucune branche ne tient, c'est la branche <else> qui verra son activité exécutée, dans le cas où une telle branche est définie.

La façon dont l'activité `<if>` est définie nous a conduit à considérer un modèle *en cascade* pour sa traduction en B-CFG. Au fait, le patron de traduction de cette activité, décrit par la figure 4.7(b), est un sous-graphe B-CFG qui pourrait être vu comme un arbre binaire dont les feuilles convergent vers un même noeud de fusion. Chacune de ces feuilles correspond à une activité Ai définie par l'une des branches conditionnelles de l'activité `<if>` traduite. L'ensemble de ces feuilles est desservi par des arcs sortant de noeuds de décision, pour finalement rejoindre un même noeud de fusion. La profondeur (i.e. nombre de niveaux) de cet arbre est équivalue au nombre des éléments `<if>` et `<elseif>` présents dans l'activité `<if>` considérée. Chaque arc E_i (e.g., $i = 1, \ldots, 2$ pour le cas de la figure 4.7(b)) est doté d'un prédicat $pr = C_i$, tandis que

[12]L'occurrence des événements est liée à l'aspect temporel qui ne peut être représenté dans un graphe CFG tel que le nôtre.

son rival l'arc E'_i est doté d'un prédicat qui est une négation du sien (i.e., $pr = !C_i$), où C_i est l'expression booléenne conditionnant l'exécution de l'activité Ai définie par la $i^{ème}$ branche.

À l'exception des autres activités et leur façon de s'exécuter, l'exécution des activités conditionnelles <if> et <pick> ne peut être limitée qu'à l'exécution de l'activité Ai de la branche sélectionnée. En effet, cette exécution devra aussi considérer le cas où les activités, des autres branches non sélectionnées, seraient cibles ou sources de liens de contrôle[13]. Au fait, afin de maintenir les dépendances de contrôle du processus traduit, et ainsi se conformer à la norme BPEL et sa DPE (voir 1.4.3.3), les liens de contrôle sortant des activités des branches non-sélectionnées devront être mis à l'état *dead path*. Pour ce faire, nous avons choisi de doter les patrons de traduction de telles activités d'un *chemin de saut*. Outre la mise à l'état *dead path* des liens de contrôle sortant de ces activités, ce chemin permettra aussi, le cas échéant, de donner en amont la chance à leurs liens de contrôle entrants d'être synchronisés et d'être par la suite reconduits vers le nœud de fin de saut de l'activité (e.g. le nœud f'_{A31} de la figure 4.8).

Nous nous sommes, certes, engagés à maintenir les dépendances de contrôle de tout processus BPEL traduit en **B**-CFG, mais la démarche que nous entreprenons pour le faire diffère de celle proposée par la norme BPEL. Ladite norme exige, pour le maintien de ces dépendances, une application *systématique* de la mise à l'état *dead path* des liens de contrôle sortants de *toute* activité, faisant partie d'un <pick> ou d'un <if>, et n'ayant pas été exécutée. Contrairement à cela, notre démarche pour cette mise à l'état *dead path* sera *sélective*. Autrement dit, les activités n'ayant pas été exécutées, et qui seront concernées par cette mise à l'état *dead path* de leurs liens de contrôle sortants, seront sélectionnées. Par conséquent, juste les patrons des activités sélectionnées seront dotés d'un chemin de saut. Outre cela, notre démarche prévoit aussi de doter les activités qui sont cibles de liens de contrôle par des chemins de saut. Ces activités seront elles aussi sélectionnées. Plus de détails sur cette question des chemins de saut sont fournis par la section suivante 4.6.2.

Par définition et application directe de cette démarche sélective, nous avons choisi de ne considérer les chemins de saut dans un <pick> (ou dans un <if>) que si *au moins* l'une des activités de ses branches est cible ou source de liens de contrôle *dépassant* ses frontières, frontières

[13]Toute dépendance de contrôle (i.e. lien de contrôle) définie entre deux activités faisant partie de deux branches distinctes d'une même activité conditionnelle X (e.g. dépendance qui relierait les activités A1 et A2 de la figure 4.8) ne sera pas prise en compte. Une dépendance pareille n'a aucun intérêt conceptuel et ne serait pas envisagé pour une définition sensée d'un processus BPEL donné.

du <pick> (resp. du <if>) enveloppant. Ce choix a été adopté pour deux raisons. Premièrement, nous avons jugé que dans le cas contraire[14] la sémantique de saut sera inutile vu qu'aucune des dépendances de contrôle, définie à l'extérieur du <if> ou du <pick> concerné, ne sera perturbée ou violée. Deuxièmement, ce choix permettra de simplifier d'une façon considérable les graphe B-CFG incluant de tels <pick> et <if>.

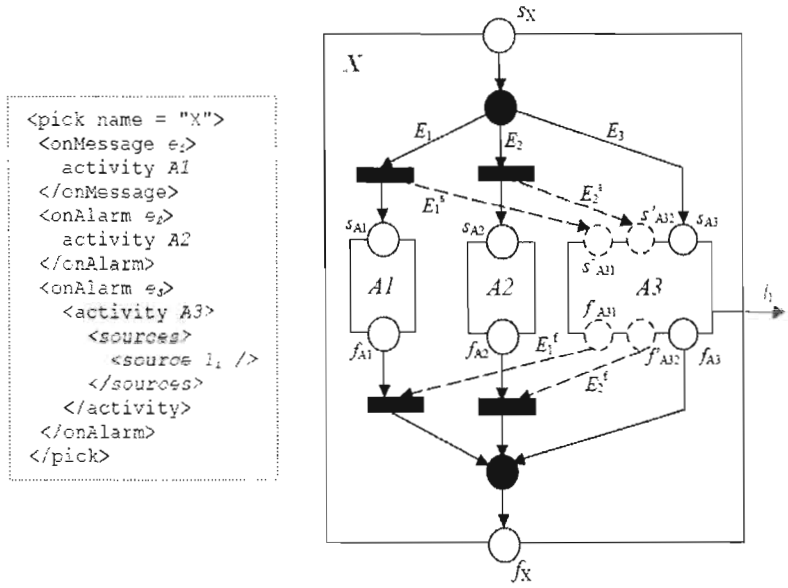


Figure 4.8 Les chemins de saut dans le patron d'un <pick>.

La figure 4.8 présente le patron de traduction d'une activité conditionnelle <pick> dont l'une des branches enveloppe une activité qui est source d'un lien de contrôle dépassant les frontières de ce <pick>. Conformément à ce qui a été décrit plus haut, l'activité $A3$ qui est source de ce lien doit être sautée si l'activité $A1$ ou $A2$, faisant respectivement partie de la $1^{ère}$ et $2^{ème}$ branche, est exécutée. Cela est nécessaire pour ne plus laisser le lien de contrôle[15] l_1, sortant de l'activité $A3$, à un état non déterminé *unset*, mais plutôt le mettre à l'état *dead path*. Ainsi, nous saurons maintenir les dépendances de contrôle du processus traduit en dotant l'activité $A3$ d'un chemin de saut. En revanche, si par exemple l'activité $A3$ correspondait à un

[14]Le cas où les activités des branches d'un <pick> ou d'un <if> sont cibles ou sources de liens de contrôle dont *aucun* ne dépasse les frontières de ce <if> ou de ce <pick> enveloppant.

[15]La flèche représentant le lien de contrôle l_1 dans la figure 4.8 n'est pas une traduction de ce lien en B-CFG, mais juste un moyen pour souligner sa présence comme lien sortant de l'activité $A3$. Pour la traduction d'un tel lien voir 4.6.3.

<flow> enveloppant deux sous-activités B_1 et B_2 qui seraient inter-liées par le lien de contrôle l_1, dans ce cas aucun chemin de saut ne sera considéré dans le <pick> enveloppant, car il n'y aura plus de liens de contrôle dépassant les frontières de ce <pick>. Cependant, pour un tel cas de figure, la norme BPEL aurait toujours exigé la considération de la mise à l'état *dead path* du lien l_1, et donc un recours aux chemins de saut.

Afin de matérialiser le maintien des dépendances de contrôle dans la figure 4.8, l'activité A3 a été dotée d'un chemin de saut, en sus d'un nœud de bifurcation qui a été associé à chacune des $1^{ère}$ et $2^{ème}$ branches. L'un des arcs sortants de ce nœud est un arc de saut (e.g. E_1^s). De plus, chacune de ces $1^{ère}$ et $2^{ème}$ branches a été dotée d'un nœud de jointure afin de montrer que ce <pick> ne prendra fin que si l'activité, de la branche étant sélectionnée parmi ces deux branches, aurait terminée son exécution, et que les activités des autres branches qui sont appelées à être sautées l'auraient été effectivement (e.g. l'activité *A3* de la $3^{ème}$ branche). Ce nœud joindra l'arc normal sortant de la branche sélectionnée (e.g. celui sortant de f_{A1}) à ceux de saut émanant du reste des branches non sélectionnées et dont les activités sont à sauter (e.g. E_1^f).

Les mêmes règles de saut s'appliquent aussi pour le cas où une actvité, faisant partie d'une branche, est *cible* de liens de contrôle provenant de l'extérieur du <pick> l'enveloppant. De même, nous aurions pu prendre pour exemple l'activité <if> à la place du <pick> vu que ces règles de saut s'appliquent de la même façon pour ces deux activités.

Les nœuds s'_{A31} et s'_{A32} marquent le début de la sémantique de saut dont l'activité *A3* est dotée, alors que les nœuds f'_{A31} et f'_{A32} en marquent la fin. Les deux nœuds s'_{A31} et s'_{A32}, comme leurs congénères f'_{A31} et f'_{A32}, auraient pu être fusionnés, et par conséquent les deux arcs E_1^s et E_2^s auraient partagé le même nœud de destination, de même que les arcs E_1^f et E_2^f auraient partagé le même nœud source. Cependant, si ce partage des nœuds source et destination était adopté, nous aurions été contraints, lors de la génération des chemins de test, de vérifier si un arc de saut entrant E_i^s et un arc de saut sortant E_j^f peuvent cohabiter dans le même chemin sans en faire un chemin de test infaisable (e.g. un chemin incluant l'arcs E_1^s et l'arc E_2^f, à la place de l'arc E_1^f, est infaisable).

Comme règle générale, nous avons donc choisi d'éviter cette vérification en optant pour cette alternative qui consiste à copier le chemin de saut d'une activité Ai en plusieurs chemins similaires mais distincts, et ayant pour chacun d'eux un nœud de début s'_{Aij} et de fin f'_{Aij}. Le nombre de ces chemins (i.e. nombre des nœuds s'_{Aij} et celui des nœuds f'_{Aij}), joints à l'activité principale Ai d'une $i^{ème}$ branche, est égal au nombre des branches présentes dans le <pick>

englobant, réduit d'une unité. Cette alternative qui rend les graphes **B**-CFG plus explicites et ainsi évite une telle vérification, nous l'avons aussi adoptée pour séparer les chemins de saut des chemins normaux en faisant joindre au nœud de début s_{Ai} et de fin f_{Ai} de toute activité Ai à sauter, un ou des nœuds s'_{Aij} et f'_{Aij}. En définitive, cette façon de faire va nous permettre de diminuer d'une façon considérable le nombre des chemins infaisables.

4.6.2 Saut des activités

Dans cette section, nous présentons les patrons *augmentés* d'un cercle restreint d'activités structurées. Ces patrons seront augmentés par les chemins de saut dont ils seront dotés pour tenir compte des cas où les activités, qu'ils représentent, seraient susceptibles d'être sautées[16]. En fait, le cercle des activités dont les patrons augmentés seront présentés dans cette section se réduit à celles enchâssant des sous-activités qui sont cibles ou sources de liens de contrôle, mais qui ne sont pas elles-mêmes sources ou cibles de tels liens.

4.6.2.1 Les cas de considération des chemins de saut

Comme nous l'avons indiqué précédemment, nous ne considérons les chemins de saut pour des activités, étant enveloppées par un `<pick>` ou par un `<if>`, que s'il se trouve qu'*au moins* l'une de ces activités est cible ou source de liens de contrôle *dépassant* les frontières du `<pick>` ou du `<if>` l'enveloppant. Cette même condition[17] s'appliquera aussi pour les autres cas de figure où des activités, se trouvant enveloppées dans des constructeurs BPEL particuliers, seraient susceptibles d'êtres sautées. En fait, dans un constructeur A – constructeur qui correspond à un `<pick>`, à un `<if>`, à un `<scope>`, à un `<fault/terminationHandler>` ou à une activité structurée qui risque d'être sautée ou non-exécutée –, nous ne considérons les chemins de saut pour les activités y étant enchâssées que si au moins l'une de ces activités :

1. Est elle-même cible ou source de liens de contrôle dépassant les frontières du constructeur A l'enveloppant.

---

[16] Les cas où une activité serait susceptible d'être sautée sont énumérés dans la section 4.2.2.

[17] Condition que nous avons considérée en nous inspirant de celle imposée par la norme BPEL dans le cas où une erreur surviendrait lors de l'évaluation de la `<transitionCondition>` d'un lien de contrôle L. Dans une telle situation, si l'activité source du lien L fait partie d'un `<scope>` S, alors que son activité cible se trouve à l'extérieur de ce `<scope>`, son état sera mis à *dead path*. Par contre, si et l'activité source et l'activité cible du lien L se trouvent dans le même `<scope>` S, son état sera sans importance et peut donc être laissé à l'état *unset*.

2. Elle correspond à une activité structurée qui enchâsse des sous-activités étant cibles ou sources de liens de contrôle qui dépassent et ses frontières et les frontières du constructeur A l'enveloppant. La figure 4.9 fournit un exemple d'une telle situation. Cette figure décrit le patron augmenté d'une activité X qui correspond à un `<flow>` enveloppé dans un `<if>` Y, et ayant des sous-activités A_i dont une est source d'un lien de contrôle qui est supposé dépasser et les frontières du `<flow>` et les frontières du `<if>` enveloppant.

3. Elle – comme activité structurée – et ses sous-activités sont cibles ou sources de liens de contrôle qui dépassent les frontières du constructeur A les enveloppant. Ce troisième cas est une combinaison des deux premiers cas.

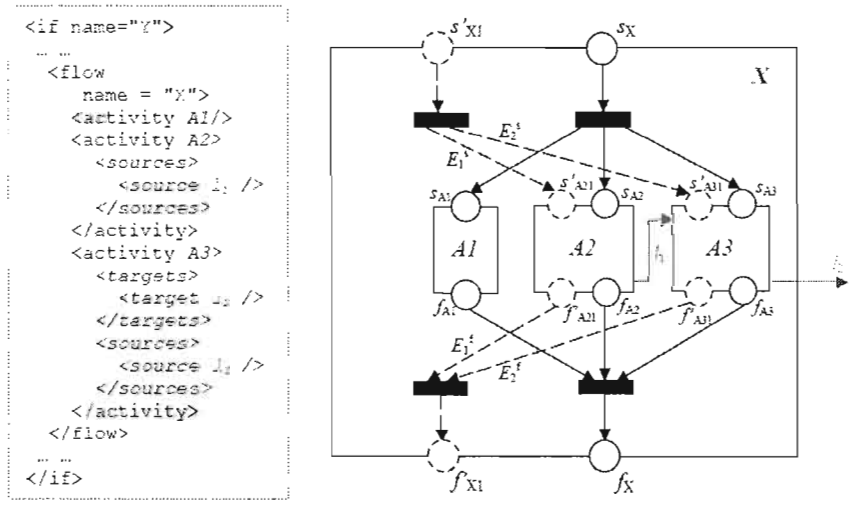


Figure 4.9 Saut de l'activité `<flow>`.

Dans la présente section (i.e., 4.6.2), seul le $2^{ème}$ cas sera détaillé, alors que les 1^{er} et $3^{ème}$ cas seront, certes, eux aussi considérés, mais ne seront détaillés que plus tard, dans la section 4.6.3.5. Les patrons augmentés qui seront donc présentés dans ce paragraphe ne concernent que les activités qui, étant de nature structurée et susceptibles d'être sautées, enchâssent au moins une sous-activité qui est cible ou source de liens de contrôle dépassant leurs frontières.

4.6.2.2 Définition de l'algorithme de sélection

Dans un constructeur A – constructeur qui correspond à un `<pick>`, à un `<if>`, à un `<scope>`, à un `<fault/terminationHandler>` ou à une activité structurée qui risque d'être sautée ou non-exécutée –, les activités qui seraient à sauter, et donc qui verront leurs patrons

dotés de chemins de saut, seront sélectionnées. Cela se conforme à la démarche *sélective* décrite plus haut. En fait, la sélection des activités B qui seraient à sauter dans un tel constructeur A se fera selon les étapes décrites par l'algorithme de *sélection* donné ci-dessous :

$E_{\grave{A}-sauter}$ selectionnerActivitesASauter(constructeur A)

1. Repérer les activités qui sont ***directement*** enveloppées dans le constructeur A et qui sont i) soit elles-mêmes cibles ou sources de liens de contrôle traversant les frontières de A, ii) soit ayant des sous-activités qui sont sources ou cibles de tels liens[18]. Si aucune activité n'est repérée, la méthode prend fin.
2. Mettre les activités repérées à l'étape précédente dans l'ensemble $E_{\grave{A}-traiter}$
3. Pour chaque activité $B \in E_{\grave{A}-traiter}$, faire :
   (a) Ajouter B dans l'ensemble $E_{\grave{A}-sauter}$
   (b) Repérer toutes les activités B^s qui sont ***directement*** enveloppées dans le constructeur A et qui sont i) soit elles-mêmes sources de liens de contrôle ayant pour cible l'activité B, ii) soit ayant des sous-activités qui sont sources de tels liens.

   De même, repérer toutes les activités B^t qui sont ***directement*** enveloppées dans le constructeur A et qui sont i) soit elles-mêmes cibles de liens de contrôle ayant comme source l'activité B, ii) soit ayant des sous-activités qui sont cibles de tels liens. Si aucune activité B^s ou B^t n'est repérée, passer à 3(d).
   (c) Ajouter les activités B^s et/ou B^t, repérées à la sous-étape précédente, dans l'ensemble $E_{\grave{A}-traiter}$. Cependant, avant l'ajout de chacune de ces activités à l'ensembles $E_{\grave{A}-traiter}$, s'assurer que l'activité à ajouter ne fait pas déjà partie ni de cet ensembles ni de l'ensemble $E_{\grave{A}-sauter}$.
   (d) Retrancher la sous-activité B de l'ensemble $E_{\grave{A}-traiter}$.
4. Si $E_{\grave{A}-sauter}$ contient des activités structurées :
   (a) Soit $E_{AS-\grave{a}-traiter}$ l'ensemble de ces activités structurées contenues dans $E_{\grave{A}-sauter}$
   (b) Pour chaque activité B $\in E_{AS-\grave{a}-traiter}$, faire :

$$E_{\grave{A}-sauter} = E_{\grave{A}-sauter} \bigcup \text{selectionnerActivitesASauter(B)}$$

[18] e.g. La sous-activité *A3* de la figure 4.9 est source du lien l_2 qui traverse les frontières et de l'activité X l'enveloppant et du constructeur Y enveloppant le tout.

L'algorithme présenté est défini comme une méthode récursive. Cette méthode prend comme argument un constructeur A pour retourner un ensemble $E_{\grave{A}-sauter}$ contenant les activités sélectionnées qui sont enchâssées à différents niveaux dans A et qui verront leurs patrons de traduction respectifs dotés de chemins de saut. En effet, la méthode commence par sélectionner parmi les activités qui sont *directement* enchâssées (se trouvant au premier niveau de profondeur) dans le constructeur A. Les activités sélectionnées à ce premier niveau correspondent à celles qui sont repérées à l'étape (1) et à celles qui, ayant une dépendance de contrôle avec des activités préalablement sélectionnées, sont repérées à l'étape 3(b). En dernier lieu, la méthode s'applique d'une façon récursive sur les éventuelles activités structurées de l'ensemble $E_{\grave{A}-sauter}$ dans l'optique de sélectionner d'autres activités se trouvant à d'autres niveaux de profondeur.

4.6.2.3 Exemples d'application de l'algorithme de sélection

L'application de l'algorithme de sélection sur l'exemple du code BPEL relaté par la figure 4.9 se fera au moins en deux temps :

- Premièrement, en considérant l'activité Y comme le constructeur A enveloppant (situation où A correspond à un `<pick>`), l'activité X y étant enchâssée sera sélectionnée. Cette sélection aura lieu à l'étape **1** de l'algorithme, suite à un premier appel à ce dernier. Le `<flow>` X sera sélectionné car il enveloppe une sous-activité $A3$ qui est source d'un lien de contrôle l_2 dépassant ses frontières et les frontières du constructeur Y. Éventuellement, d'autres activités qui sont directement enchâssées dans le constructeur Y pourraient être sélectionnées dans le cadre de ce premier appel.
- Deuxièmement, il est certain que nous aurons besoin d'au moins un deuxième appel à l'algorithme. Cet appel s'appliquera récursivement sur l'activité structurée X étant sélectionnée au début du premier appel dudit algorithme. Au cours du deuxième appel, le `<flow>` X sera pris pour le constructeur A enveloppant (situation où A correspond à une activité X qui risque d'être sautée) et l'activité $A3$ y étant enchâssée sera sélectionnée. La sélection de l'activité $A3$ aura lieu à l'étape **1** de ce deuxième appel de l'algorithme, car elle est source d'un lien de contrôle l_1 qui dépasse les frontières du constructeur X l'enveloppant. En sus, l'activité $A2$ sera elle aussi sélectionnée, mais pas à la même étape. Cette sélection de $A2$ aura lieu à la sous-étape 3(b) de ce deuxième appel et sera donc due à une dépendance de contrôle qui la lie avec l'activité $A3$. Cette dépendance se traduit par le lien l_1 dont l'activité $A2$ est source et ayant pour cible l'activité $A3$.

Au final, les appels récursifs à l'algorithme de sélection, appliqués à l'exemple du code BPEL relaté par la figure 4.9, mèneront au moins à la sélection des activités X, $A2$ et $A3$. D'ailleurs, comme le montre la figure en question, les patrons de traduction de chacune de ces activités seront dotés d'un chemin de saut.

Un autre exemple d'application de l'algorithme de sélection est donné par la figure 4.10. Cet exemple expose un <pick> qui risque d'être sauté comme un tout, et qui voit donc son patron de traduction doté d'un chemin de saut — chemin qui commence par le noeud s'_{X1} pour ensuite converger vers le noeud f'_{X1}. Le <pick> concerné enveloppe trois activités dont : une première, l'activité $A1$, est cible d'un lien de contrôle traversant les frontières de ce <pick> enveloppant, et une troisième, l'activité $A3$, qui est source d'un lien traversant ces mêmes frontières. Partant de ce fait, les deux activités $A1$ et $A3$ sont sélectionnées, avec le <pick> enveloppant, pour que leurs patrons respectifs soient dotés d'un chemin de saut. De plus, il est à noter que le saut des deux activités $A1$ et $A3$ se fera en parallèle.

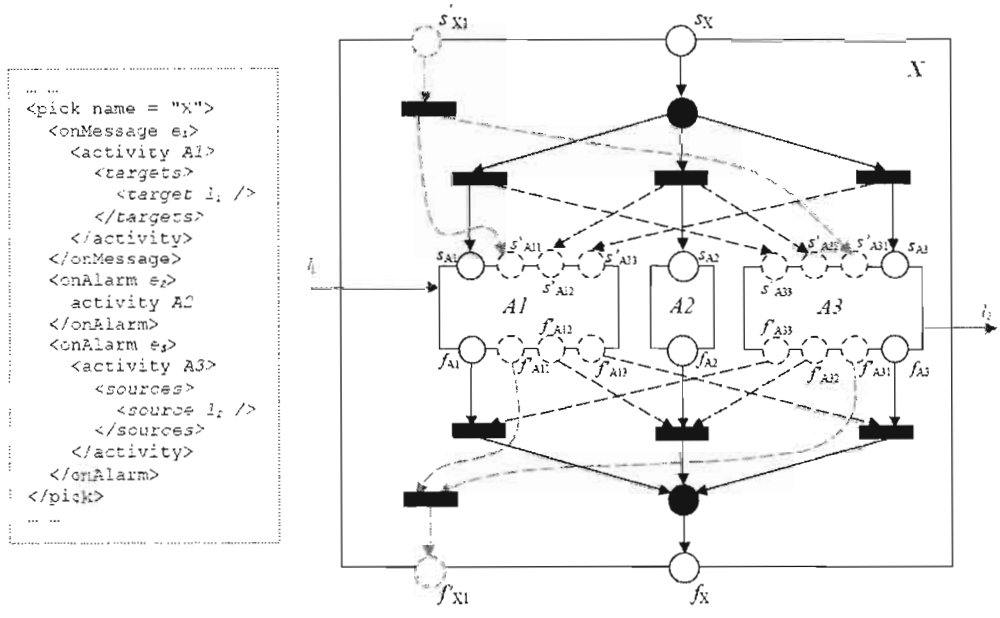


Figure 4.10 Saut de l'activité <pick>.

Par ailleurs, pour chacun des patrons des deux activités $A1$ et $A3$, le chemin de saut sera copié en plusieurs chemins similaires ayant pour chacun un nœud de début s'_{Aij} et de fin f'_{Aij}. Comme expliqué plus haut, cela s'applique car la même sémantique de saut, qui est associée aux patrons des deux activités $A1$ et $A3$, pourrait être déclenchée suite à différents scénarios

d'exécution — scénarios à ne pas confondre pour ne pas résulter en des chemins infaisables. Ce qui s'applique comme règles pour le saut d'un <pick> est aussi valable pour le saut d'un <if>.

Une autre application de l'algorithme de sélection est présentée à la figure 4.11. Cette figure expose un <sequence> qui risque d'être sauté, et qui voit donc son patron de traduction doté d'un chemin de saut. Ce <sequence> enveloppe trois activités dont deux, l'activité $A1$ et $A3$, sont sélectionnées pour que leurs patrons respectifs soient eux aussi dotés d'un chemin de saut. La sélection de ces deux activités a lieu pour les mêmes raisons pour lesquelles les activité $A1$ et $A3$ du <pick> précédent ont été sélectionnées. Même la façon de sauter ces activités est semblable à celle appliquée pour le <pick> précédent : un saut *parallèle* des activités sélectionnées.

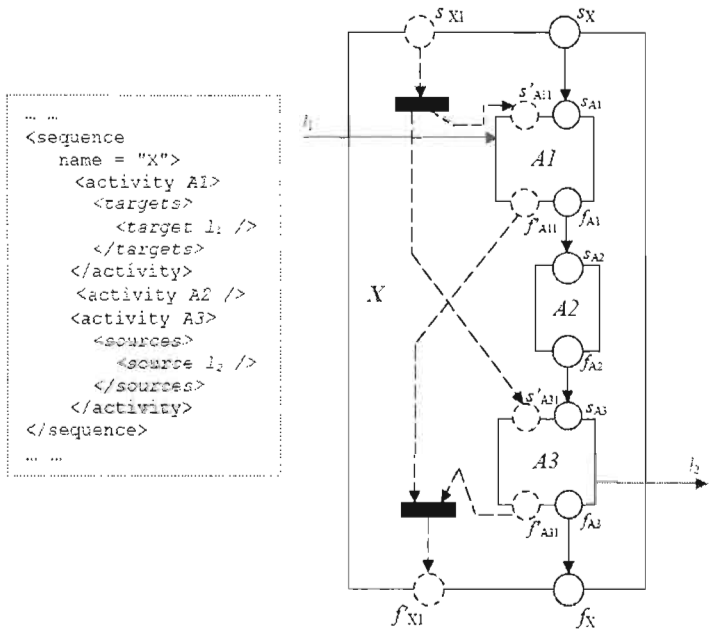


Figure 4.11 Saut de l'activité <sequence>.

Ce saut parallèle des activités se trouvant enveloppées dans un tel <sequence> laisse sous-entendre un semblant de contraste avec l'essence même du constructeur <sequence>, essence d'un constructeur qui contraint les activités y étant enchâssées à s'exécuter *séquentiellement*. Or, dans le cas d'un saut, il ne s'agit point d'exécuter les activités enveloppées (i.e. exécuter leurs actions), mais plutôt de les sauter (i.e. sauter leurs actions pour n'entreprendre que la mise de leurs liens de contrôle à l'état *dead path*) afin de conserver les dépendances de contrôle du code BPEL traduit. Par conséquent, ce qui importe le plus c'est de sauter les activitées qui,

étant enveloppées dans un <sequence>, sont cibles ou sources de liens de contrôle, et non pas se soucier dans quel ordre elles devront l'être.

Pour ce faire, nous avons choisi de faire usage de l'algorithme de sélection pour désigner les activités qui seraient à sauter et afin de rendre possible le saut parallèle de ces dernières. Un tel saut nous évitera de penser à comment relier séquentiellement les activités désignées jusqu'à arriver au noeud de fin de saut du <sequence> enveloppant (e.g., dans la figure 4.11, relier, respectivement, les noeuds s'_{Aij} et f'_{Aij} des activités $A1$ et $A3$ à un noeud de bifurcation et à un autre de jointure est plus systématique que de procéder à relier s'_{X1} à s'_{A11}, f'_{A11} à s'_{A31} et enfin f'_{A31} à f'_{X1}). Mieux encore, ce saut parallèle nous évitera de penser à tout sauter dans un <sequence> rien que pour pouvoir relier ce tout par un chemin de saut séquentiel jusqu'à arriver à son noeud final (e.g., éviter le cas dans lequel le patron de l'activité $A2$ sera contraint à être doté d'un chemin de saut rien que pour pouvoir définir un chemin de saut global qui se veut séquentiel, allant de s'_{X1} jusqu'à f'_{X1} en passant par les activités $A1$, $A2$ et $A3$).

La démarche décrite ci-dessus n'est pas fortuite, mais a été adoptée pour que nous puissions faire d'une pierre deux coups, à savoir :

1. Réduire la présence des chemins de saut dans un graphe **B**-CFG qui, par conséquent, verra sa taille globale réduite d'une façon considérable (e.g. dans la figure 4.11, le patron de l'activité $A2$ n'a pas été doté de chemin de saut). Réduire la présence de ces chemins est équivalant à réduire la taille du code mort (i.e. code non-exécuté) que ces chemins représentent dans un graphe B-CFG, un code mort qui, comme le souligne Lohmann (32), constitue la raison principale qui rend la taille d'un modèle représentant du code BPEL si grande.
2. Éviter de prévoir, pour un certain nombre d'activités (e.g., les activités de base et répétitives qui ne sont pas elles-mêmes source ou cible de liens de contrôle), de définir des patrons de traduction qui sont à doter de chemins de saut (e.g., il n'y a pas besoin de doter le patron de traduction, présenté à la sous-section 4.6.1.1, d'un chemin de saut, ou de définir un nouveau qui en soit doté).

Contrairement à notre démarche, l'équipe d'Ouyang (48) a choisi de tout sauter dans un <sequence>. Il est vrai que cela, d'une part, facilite sa traduction des liens de contrôle et en particulier leur mise à l'état *dead path* (i.e., aucune sélection des activités n'est à envisager), et que, d'autre part, cela fait en sorte que son saut d'activités épouse l'essence du <sequence> en ayant lieu d'une façon séquentielle. Cependant, ce choix entrepris par ladite équipe mènerait à

des réseaux de Petri de taille assez importante et complexe, comme il la contraindra de prévoir, pour *toute* activité BPEL, des patrons augmentés de chemins de saut. Par ailleurs, on trouve que Stahl (56), certes, adopte un saut parallèle, mais il n'évoque nulle part dans son rapport la question de sélection des activités à sauter.

4.6.2.4 Discussion

Dans un spectre plus large, dépassant celui du `<sequence>` pour englober tous les constructeurs BPEL, ne fallait-il pas, pour cette question du maintien des dépendances de contrôle, opter à la base pour une mise *directe* des liens de contrôle à l'état *dead path*, à la place de passer par un saut récursif des activités pour aboutir à une telle mise à pareil état? Autrement dit, fallait-il vraiment arriver au stade de devoir faire un choix de sélectionner ou pas les activités qui sont à sauter, voire même un choix entre un saut séquentiel et un autre parallèle? Dans ce qui suit, nous nous expliquons vis-à-vis de ces questions en comparant nos choix à ceux des autres.

Dans la littérature, on trouve qu'Ouyang (48) et son équipe suivent religieusement la norme BPEL, en optant pour un saut récursif dans l'optique de parvenir à mettre les liens de contrôle, sortant des activités à sauter, à l'état *dead path*. Cependant, cela a pour résultat d'aboutir à des graphes de grande taille et, en particulier, à des problèmes d'explosion d'états vu que le saut des activités (i.e., considération du code mort représenté) et l'exécution des activités non-sautées (i.e., considération du code à exécuter) s'entrelacent (30; 33). De plus, et comme il est mentionné plus haut, l'équipe d'Ouyang prévoit même de doter les patrons des activités, n'ayant pas de liens de contrôle sortants ou entrants, de chemins de saut, ce qui mènerait à des graphes de taille encore plus importante et où les problèmes d'explosion d'états auraient plus de chance à se manifester.

De son côté, Lohmann (30) a voulu remédier à ces problèmes d'étendue de taille et d'explosion d'états en proposant une mise *directe* des liens de contrôle à l'état *dead path*. Pour ce faire, il a eu recours à une « *overapproximation* » du comportement exact du processus à traduire. Certes, cette *overapproximation* a porté fruit, mais a malheureusement engendré de nouveaux défauts (du moins vis-à-vis de notre objectif de test), à savoir :

- L'émergence de scénarios d'exécution additionnels qui sont modélisés dans le graphe résultant, alors qu'ils sont inexistants dans le code BPEL représenté par ce graphe. Dans un cadre de test comme le nôtre (test boîte blanche), de tels scénarios ne peuvent être acceptés pour deux raisons principales :

1. Lors de la génération des chemins de test, on risque de tomber sur des chemins scénarios qui ne correspondraient à rien dans le code, ce qui biaiserait notre résultat du test. Au fait, l'essence même d'un test boîte blanche, c'est de tout fonder sur le code source à tester. Cela nous contraint donc à devoir représenter ce code par un modèle qui lui est fidèle pour qu'on n'ait, en aucun cas, l'impression que c'est un autre code qui est testé.
2. Lohmann (30; 33) lui-même fait savoir que, suite à l'application d'une *overapproximation*, il est possible que le graphe résultant contienne des erreurs qui ne sont pas présentes dans le processus BPEL traduit. Chose qui reviendrait à générer des chemins scénarios révélant des erreurs qui n'existent même pas dans le code concerné! Lohmann propose de procéder à une analyse statique du processus à tester afin de pouvoir détecter ces erreurs, tandis qu'à la base, tout cela aurait pu être évité.

- La mise de côté de la règle qui, dictée par la norme BPEL, consiste à devoir attendre à ce que tous les liens de contrôle entrants, des activités à sauter, soient évaluées avant qu'on procède à la mise à l'état *dead path* de leurs liens sortants. En effet, Lohmann a choisi de ne pas attendre une telle évaluation en mettant, immédiatement et d'un trait, tous ces liens sortants à l'état *dead path*. Cela aurait pour conséquences, dans le réseau de Petri simulé par Lohman, de laisser la chance à des jetons de toujours y figurer même après l'atteinte de la place finale de ce réseau. Autrement dit, sa traduction d'un processus BPEL en un réseau de Petri pourrait résulter en un modèle qui serait privé de la propriété «*soundness*»[19]. Dans le cadre de notre travail, si nous procédons de la sorte, nous risquerions de générer des chemins de test dont des composantes seraient incapables de converger et donc d'atteindre le noeud final du graphe **B**-CFG parcouru (voir sous-section 4.6.3.2).
- La perte de la mise *naturelle* des liens de contrôle à l'état *dead path* — une mise qui est basée sur un saut récursif des activités qui sont sources de liens à mettre à pareil état. Un tel saut récursif a l'avantage de rendre possible à un constructeur BPEL de propager, par délégation, la mise à l'état *dead path* aux sous-constructeurs qui y sont enchâssés, de façon à ce que ces derniers délèguent à leurs tours cette mise à pareil état aux autres sous-constructeurs qu'ils enveloppent. Cela est plus naturel qu'une mise à l'état *dead path* qui serait centralisée et qui nous contraindra à réinventer la roue et trouver une solution

[19]Dans ce cas-ci, nous parlons de la propriété «*soundness*» qui est vérifiée dans l'optique de voir si les réseaux de Petri générés auraient des *terminaisons propres* (sans aucun jeton, sauf dans la place finale) ou pas, et non pas dans l'optique de voir s'il y aurait ou pas un *deadlock* dans ces réseaux, voire des *tâches irréalisables*.

(e.g., une *overapproximation*) pour la mettre en oeuvre, sans pour autant être à l'abri des effets secondaires qu'elle pourrait induire.

Notre démarche, par rapport à cette question critique de la mise des liens de contrôles (sortant des activités à sauter) à l'état *dead path*, consiste à considérer et concilier les apports des approches des deux équipes, celle d'Ouyang et celle de Lohmann. Cette conciliation se fera tout en évitant les problèmes que chacune des deux approches pourrait engendrer.

Dans notre démarche, nous veillons, d'une part, à remédier aux problèmes d'étendue de taille et d'explosion d'états dans les graphes résultants, tout en évitant une *overapproximation* qui pourrait engendrer de nouveaux défauts (e.g., l'émergence de nouveaux scénarios inexistants dans le code, mais présents dans le graphe le représentant). Pour ce faire, nous faisons usage de notre algorithme de sélection qui nous permettra de réduire la présence des chemins de saut (i.e., présence du code mort) dans les graphes résultants.

D'autre part, notre démarche conserve un saut récursif des activités qui est plus naturel, et qui aide à respecter la règle de la norme BPEL contraignant à l'attente de l'évaluation des liens de contrôle entrants avant la mise de leurs congénères sortants à l'état *dead path*. Un tel saut se fera de manière à ce que nous ne serons pas amener à tout sauter, et cela en sélectionnant les activités concernées par ce saut — saut qui se fera toujours de façon parallèle. Ainsi, nous ne passerons pas d'un extrême (i.e., doter tous les patrons de toutes les activités de chemins de saut) à l'autre (i.e., saut direct des activités sans prêter attention à l'une des règles de la norme BPEL et à la fidélité requise au graphe représentant le code traduit), mais nous croyons avoir arrivé à un compromis.

Toujours en ce qui concerne cette question critique du maintien des dépendances de contrôle par la mise des liens de contrôle (sortant des activité à sauter) à l'état *dead path*, aucune considération et aucun traitement ne lui ont été consacrés par Yuan (81) et Li (26). Que ce soit dans le travail de Yuan ou celui de Li (qui en est la suite), on parle justement du cas de la mise d'un lien de contrôle à l'état *dead path* suite à une évaluation négative de sa <transitionCondition>, et aussi de l'effet d'une DPE. En revanche, les deux auteurs et leurs équipes n'ont (nulle part dans leurs articles) ni mentionné ni traité le cas où « les liens de contrôles sortants, d'une activité B enchâssée dans une activité A, devront être mis à l'état *dead path* si durant la performance de l'activité A (e.g., <if>), la sémantique de cette dernière impose à ce que sa sous-activité B ne fera pas partie de ses sous-activités à exécuter» (1). La non considération de

ce cas de figure, et donc d'un volet important de la question du maintien des dépendances de contrôle, pourrait mener à des graphes qui ne reflèteront pas d'une façon fidèle le code BPEL traduit, ce qui biaiserait le résultat de test.

Par ailleurs, le fait de ne pas prendre en compte ce volet important de la question du maintien des dépendances de contrôle influence la traduction de Yuan et de Li de sorte à ce qu'elle semble résulter en des graphes de taille réduite, alors qu'on sait bien à quel point cette taille dépend de la façon avec laquelle ledit volet (i.e., la présence du code mort) est traité, s'il n'est pas ignoré. Outre cela, ce qui donnerait aussi l'impression que la taille des graphes des deux auteurs serait réduite est l'absence d'un souci d'uniformisation pour leur traduction. Contrairement à la nôtre, leur traduction n'est pas basée sur des patrons (des sous-graphes prédéfinis) partageant une forme commune et qui n'ont qu'à s'interconnecter pour constituer un graphe globale représentant le processus traduit. Une troisième raison qui serait derrière la taille réduite des graphes des deux auteurs est le fait qu'ils n'avaient pas à trop détailler leur traduction puisque le type de test visé était un test boîte grise qui ne nécessite pas une représentation détaillé du code à tester.

Comme nous l'avons mentionné plus haut, grâce à l'adoption de l'algorithme de sélection, nous ne sommes pas contraints à tout sauter et donc à prévoir, pour toutes les activités BPEL, un patron de traduction qui soit doté d'un chemin de saut. Parmi les activités dont les patrons seront exonérés de cette dotation, il y a les constructeurs répétitifs (i.e., `<while>`, `<repeatUntil>` et le `<forEach>` séquentiel) qui ne sont pas eux-mêmes sources ou cibles de liens de contrôle. Cela est dû au fait qu'aucune activité faisant partie de l'un de ces constructeurs répétitifs ne peut être sélectionnée pour voir son parton être doté d'un chemin de saut, puisque la norme BPEL interdit à une telle activité d'avoir des liens de contrôle sortants ou entrants qui dépassent les frontières du constructeur répétitif l'enveloppant.

À l'exception des patrons de traduction proposés pour les activités répétitives et de base, les patrons de traduction exposés par les deux sous-sections 4.6.1 et 4.6.2 ont été taillés pour des activités structurées dont le nombre des sous-activités est égal à trois. Cependant, rien n'empêche à ce que ces patrons soient étendus pour des activités structurées enveloppant un nombre de sous-activités supérieur ou inférieur à trois.

4.6.3 Liens de contrôle

L'activité `<flow>`, qui rend possible une exécution concurrente à ses sous-activités, permet aussi de définir des dépendances de synchronisation entre ces dernières. Comme nous l'avons indiqué précédemment, ces dépendances sont définies à travers des liens de contrôlés qui sont à déclarer dans le `<flow>` enveloppant. Dans cette section, il est question de cette catégorie de liens et de leur traduction.

Dans la section 4.2.2, nous avons fait savoir qu'un lien de contrôle peut être mis principalement à l'un des deux états : *exécuté* ou *dead path*. Partant de ce fait, la représentation des liens de contrôle en **B**-CFG se fera en traduisant chacun de ces liens par deux arcs : un arc normal reflétant le cas où le lien serait à l'état exécuté, et un arc mort reflétant le cas où ce même lien serait à l'état *dead path*. Le seul cas où un lien de contrôle ne sera traduit que par un seul arc (arc normal) correspond à celui où la `<transitionCondition>` du lien est omise, où son activité source n'est pas susceptible d'être sautée et où cette dernière se retrouve avec un *suppressJoinFailure* qui vaut "no".

La traduction imagée des liens de contrôle, par des arcs normaux et morts, sera complétée en associant à chacun de ces liens une variable globale $\boldsymbol{es}$ (i.e., *edge status*) qui décrit son état. Une telle variable prendra toujours, comme valeur d'initialisation, la valeur *unset*. Les deux autres valeurs que cette variable pourrait prendre sont les valeurs *true* et *false* qui sont respectivement associées aux deux états : *exécuté* et *dead path*. Finalement, il faut rappeler que toute variable $\boldsymbol{es}$ fera partie du deuxième sous-ensemble de l'ensemble $\boldsymbol{V}$ — ensemble des variables globales du graphe résultant.

Comme illustré dans la figure 4.12, chaque lien de contrôle l_i sera représenté par un arc normal E_{li} et un arc mort E'_{li}, avec : pour l'arc E_{li}, $\boldsymbol{ac} = (es_{li} = true)$, alors que pour l'arc E'_{li} $\boldsymbol{ac} = (es_{li} = false)$. Ladite figure dépeint le patron de traduction *générique* qui s'applique pour toute activité BPEL X qui est source et cible de liens de contrôle. Le patron en question est constitué de trois composantes :

- La composante T_X qui renferme un sous-graphe traduisant le conteneur <targets> de l'activité X (e.g., dans la figure 4.12, ce conteneur accueille k liens entrants).
- La composante S_X qui renferme un sous-graphe traduisant le conteneur <sources> de l'activité X (e.g., dans la figure 4.12, ce conteneur enveloppe n liens sortants).
- Une composante centrale qui renferme un sous-graphe mettant au premier plan le coeur

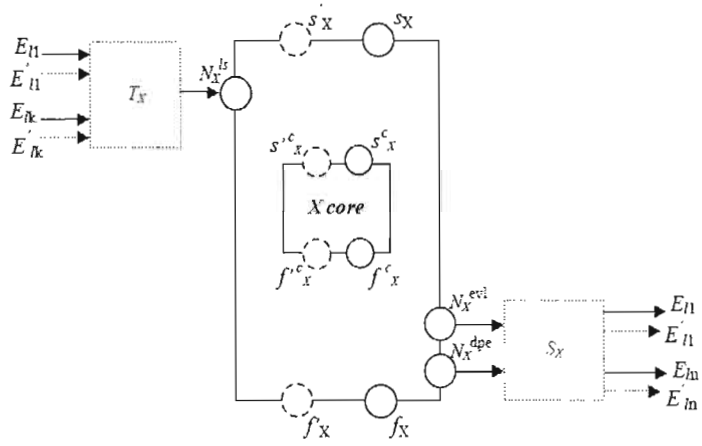


Figure 4.12 Patron générique pour les activités sources et cibles de liens de contrôle.

de l'activité X, tout en dévoilant l'impact des attributs et éléments standard de cette dernière sur son comportement. En particulier, le coeur de l'activité X est représenté par la sous-composante X *core*. Cette sous-composante n'est en fait rien d'autre que la traduction de l'activité X sans prise en compte de ses liens de contrôle sortants et entrants, *i.e.*, une activité X qui serait traduite comme si elle ne disposait pas de liens entrants et sortants, alors qu'elle en a. Au fait, dépendamment de quel type l'activité X serait, la sous-composante X *core* sera remplacée par l'un des patrons donnés à la section 4.6.1 — e.g., si l'activité X correspondrait à une activité de base disposant de liens entrants et sortants, son X *core* sera remplacé par le patron donné à la figure 4.2. Les noeuds s'^c_X et f'^c_X, qui marquent le début et la fin d'un chemin de saut, ne seront considérés que si X correspondrait à une activité structurée dont une ou plusieurs sous-activités sont à sauter.

Les deux composantes T_X et S_X du patron générique résultant sont liées avec sa composante centrale à travers les noeuds N_X^{ls}, N_X^{evl} et N_X^{dpe}. Ces noeuds constituent donc des interfaces entre la composante centrale et les deux autres composantes adjacentes. Le contenu de chacune de ces trois composantes sera dévoilé dans les sous-sections qui suivent.

4.6.3.1 Traduction du conteneur <targets>

Le conteneur `<targets>` d'une activité X énumère les liens de contrôle dont cette activité est cible – liens de contrôle incarnés par des éléments `<target>`. Optionnellement, ce conteneur pourrait aussi inclure l'élément `<joinCondition>`. La traduction de cet élément sera reportée

pour être abordée lors du traitement de la composante centrale (voir 4.6.3.2). Dans la présente sous-section, l'emphase n'est mis donc que sur les éléments <target> du conteneur <targets>.

La figure 4.13 dépeint le patron de traduction d'un conteneur <targets>. Le conteneur en question accueille k liens de contrôle. Chacun de ces liens est traduit par un couple d'arcs : un arc mort et un autre normal. Chaque couple est censé voir ses deux arcs converger vers un même noeud de fusion. Outre cela, un noeud de jointure tel que le noeud N_X^{syn} est nécessaire. La présence de ce noeud est vitale pour assurer le respect de la règle BPEL imposant à ce que les états de tous les liens de contrôle ayant pour cible une activité X soient déterminés avant de passer à l'évaluation de la <joinCondition> correspondante, et donc à l'exécution ou non de l'activité X concernée.

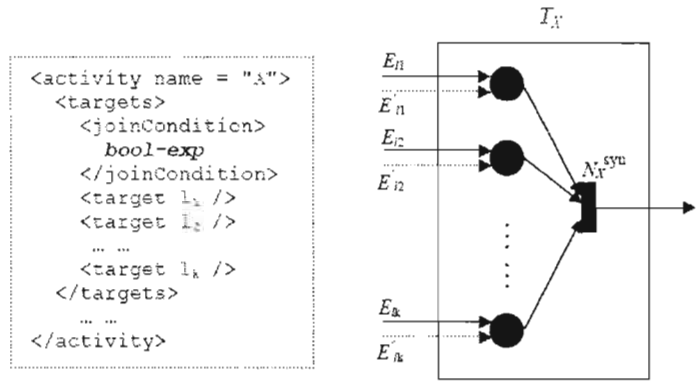


Figure 4.13 Traduction du conteneur <targets>.

Le patron donné ci-dessus pourrait être qualifié de patron générique. Toutefois, cette qualification ne nous empêche pas d'envisager d'autres patrons de traduction pour des conteneurs <targets> ayant des configurations un peu particulières (e.g., un conteneur <targets> n'accueillant qu'un seul lien). Deux des configurations particulières les plus importantes sont traitées dans l'annexe B. Par ailleurs, une autre version de la traduction des conteneurs <targets> pourrait être envisagée pour le cas d'un test exigeant une couverture du type « *toutes-les-conditions* »; une telle traduction pourrait être inspirée de celle proposée par Ouyang (48).

4.6.3.2 Traduction des activités cibles de liens de contrôle

Les scénarios d'exécution à envisager pour une activité qui est cible de liens de contrôle dépendent principalement du résultat d'évaluation de l'élément <joinCondition> qui est défini par le conteneur <targets> associé à cette activité. Cet élément se définit comme une expression

booléenne qui est exclusivement construite à base des états des liens de contrôle entrants et d'opérateurs logiques. Ainsi, le résultat d'évaluation de l'élément <joinCondition> ne dépend que des états des liens entrants. Comme exemple, la <joinCondition> JC, présentée par la figure 4.14, se définit comme suit : $JC = \$l_1\ op_1\ \$l_2\ op_2 ... op_n\ \$l_k$; où les l_i $(i = 1, \ldots, k)$ désignent les liens entrants, alors que les op_j $(j = 1, \ldots, n)$ désignent les opérateurs logiques qui les lient. En fait, l'évaluation de l'élément <joinCondition> aiguille l'exécution de l'activité le définissant vers l'un des trois scénarios d'exécution suivants :

1. Si la <joinCondition> de l'activité est évaluée à vrai, son action sera exécutée. Éventuellement, si cette même activité dispose aussi de liens de contrôle sortants (e.g. voir 4.6.3.5), les états de ces liens seront fixés dépendamment du résultat d'évaluation de leurs <transitionCondition> respectives.
2. Si la <joinCondition> de l'activité est fausse[20] et que son attribut <suppressJoinFailure> vaut "yes", alors dans ce cas son action ne sera pas exécutée mais contournée. Éventuellement, si l'activité dispose aussi de liens de contrôle sortants, tous ces liens seront mis à l'état *dead path*.
3. Si la <joinCondition> de l'activité est fausse et que son attribut <suppressJoinFailure> vaut "no", une erreur sera retournée au <scope> enveloppant pour que cette dernière soit traitée par le <faultHandler> approprié. Ce cas de figure sera abordé à la section 4.6.7.

Le patron de la figure 4.14 modélise une activité X qui est cible de k liens de contrôle et dont l'attribut <suppressJoinFailure> est à "yes". Ainsi, ce patron ne modélise que deux des trois scénarios d'exécutions énumérés ci-dessus; le premier et deuxième scénario.

En particulier, ledit patron est doté d'un noeud de décision d'où sortent deux arcs. Le premier, l'arc E_X^{exe} (*exe* pour execution), incarne le scénario dans lequel l'action (i.e. le X *core*) de l'activité traduite est exécutée suite à une évaluation positive de sa <joinCondition>. Le prédicat de ce premier arc est $\boldsymbol{pr} = JC$ et son $\boldsymbol{ac} = Null$. Quant au deuxième, l'arc E_X^{sjf} (*sjf* pour suppress join failure), il incarne le scénario dans lequel l'action de l'activité traduite n'est pas exécutée suite à une évaluation négative de sa <joinCondition>; l'exécution de l'action en question est contournée de façon à ce qu'on puisse atteindre le noeud final f_X. Le prédicat de

[20] Si la <joinCondition> d'une activité ne tient pas, l'erreur *joinFailure* a lieu. À vrai dire, l'erreur en question sera supprimée si l'attribut <suppressJoinFailure> est à "yes", tandis que si cet attribut est à "no", l'erreur sera effectivement retournée au scope enveloppant l'activité.

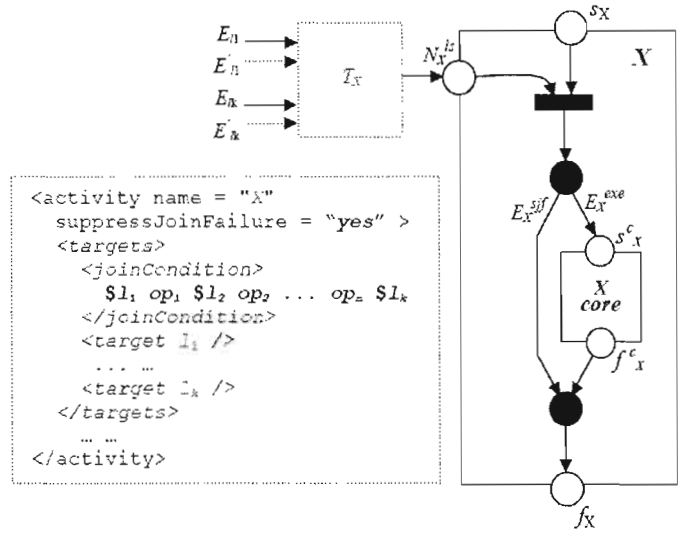


Figure 4.14 Traduction d'une activité cible de liens de contrôle.

ce deuxième arc est $\boldsymbol{pr} = !JC$ et son action $\boldsymbol{ac} = Null$. Par ailleurs, le noeud N_X^{ls} (ls pour link synchronisation), qui, jouant le rôle d'interface entre la composante principale de l'activité traduite et sa composante adjacente T_X, veille à ce que les liens de contrôle entrants soient évalués et synchronisés avant de permettre à la `<joinCondition>` d'être à son tour évaluée.

Finalement, il faut noter que, lors de la traduction en **B**-CFG, toute `<joinCondition>` JC verra ses termes $\$l_i$ $(i = 1, \ldots, k)$ remplacés par les variables es_{li} qui sont respectivement associées aux liens de contrôle l_i désignés par cette `<joinCondition>`. Ainsi, une `<joinCondition>` du genre : $JC = \$l_1$ and $\$l_2$ sera transformée en une autre du genre $JC = es_{l1}$ and es_{l2}, où es_{l1} et es_{l2} sont les variables globales qui sont associées aux deux liens de contrôle l_1 et l_2 pour décrire leurs états respectifs.

Une activité étant cible de liens de contrôle entrants, et étant définie dans un contexte où elle serait susceptible d'être sautée, devrait être traduite par un patron où une sémantique de saut est introduite. Comme l'illustre la figure 4.15, pour qu'une telle sémantique soit introduite dans ce patron, ce dernier devrait se doter d'un chemin de saut. Un chemin dont le début et la fin sont respectivement marqués par le noeud s'_X et le noeud f'_X. De plus, le flot qui sera porté par ce chemin devrait joindre celui étant suspendu au noeud N_X^{ls} afin de lui donner suite. La jointure de ces deux flots a lieu au niveau du noeud N_X^{skip}. Ce noeud matérialise la règle imposant à ce que les états de tous les liens entrants soient déterminés avant de procéder au saut de l'activité qu'ils ciblent.

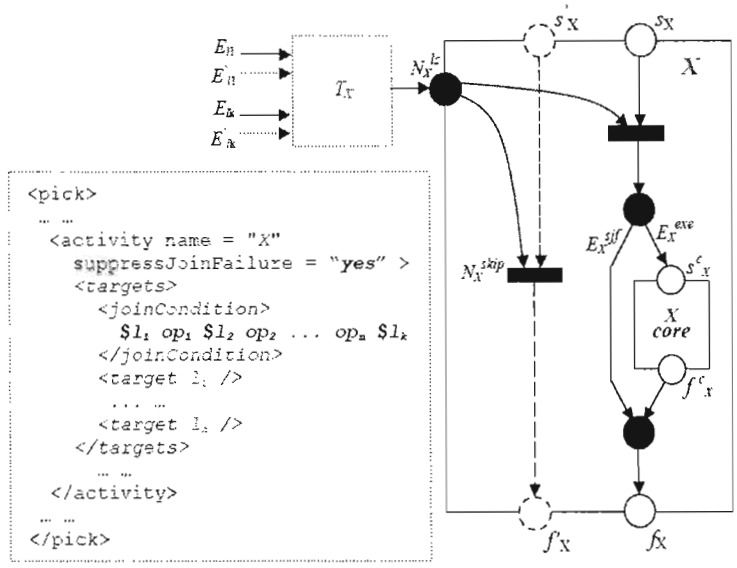


Figure 4.15 Saut d'une activité cible de liens de contrôle.

L'introduction des chemins de saut ainsi que leur séparation des chemins normaux rendent la traduction des liens de contrôle plus facile et plus explicite. L'introduction de ces chemins est aussi indispensable pour permettre une génération sensée des chemins de test.

Au fait, si une activité X, qui est cible de liens de contrôle (e.g. l'activité A de la figure 4.23), n'est jamais censée être prise en compte lors de la sélection des activités à sauter, son patron de traduction ne sera alors jamais doté d'un chemin de saut et jamais un arc de saut ne lui sera destiné. Or, il se trouve que c'est possible que cette activité X, dont le patron n'était jamais doté d'un chemin de saut, ne soit pas exécutée et que néanmoins ses liens de contrôle entrants se présentent à la composante T_X de son patron pour y être synchronisés – e.g., cas où l'activité K de la figure 4.23 sera choisie pour être exécutée, alors que le patron de l'activité A ne sera pas doté d'un chemin de saut et se retrouvera donc privé d'un arc de saut qui lui aurait été destiné. Dans un tel cas de figure, la partie du flot portée par les liens entrants de l'activité X se bloquera au noeud N_X^{ls} (e.g., N_A^{ls} de la figure 4.23) car cette activité ne sera ni exécutée ni sautée. Cela se révèle problématique et confirme ainsi la nécessité des chemins de saut pour les activités qui sont et cibles de liens de contrôle et susceptibles d'être sautées.

Cette problématique se fait constatée chez Stahl (56) comme chez Lohmann (30) puisque leurs traductions respectives ne prévoient en aucun cas de doter de chemins de saut les activités qui sont et cibles de liens de contrôle et susceptibles d'être sautées. Cela pourrait alors amener

leurs traductions à aboutir à des réseaux de Petri qui seraient privés de la propriété *soundness* — réseaux qui, lorsque simulés, laisseraient la possibilité à des jetons de toujours y figurer même après l'atteinte de la place finale. Pour notre traduction, si nous procédons de la sorte, nous risquerions de générer des chemins de test dont des composantes réussiront à converger vers le noeud final du graphe **B**-CFG parcouru, tandis que d'autres n'y arriveraient pas. D'où la nécessité d'un chemin de saut pour toute activité qui est et cible de liens de contrôle et susceptible d'être sautée. Finalement, nous rappelons que le chemin de saut dont une activité est dotée peut être dupliqué, comme c'était le cas pour l'activité $A3$ du patron relaté par la figure 4.8.

4.6.3.3 Traduction des activités sources de liens de contrôle

Une activité qui est source de lien de contrôle est une activité dont l'exécution se fait en deux temps. En premier lieu, c'est l'action constituant le coeur de cette activité qui est exécutée. Cependant, ce n'est qu'après une exécution réussie de ladite action que l'évaluation des liens sortants sera entamée. Le cas où une erreur serait déclenchée au moment de l'exécution de l'action sera traité à la section 4.6.7.

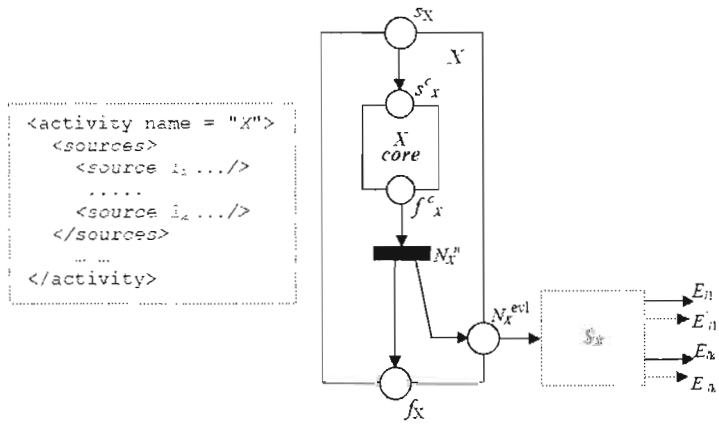


Figure 4.16 Traduction d'une activité source de liens de contrôle.

La figure 4.16 représente le patron de traduction d'une activité qui est source de k liens de contrôle. L'action à exécuter pour cette activité est incarnée par la sous-composante X *core*. À chaque fois que cette action est exécutée, le flot de contrôle est scindé en deux, par le biais du noeud de bifurcation N_X^n (n pour normal). Une première partie de ce flot rejoint le noeud final f_X, tandis que la deuxième rejoint le noeud N_X^{evl} (*evl* pour evaluation). Ce dernier joue le rôle d'interface entre la composante principale et sa composante adjacente S_X qui renferme un

sous-graphe traduisant le conteneur <sources> de l'activité X. C'est donc à travers ce noeud que l'on donne le feu vert à l'évaluation des liens de contrôle sortants.

Une activité étant source de liens de contrôle sortants, et étant définie dans un contexte où elle serait susceptible d'être sautée, devrait être traduite par un patron où une sémantique de saut est introduite. Pour qu'une telle sémantique soit introduite dans ce patron, ce dernier devrait, à l'image du patron relaté par la figure 4.15, se doter d'un chemin de saut. Comme l'illustre la figure 4.17, le début et la fin de ce chemin sont respectivement marqués par le noeud s'_X et le noeud f'_X.

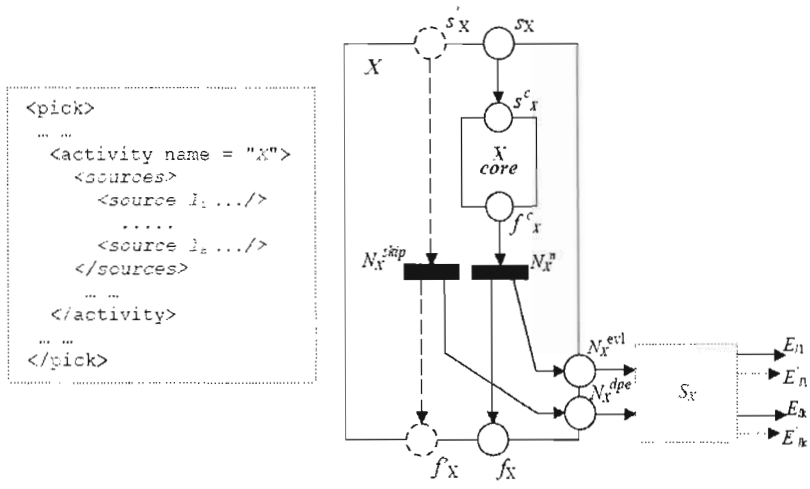


Figure 4.17 Saut d'une activité source de liens de contrôle.

Au fait, si une activité est appelée à être sautée, le flot porté par son chemin de saut sera dans ce cas scindé en deux, par le biais du noeud de bifurcation N_X^{skip}. Une première partie de ce flot rejoint le noeud de fin de saut f'_X, tandis que la deuxième rejoint le noeud N_X^{dpe} (*dpe* pour *dead path elimination*). Ce dernier noeud joue le rôle d'interface entre la composante principale et sa composante adjacente S_X de façon à véhiculer la consigne imposant la mise des liens de contrôle sortants à l'état *dead path*. À vrai dire, la présence d'une sémantique de saut, dans le patron de traduction d'une activité qui est source de liens de contrôle et susceptible d'être sautée, est nécessaire pour la mise de ses liens sortants à l'état *dead path*, et cela dans l'optique d'assurer que toutes les dépendances de contrôle seront maintenues.

4.6.3.4 Traduction du conteneur <sources>

Le conteneur <sources> d'une activité X enveloppe la déclaration des liens de contrôle dont cette activité est la source — liens de contrôle dont chacun est incarné par un élément <source>. Nous rappelons aussi, qu'optionnellement, chaque élément <source> pourrait inclure un élément <transitionCondition>. Ce dernier élément définit une condition C_i qui, parmi d'autres, régie l'état du lien de contrôle en concordance avec le résultat de son évaluation. Contrairement aux <joinConditions> qui ne peuvent être définies qu'à base des états des liens de contrôle entrants, les <transitionConditions> peuvent, quant à elles, s'exprimer à base de valeurs de variables arbitraires.

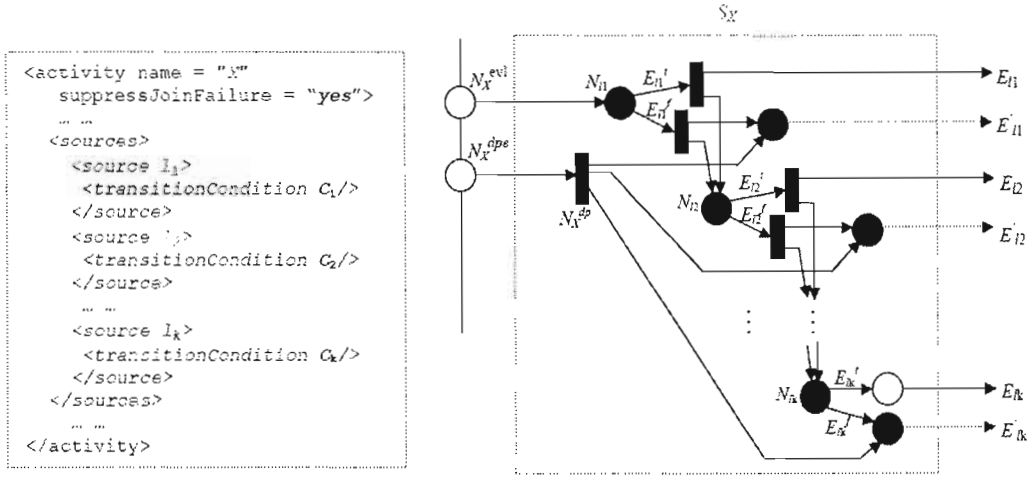


Figure 4.18 Traduction du conteneur <sources>.

La figure 4.18 dépeint le patron de traduction du conteneur <sources> d'une activité X. Ce patron respecte la norme BPEL qui contraint à ce que les <transitionConditions>, qui sont associées aux liens de contrôle enveloppés dans un contenceur <sources>, soient évaluées dans un ordre séquentiel (1). Cet ordre séquentiel est calqué sur celui dans lequel ces liens sont déclarés au sein du conteneur <sources> les enveloppant — e.g., dans la figure 4.18, la <transitionCondition> du lien l_1 devrait être évaluée avant celle du lien l_2, celle de ce dernier avant celle du lien l_3 et ainsi de suite.

Par ailleurs, à chaque lien de contrôle l_i, nous associons l'arc E^t_{li} (t pour true) dont $\boldsymbol{pr} = C_i$ et $\boldsymbol{ac} = Null$, de même que l'arc E^f_{li} (f pour false) dont $\boldsymbol{pr} = !C_i$ et $\boldsymbol{ac} = Null$. Cependant, tel que mentionné plus haut, chaque lien l_i sera principalement représenté par le

couple d'arcs (E_{li}, E'_{li}) où le premier a un $\boldsymbol{ac} = (es_{li} = true)$ et un $\boldsymbol{pr} = Null$, alors que le deuxième a un $\boldsymbol{ac} = (es_{li} = false)$ et toujours un $\boldsymbol{pr}$ qui garde la valeur $Null$. Les arcs sortant du noeud N_X^{dp} assurent, le cas échéant, la mise des liens de contrôle à l'état *dead path*; une telle mise à pareil état se fait, d'un trait, d'une façon concurrentielle, et cela contrairement à l'évaluation des `<transitionCondition>` qui se fait dans un ordre séquentiel.

Cet ordre séquentiel appliqué à l'évaluation des `<transitionCondition>` associées aux liens de contrôle est surtout nécessaire dans des cas où une erreur serait déclenchée au moment de l'évaluation de l'une de ces `<transitionCondition>`. Au fait, si une erreur a lieu lors de l'évaluation de la `<transitionCondition>` d'un lien l_j qui fait partie des k liens sortants d'une activité X, la norme BPEL exige à ce que les liens l_i, déclarés *après* le lien l_j dans le même conteneur `<sources>` (i.e., $i = (j+1)..k$), soient tous mis à l'état *dead path* ou *unset*[21]. Or, si l'évaluation des `<transitionCondition>` se faisait d'une façon concurrentielle et non pas séquentielle, il y aurait risque que ces liens l_i soient mis à l'état *exécuté* alors qu'ils sont censés être mis à l'état *dead path* ou *unset*, ce qui changerait le comportement attendu du processus traduit. C'est donc pour éliminer ce risque que le respect de cet ordre séquentiel est nécessaire.

En revanche, si l'on choisit de ne pas prendre en compte les cas où une erreur serait déclenchée au moment de l'évaluation d'une `<transitionCondition>`, le respect d'un ordre séquentiel pour l'évaluation des `<transitionCondition>` des liens passerait du nécessaire au facultatif. Dans ce cas, le patron de traduction des conteneurs `<sources>` pourrait être simplifié en optant pour une évaluation concurrentielle des états des liens et de leurs `<transitionCondition>`[22]. La figure 4.19 décrit le patron de traduction à envisager dans le cas d'une telle simplification.

Pour leur traduction des conteneurs `<sources>`, Yuan (81) et Li (26) se sont différenciés des autres auteurs par leur transformation du patron *multiple-choice*, incarné par chaque

---

[21] Dans ce cas, le lien l_j sera lui aussi mis à l'état dead path ou gardé à l'état unset, alors que les états, des liens l_r qui sont déclarés *avant* le liens l_j dans le même conteneur `<sources>` (i.e. $r = 1, . . , (j-1)$), dépendront du résultat d'évaluation des `<transitionCondition>` associées à ces liens.

[22] Étant donné que la première version de la norme BPEL (9) ne précise pas que l'évaluation des états des liens et de leurs `<transitionCondition>` doit être faite selon un ordre séquentiel, les traductions respectives de Stahl (56), de Yan (76), de Yuan (81) et de Li (26), qui, toutes ne traitent que cette première version, ont choisi l'option la plus simple, à savoir la considération d'une évaluation concurrentielle des états des liens. Cette même option a été adoptée par les traductions respectives de Lohmann (30; 31) et d'Ouyang (48), même si ces deux traductions affirment traiter la version 2.0 de la norme BPEL (1) et étaient donc toutes les deux censées respecter l'ordre séquentiel exigé par cette deuxième version pour l'évaluation des états des liens.

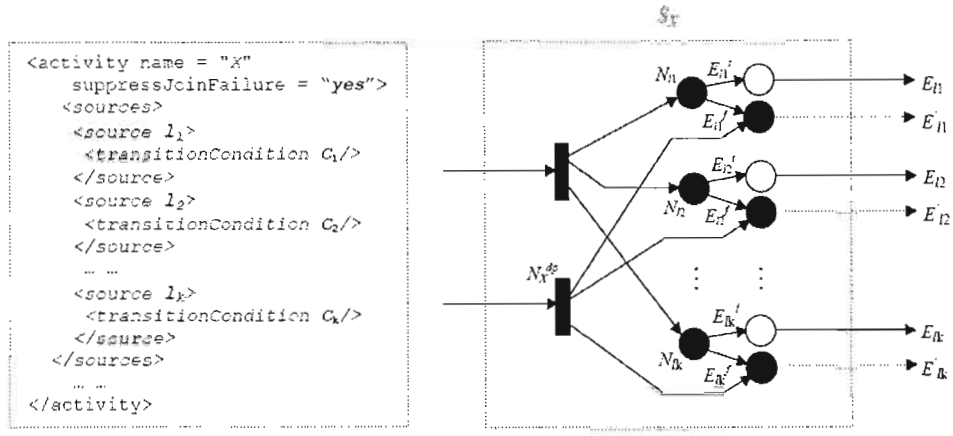


Figure 4.19 La traduction simplifiée du conteneur <sources>.

conteneur <sources> disposant de k liens (où $k > 1$), en une structure BFG de type *exclusive-choice*. Cette transformation, qui a vocation à faciliter la recherche des chemins de test, exclut l'adoption d'un ordre séquentiel pour l'évaluation des états des liens d'un même conteneur <sources>. En sus, une telle transformation risque d'induire le problème d'explosion du nombre de noeuds présents dans un graphe BFG puisque la taille de toute structure *exclusive-choice*, représentant un conteneur <sources> de k liens, est une fonction exponentielle de ce nombre k. Ce risque s'accroît si le processus traduit inclut un conteneur <sources> dont chaque lien a une <transitionCondition> qui est définie à base de données indépendantes des données à base desquelles les <transitionCondition> des autres liens, du même conteneur, sont définies.

Le patron de traduction relaté par la figure 4.18 (ou éventuellement celui de la figure 4.19) pourrait être qualifié de patron générique. Cependant, cette qualification ne nous empêche pas d'envisager d'autres patrons de traduction pour des conteneurs <sources> ayant des configurations un peu particulières (e.g., un conteneur <sources> n'enveloppant qu'un seul lien). Les configurations particulières les plus importantes sont traitées dans l'annexe B.

4.6.3.5 Traduction des activités à la fois cibles et sources de liens de contrôle

La figure 4.20 expose le patron de traduction à adopter pour le cas d'une activité qui est et source et cible de liens de contrôle. Ce patron est le résultat du mariage des deux patrons présentés par les figures 4.14 et 4.16. La seule nouveauté à noter est incarnée dans la présence du noeud N_X^{sjf}. Ce noeud permet de souligner le fait que les liens de contrôle sortant d'une activité X devront être mis à l'état *dead path* dans le cas où la <joinCondition> de cette activité est évaluée à faux et que son <suppressJoinFailure> vaut "yes".

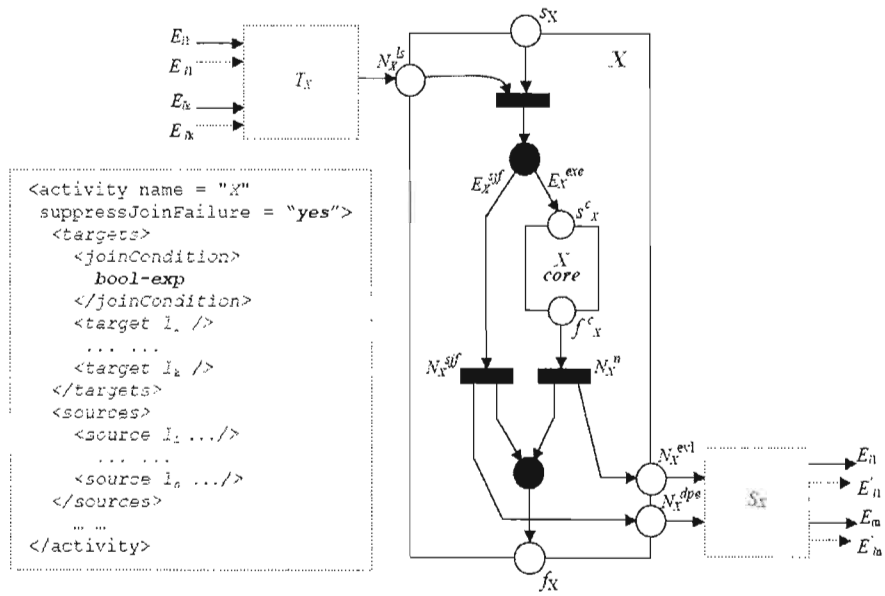


Figure 4.20 Traduction d'une activité cible et source de liens de contrôle.

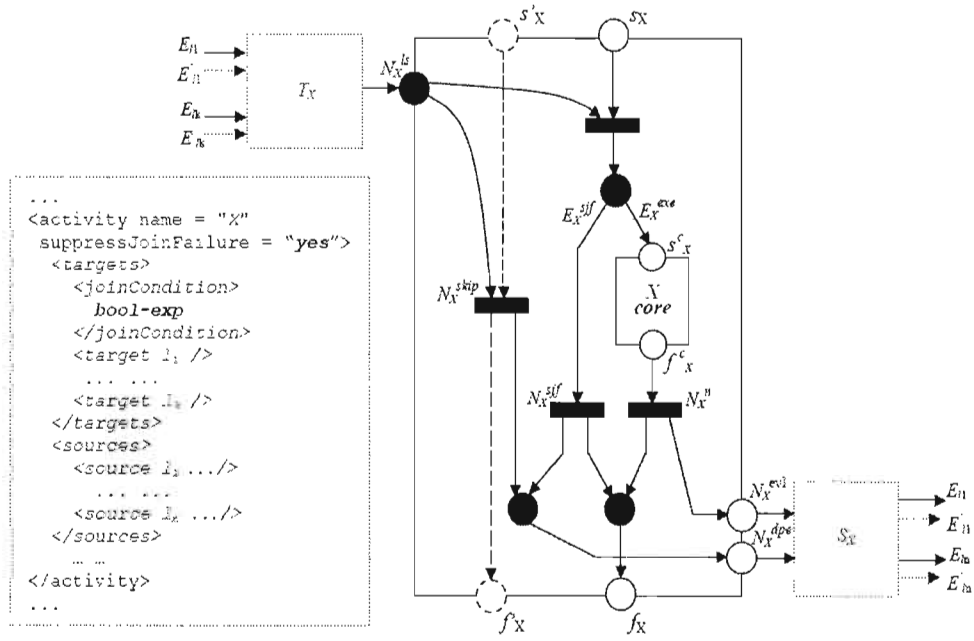


Figure 4.21 Traduction d'une activité qui est susceptible d'être sautée et qui est à la fois cible et source de liens de contrôle.

La figure 4.21 expose le patron de traduction à adopter pour le cas d'une activité qui, en plus d'être source et cible de liens de contrôle, se trouve dans un contexte où elle est susceptible d'être sautée. Ce patron est le fruit du mariage des patrons dépeints par les figures 4.15 et 4.17.

Finalement, le patron de la figure 4.22 traite le cas où X correspond à une activité *structurée* qui, en plus d'être elle-même cible et source de liens de contrôle, enchâsse des sous-activités qui sont elles aussi cibles ou sources de tels liens (e.g., les liens l^{in} et l^{out}) et qui pourraient donc être appelées à être sautées. Au fait, dans le cas où l'activité X serait à sauter ou que sa `<joinCondition>` serait évaluée à faux, ses liens de contrôle, comme ceux de ses sous-activités, devront être mis à l'état *dead path*. Pour ce faire, et comme l'illustre la figure 4.22, dans le premier cas (i.e., X est à sauter), on emprunte le chemin de saut dont le noeud de début et de fin correspondent respectivement aux noeuds s'^{c}_{X2} et f'^{c}_{X2}, alors que dans le deuxième cas (i.e., la `<joinCondition>` de X est fausse), c'est l'autre chemin, débutant au noeud s'^{c}_{X1} et se terminant au noeud f'^{c}_{X1}, qui est à emprunter. Ces deux chemins sont distincts, mais similaires.

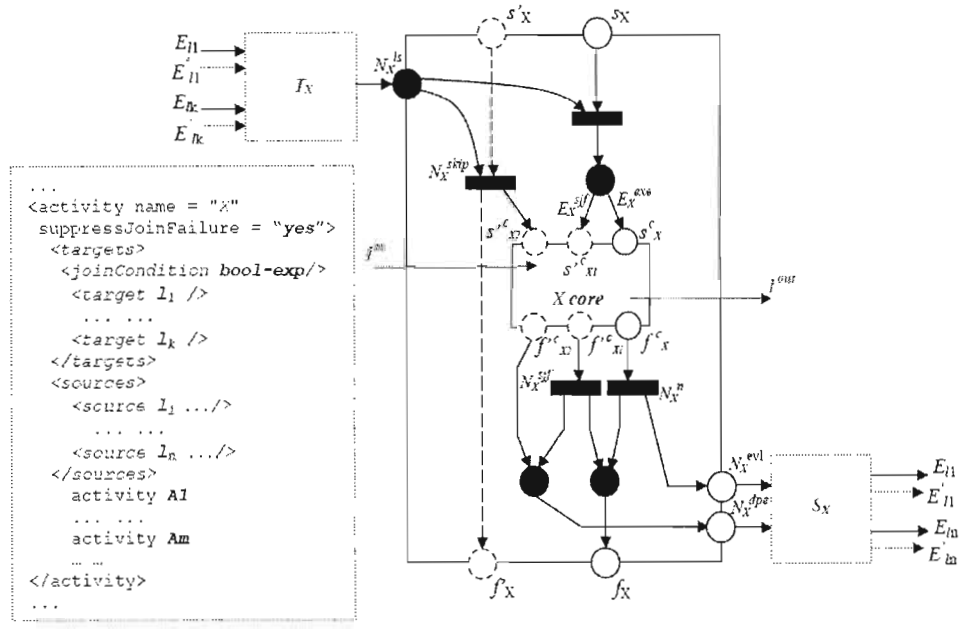
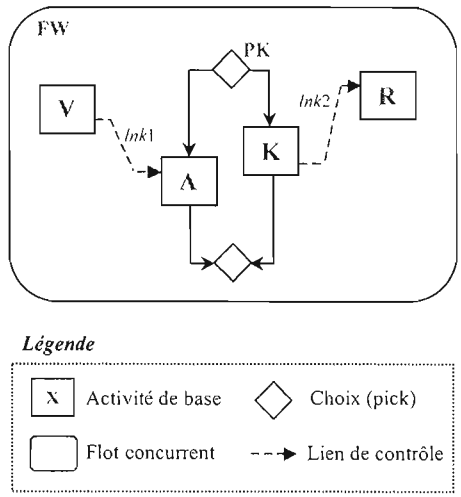


Figure 4.22 Saut des sous-activités d'une activité cible et source de liens de contrôle.

4.6.4 Exemple de traduction

Nous présentons ci-dessous le code d'un processus BPEL exemple que nous traduisons en B-CFG. La traduction de ce processus est illustrée par la figure 4.23.

```
<process name="traductionExample"
        targetNamespace=http://exemple.com
        suppressJoinFailure="yes"
                … …. …..
        xmlns="http://schemas.xmlsoap.org/ws//business-process/">

<flow name="FW" suppressJoinFailure="yes">
   <links>
     <link name="lnk1"/>
     <link name="lnk2"/>
   </links>
   <receive name="V">
      <sources> <source linkName="lnk1"/> </sources>
   </receive>
   <pick name="PK">
     <onMessage e1>
       <assign name="A">
         <targets>
           <target linkName="lnk1"/>
         </targets>
         ... ...
       <assign/>
     </onMessage>
     <onAlarm e2>
       <invoke name="K">
         <sources>
          <source linkName="lnk2"/>
            <transitionCondition C1/>
          <source/>
         </sources>
         ... ...
       </invoke>
     </onAlarm>
   </pick>
   <replay name="R">
     <targets>
       <target linkName="lnk2"/>
     </targets>
     ... ...
   </reply>
</flow>
</process>
```



Légende

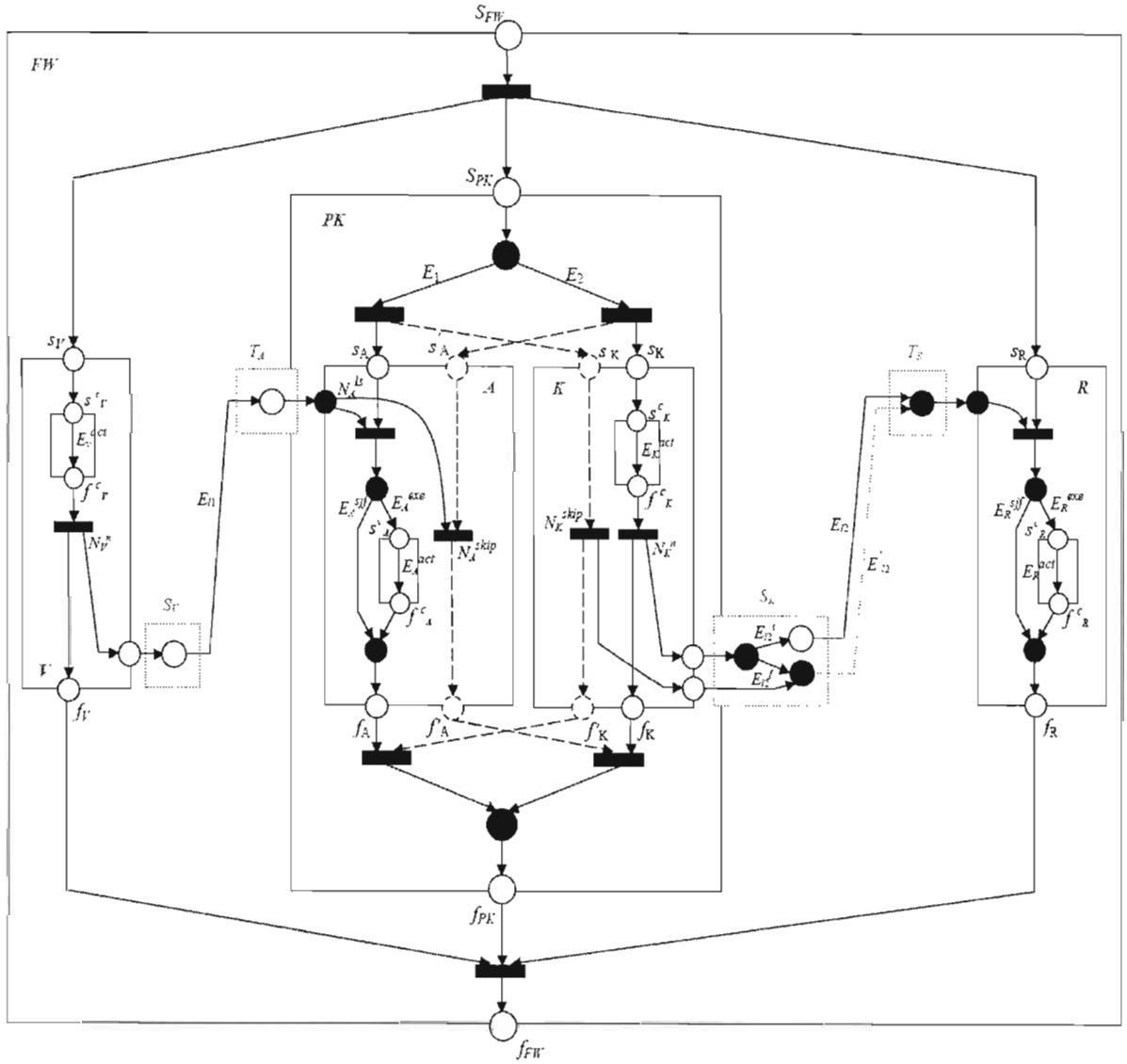


Figure 4.23 Exemple de traduction.

4.6.5 Les scopes

Dans cette section, il est question de la traduction de l'activité structurée <scope>. Cette dernière se démarque, de par la nature et le nombre d'éléments qui la composent, des autres activités structurées. Une activité <scope> est censée envelopper une activité principale A qui définit son comportement normal, comme elle est censée envelopper des éléments qui sont du moins définis par défaut (e.g., un <faultHandler>, <terminationHandler>, ou <compensationHandler>). En sus, une activité <scope> peut fournir des éléments optionnels tel qu'un <eventHandler>, des <variables> ou des <correlationSets>. Grâce aux éléments précités, un <scope> parvient donc à la définition d'un contexte d'exécution complet pour son activité principale A. Ce contexte ressemble à celui défini par le constructeur <process> qui, lui-même, est considéré implicitement comme un <scope> principale englobant le tout.

La présence des activités <scope> dans un processus BPEL sert à diviser ce dernier en plusieurs unités logiques. Chacune de ces unités est définie comme un contexte d'exécution complet. Au fait, vu que l'exécution des processus métiers correspond souvent à des transactions de longues durées, i.e., à des *LRT* (*Long-Running Transactions*), leur organisation en plusieurs unités logiques permettra donc de mieux gérer les erreurs qui ont lieu lors de telles durées et d'annuler d'une façon adéquate les transactions *ACID* (*Atomicity*, *Consistency*, *Isolation*, *Durability*) qui ont pu être validées. Ainsi, l'utilisation des <scope> est essentiellement utile pour la décomposition logique des processus BPEL dans l'optique de mieux maîtriser leur traitement d'exception.

Lors de la traduction en **B**-CFG, nous associons, à chaque <scope> traduit, une variable globale ss (i.e., *scope status*) qui décrit son état. Une telle variable sera toujours initialisée à la valeur *unset*. Dans le cas où le <scope> serait sauté, sa variable ss gardera la valeur *unset*. Cependant, si le <scope> est exécuté avec succès et donc sans qu'aucune erreur n'ait lieu (i.e. *successfully*), sa variable ss prendra la valeur *true*. Finalement, si une erreur se produit lors de l'exécution d'un <scope>, sa variable ss prendra la valeur *false* (i.e., *faulty*). Comme c'est le cas pour les variables es, toute variable ss fera partie du deuxième sous-ensemble de l'ensemble $\boldsymbol{V}$.

Dans le cas où l'on traduit un <scope> en **B**-CFG sans prendre en compte les éventuels <eventHandler> et *FCT-handlers* (i.e. <faultHandlers>, <compensationHandlers> et <terminationHandlers>) qu'il pourrait inclure, le patron de traduction à adopter est celui donné par la figure 4.24. Ce patron correspond donc à la traduction de base d'une activité <scope> P pour laquelle seule son activité principale A est considérée.

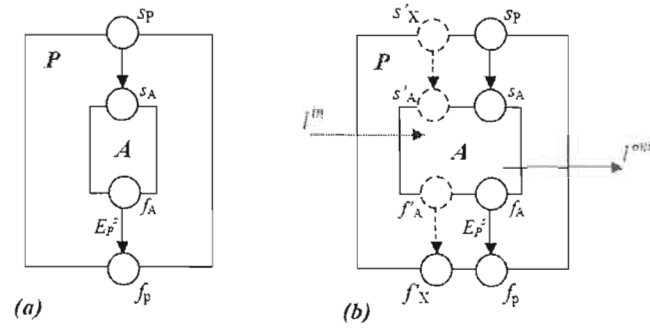


Figure 4.24 Traduction d'un <scope> avec son activité principale.

La figure 4.24(a) incarne le cas où un <scope> P n'est ni susceptible d'être sauté ni susceptible de voir une exception avoir lieu dans son enceinte. Dans pareil cas, une fois l'exécution de l'activité principale A est achevée, la variable ss associée à ce <scope> est mise à jour pour marquer une exécution couronnée de succès. La mise à jour de ladite variable se fait à travers l'arc E_P^s dont l'élément $\boldsymbol{ac} = (ss = true)$ et $\boldsymbol{pr} = Null$.

En revanche, la figure 4.24(b) incarne le cas où un <scope> P est susceptible d'être sauté sans que la production d'éventuelles exceptions ne soit considérée. Ce patron est à adopter pour un pareil <scope> dont l'activité principale A, voire (le cas échéant) les sous-activités de cette dernière, est (sont) source(s) ou cible(s) de liens de contrôle (e.g., l^{in} ou l^{out}). Dans le cas où le <scope> P est sauté, la variable ss lui étant associée garde sa valeur *unset*. Finalement, pour le cas où le <scope> P est lui-même cible ou source de liens de contrôle, les patrons de traduction qui seront considérés sont ceux décrits dans la section 4.6.3, et pour qui la sous-composante X *core* sera remplacée par l'un des deux patrons relatés par la figure 4.24.

4.6.6 Les event Handlers

Il est possible pour un <scope> donné d'avoir à traiter des événements normaux (i.e., événements de réception de messages et de déclenchement d'alarmes) qui auraient lieu au cours de son exécution. Ce traitement est rendu possible par le biais des *eventHandlers* que le <scope> peut définir. Deux types d'*eventHandlers* sont à distinguer : les <onEvent> qui sont déclenchés suite à la réception d'un message, et les <onAlarm> qui sont déclenchés suite à l'expiration d'un délais (i.e., à un *timeout*). L'activité à exécuter suite à l'avènement d'un événement, et qui est donc à associer à un eventHandler, ne peut correspondre qu'à une activité <scope>.

Si les <onEvent> sont en partie définis comme un <receive>, les <onAlarm> quant à eux sont définis à base de l'expression <for>, <until> ou <repeatEvery>. Dans un même <onAlarm>, « les deux expressions <for> et <until> sont mutuellement exclusives » (1), tandis que leurs cohabitations respectives avec une expression <repeatEvery> sont possibles. Contrairement aux <onAlarm> qui sont basés sur une expression <for> ou <until> et qui peuvent avoir lieu au plus une fois, les <onEvent> ainsi que les <onAlarm> qui sont basés sur un <repeatEvery> peuvent être déclenchés une multitude de fois tant que le <scope> auquel ils sont associés demeure actif. Par ailleurs, on trouve que la norme BPEL interdit à tout lien de contrôle de traverser les frontières d'un <eventHandler>. Cela nous ramène donc à affirmer que chaque *eventHandler* d'un <scope> P forme à lui seule une unité logique d'activités qui est indépendante et qui se trouve attachée à ce <scope> P.

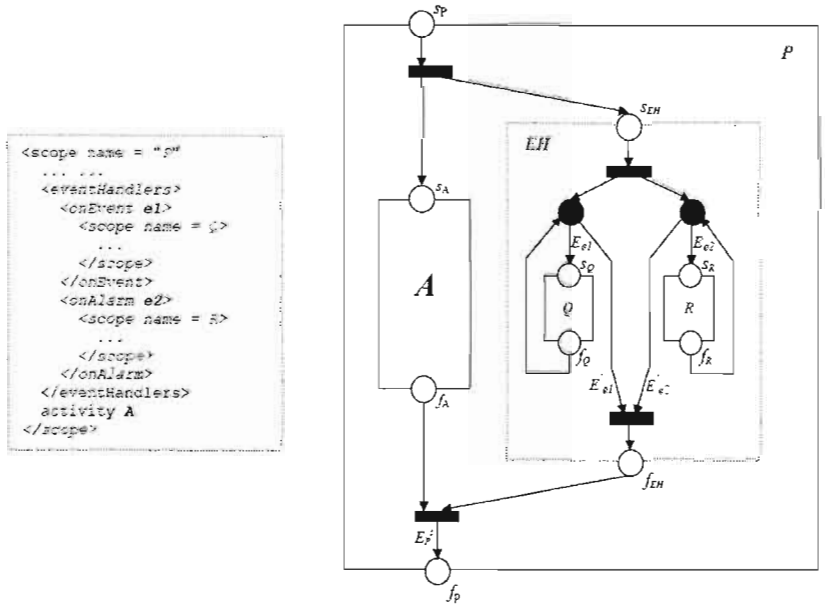


Figure 4.25 Traduction des eventHandlers de type <onEvent> et <onAlarm>.

La figure 4.25 traduit la présence de deux eventHandlers dans un <scope> P. Les deux eventHandlers sont représentés de façon à montrer que leur exécution se fait en parallèle avec l'activité principale A dudit <scope>.

Le premier eventHandler est un <onEvent> disposant d'un scope associé Q qui est exécuté suite au déclenchement d'un événement *e1*. Le déclenchement de cet événement est synonyme de l'emprunt de l'arc E_{e1}. Cet arc gardera la valeur *Null* pour son élément *pr*, mais verra son

action correspondre à un enregistrement de valeurs (i.e., $ac = Enreg$) — valeurs transmises par le message reçu pour être affectées à une/des variable(s) du scope Q. De son côté, le deuxième eventHandler est un <onAlarm> disposant d'un scope associé R qui est exécuté suite au déclenchement d'un événement *e2* et donc après emprunt de l'arc E_{e2}. Cet arc gardera la valeur par défaut *Null* pour ses deux éléments *ac* et *pr*. Les deux arcs E'_{e1} et E'_{e2} ont eux aussi des éléments *ac* et *pr* qui sont à *Null*, et le choix entre eux et leurs deux rivaux respectifs (i.e. E_{e1} et E_{e2}) est un choix non-déterministe.

La traduction de chacun des deux eventHandlers associés au scope P est enrichie par l'ajout d'un arc de retour. Un tel arc souligne le fait que le scope Q ou R peut être exécuté plusieurs fois, i.e. les eventHandlers, auxquels ces scopes sont associés, sont invoqués plusieurs fois pour résulter en des instances qui s'exécutent en série. À vrai dire, pour le cas d'un <onAlarm>, cela ne s'applique que si l'expression <repeatEvery> fait partie de sa définition. Dans le cas contraire, i.e. un <onAlarm> basé exclusivement sur l'expression <for> ou <until>, l'arc de retour sortant du noeud f_R devra joindre l'arc E'_{e2} à un noeud de fusion dont sortira un arc qui sera synchronisé avec l'arc E'_{e1}. Dans ce cas, l'eventHandler sera exécuté au plus une fois. Même pour le cas d'un <onEvent> ou d'un <onAlarm> basé sur une expression <repeatEvery>, nous pourrions, pour son test, n'en retenir qu'une seule ou aucune exécution. Cela reviendrait à adopter le critère de couverture *0-1 répétition* qui est aussi adopté pour le test des activités répétitives (voir sous-section 5.3.4).

Le patron de traduction donné par la figure 4.25 représente des eventHandlers dont chacun ne peut avoir qu'une seule instance active à la fois. Cependant, comme il l'est souligné par la norme BPEL, la création et l'exécution *simultanée* de plusieurs instances, d'un <onEvent> ou d'un <onAlarm> basé sur une expression <repeatEvery>, peuvent avoir lieu. Cela nous conduit donc à définir un autre patron de traduction, qui soit adapté à une telle situation, par duplication du sous-graphe représentant l'eventHandler concerné. Or, on ne peut pas connaître à l'avance le nombre d'instances qui seraient créées et donc le nombre de duplication à envisager, ce qui nous mènerait à considérer des graphes **B**-CFG illimités qui seraient difficiles à manipuler.

Pour contourner ce problème, il faut fixer une borne supérieure n pour le nombre d'instances allouées à s'exécuter parallèlement. Définir cette borne sera équivalent à définir le nouveau critère de couverture *0-n instances* que nous considérons pour le cas d'une exécution parallèle de plusieurs instances d'un eventHandler. Dans ce cas, le patron à considérer devrait correspondre à celui donné par la figure 4.26. En fait, comme c'était le cas pour le <forEach> parallèle,

c'est la variante *0-2 instances* du critère *0-n instances* qui sera adoptée et décrite dans ce qui suit (voir 5.3.4). Ce même problème de graphes illimités s'est posé à Ouyang (48) pour sa traduction du BPEL en réseau de Petri, ainsi qu'aux auteurs Taylor et *al.* (60) qui ont abordé le test du langage concurrent Ada et la création dynamique de plusieurs *tasks*.

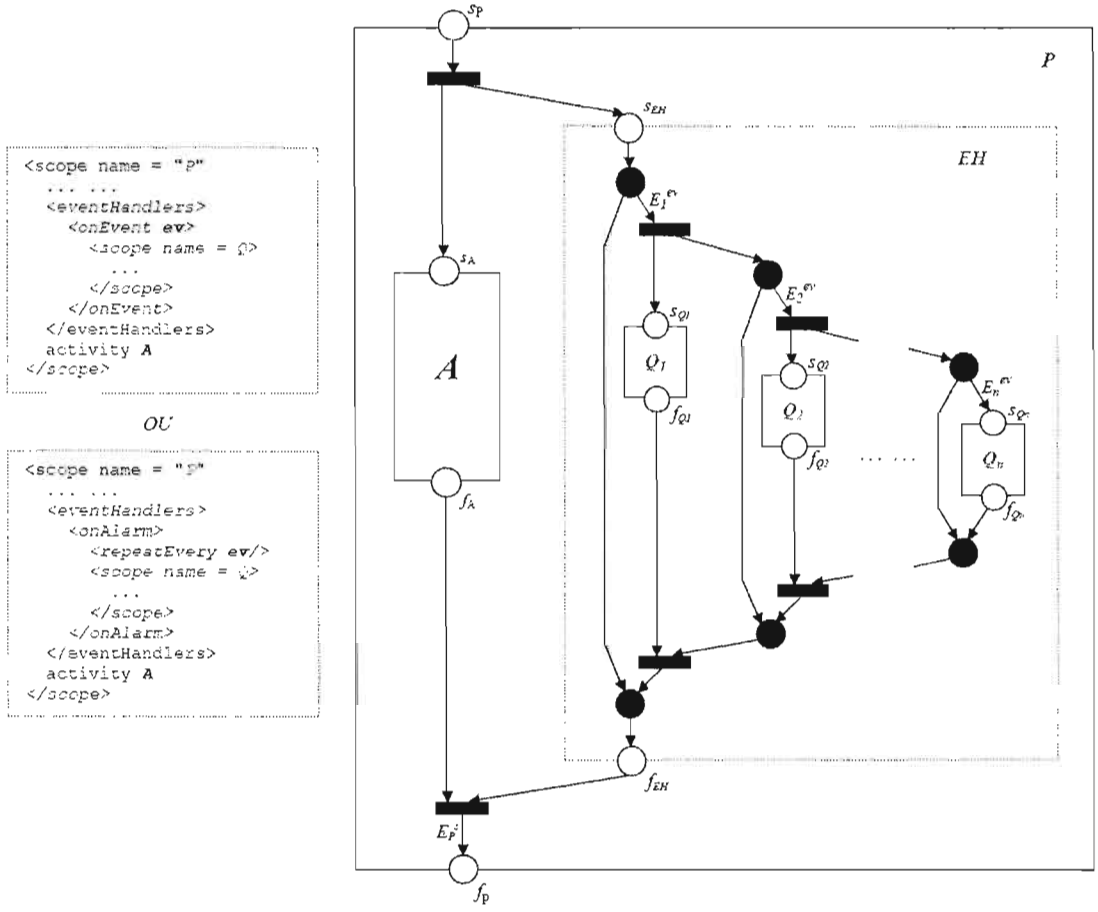


Figure 4.26 Traduction d'un `<eventHandler>` susceptible de résulter en n instances simultanées

4.6.7 Les fault Handlers

Pour le traitement d'erreurs, la norme BPEL offre aux scopes la possibilité d'inclure des composantes dédiées rien qu'à cette fin. Ces composantes sont les *faultHandlers*. Ces derniers sont généralement conçus dans l'optique d'annuler le travail qui n'a pas pu être achevé dans un scope où une erreur a eu lieu. Un faultHandler est défini d'une façon soit explicite soit implicite. Celui étant défini implicitement est désigné dans la norme BPEL par *default faultHandler*. Ce

dernier est attaché à tout scope dans lequel des faultHandlers explicites n'ont pas été définis. Cela sous-entend que chaque scope disposera au moins d'un (default) faultHandler. Quant aux faultHandlers explicites, ils donnent, via des constructeurs `<catch>` et `<catchAll>`, la possibilité de personnaliser le traitement à envisager dans le cas du déclenchement d'erreurs. En particulier, le constructeur `<catch>` est défini pour un type d'erreur bien spécifique. Le lien entre l'erreur déclenchée et un tel constructeur se fait en se basant sur le nom attribué à cette erreur ou le type du message qui serait généré pour la décrire.

Selon la norme BPEL, trois sortes d'erreurs sont à considérer : les erreurs déclenchées suite à une invocation erronée d'un service, qu'on peut qualifier de « *invocation faults* »; les erreurs définies explicitement par le programmeur via l'usage des activités `<throw>` et `<rethrow>`, qu'on peut qualifier de « *programmer faults* »; et finalement les erreurs standard que le BPEL définit (e.g., joinFailure), qu'on peut qualifier de « *system faults* ». Après la présentation du patron générique à adopter pour la traduction des faultHandlers en B-CFG (figure 4.27), deux autres patrons (figures 4.28 et 4.29), qui en sont une instanciation, seront donnés comme exemple au traitement d'erreurs de type « *invocation faults* » et « *system faults* ».

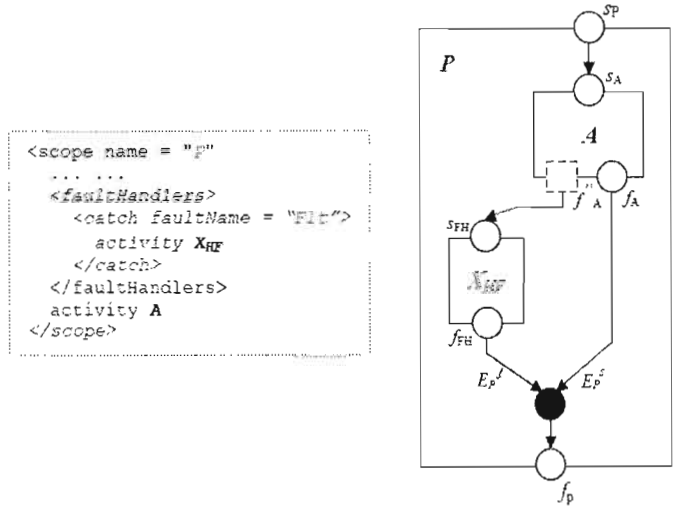


Figure 4.27 Patron générique de la traduction d'un `<faultHandler>` associé à un `<scope>`.

La figure 4.27 expose le patron générique représentant les grandes lignes de la traduction en B-CFG de tout faultHandler associé à un scope P. À vrai dire, c'est le contour intitulé X_{HF} qui représente le faultHandler traduit, et donc l'activité à exécuter si une erreur *Flt* a lieu lors de l'exécution normal du scope d'attache P. Si une telle erreur a lieu, le flot d'exécution normal

de l'activité principale A doit être interrompu pour céder la main à un flot qui se poursuivra jusqu'à arriver au noeud f''_A. Ce noeud ne marque pas la fin d'un chemin de saut, mais sert plutôt à annoncer la fin d'une terminaison forcée de l'activité principale A. Une fois que le noeud en question est atteint, l'exécution du faultHandler associé peut être entamée. Cela est en harmonie avec la condition dictée par la norme BPEL et exigeant à ce qu'un faultHandler ne soit exécuté qu'après la fin de la terminaison forcée de l'activité principale A.

Si l'arc E^s_P (s pour *successful*) sert à mettre à jour la variable ss associée au scope P pour en marquer une exécution couronnée de succès, l'arc E^f_P (f pour *faulty*) en fait l'opposé. Pour ce faire, ce dernier dispose d'un élément $ac = (ss = false)$ et d'un prédicat pr gardé à la valeur *Null*. Les deux arcs E^s_P et E^f_P rejoignent un même noeud de fusion et permettent ainsi, que l'exécution du scope soit réussie ou pas, de donner suite à un flot normal qui reprendra le flambeau. Au fait, si dans un scope P une erreur a lieu tout en étant traitée avec succès, les éventuels liens de contrôle dont ce scope serait source seront évalués comme si de rien n'était.

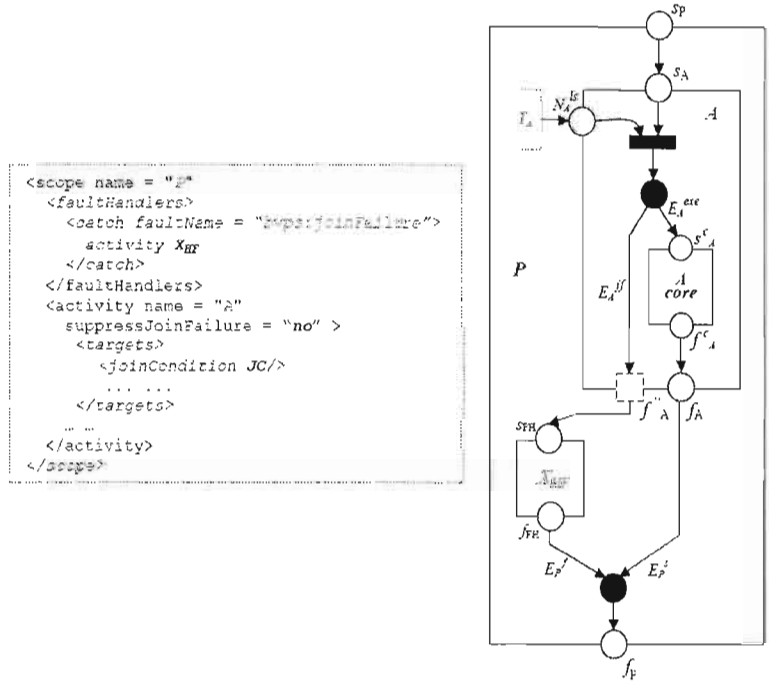


Figure 4.28 Patron du traitement d'une erreur *join failure*.

Le patron relaté par la figure 4.28 est le résultat d'une instanciation du patron générique donné ci-dessus. Ce patron instance est adopté pour le cas de traitement d'une erreur de type *system faults*, en l'occurrence l'erreur *joinFailure*. Cette erreur se déclenche dans le cas d'une

activité A qui est cible à des liens de contrôle, a un attribut $suppressJoinFailure$ = "no" et pour laquelle on évalue la <joinCondition> JC à faux. Dans pareille situation, c'est l'arc E_A^{jf}, dont le prédicat $\boldsymbol{pr} = !JC$, qui sera emprunté plutôt que l'arc E_A^{exe} dont le prédicat $\boldsymbol{pr} = JC$. L'emprunt de l'arc E_A^{jf} mène à l'exécution de l'activité X_{HF} définie par le faultHandler traduit.

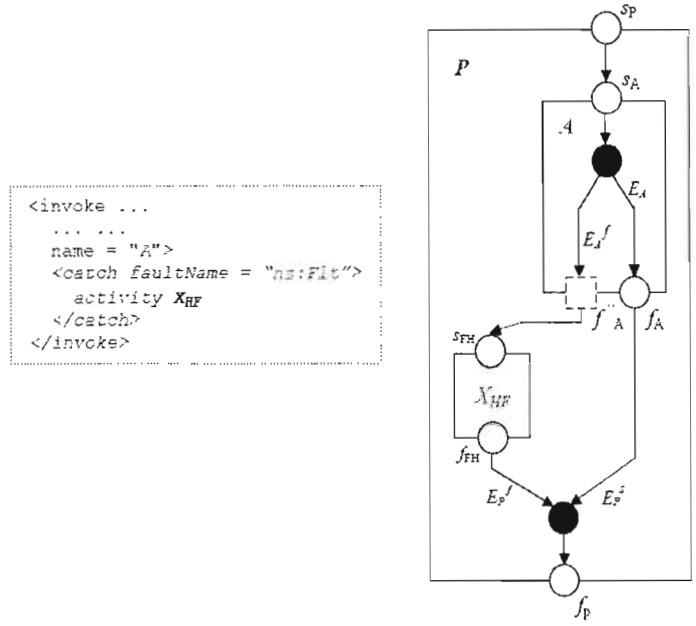


Figure 4.29 Patron du traitement d'une erreur de type *invocation faults*.

Une autre instanciation du patron générique donné ci-dessus est dépeinte par la figure 4.29. Cette deuxième instanciation présente la traduction du cas d'un faultHandler[23] qui serait appelé à s'exécuter suite à l'occurrence d'une erreur de type *invocation faults*. Une telle erreur aurait lieu si l'appel à un service, par une activité <invoke> A de type *requête-réponse*, échoue et aboutit à la réception d'un message d'erreur. Dans ce cas, l'arc E_A^f est emprunté. Cet arc permet de garder trace du message d'erreur reçu, en faisant correspondre son élément $\boldsymbol{ac}$ à l'action *Enreg*. Contrairement à cela, l'arc E_A est emprunté si l'opération d'invocation qui serait initiée par l'activité A ne résulte pas en la réception d'un message d'erreur. À l'image de l'élément $\boldsymbol{ac}$ de l'arc E_A^f, celui de l'arc E_A correspond lui aussi à l'action *Enrg*, sauf que pour chacun des deux arcs, la nature du message à sauvegarder diffère.

---

[23] Le faultHandler en question est local à un <invoke>. « Cela est équivalent à la présence d'une activité <scope> implicite qui enveloppe cet <invoke> incluant dans sa déclaration ledit faultHandler » (1). Dans le cas d'une telle déclaration, le <scope> implicite devrait, certes, prendre le même nom que celui de l'<invoke> qu'il enveloppe, mais pour des soucis de clarté, nous leur avons donnés des noms différents, en l'occurrence : A pour l'activité <invoke> et P pour le <scope> implicite.

4.7 Conclusion

Au terme de ce quatrième chapitre, nous sommes arrivés à proposer une version étendue du graphe CFG que nous avons baptisée **B**-CFG, et cela afin de pouvoir reproduire fidèlement les particularités comportementales de tout processus BPEL à tester. Cette version étendue a été *formellement* définie comme elle a été présentée comme un ensemble de patrons et de règles qui régissent l'étape de traduction. Dans le chapitre suivant, nous verrons comment nous tirons profit du graphe **B**-CFG résultant de cette première étape de traduction pour la génération des chemins et données de test, et cela en respect d'un certain nombre de critères de couverture.

CHAPITRE V

TEST DES PROCESSUS BPEL : LA GÉNÉRATION DES CAS DE TEST

5.1 Introduction

Après la traduction du processus à tester en un graphe B-CFG, les prochaines étapes à entreprendre, selon notre variante de « *l'approche orientée chemins* », correspondent aux deux étapes de génération des chemins et données de test, i.e., génération des cas de test.

Dans le présent chapitre, nous commençons par proposer une représentation formelle des chemins de test qui rompt avec celle décrite par la définition 4 de la section 2.3, et cela afin de pouvoir représenter convenablement des chemins décrivant des comportements de toute nature (séquentielle ou concurrente soit-elle). Par la suite, les algorithmes que nous avons élaborés pour la génération et l'épuration des chemins de test, en adéquation à certains types de couverture, sont présentés. Nous présentons aussi une solution pour la génération des données de test pour habiller les chemins résultants — solution qui est à mettre en oeuvre dans de futurs travaux. Finalement, des pistes pour la spécification et l'exécution des cas de test sont fournies.

5.2 Génération des chemins de test

En ce qui a trait à la génération des chemins de test pour les programmes concurrents, plusieurs approches ont été proposées. Bon nombre de ces approches se base sur le préalable de construction d'un graphe d'accessibilité (*GA*), afin d'aboutir par suite à la génération de chemins de test *sérialisés*. Certes, lesdites approches présentent certains avantages (e.g., tous les chemins générés sont faisables (73)), mais en pratique, elles se retrouvent malheureusement limitées par le problème d'explosion de l'espace d'états (23; 76; 81).

Dans la littérature, on trouve aussi d'autres approches qui se distinguent par leur adoption de méthodes qui sont basées sur l'*analyse des chemins* (20; 78). En premier, ces approches génèrent les chemins locaux propres à chaque composante (e.g., à chaque *task* d'un programme Ada) pour ensuite combiner ces derniers de façon à obtenir des chemins de test globaux. De telles approches « s'appliquent à des programmes consistant en des processus ou tâches de communication, comme ceux écrits en Ada ou CSP, mais sont inappropriées pour BPEL qui n'as ni une séparation explicite des processus (*tasks*) individuels ni de synchronisation via rendez-vous » (76; 81).

Dans notre cas, nous avons opté donc pour une approche qui permet de respecter les spécificités du BPEL et de nous éviter la construction d'un graphe d'accessibilité qui induirait un problème d'explosion de l'espace d'états. Cette approche ne se basera que sur le graphe **B**-CFG et permettra par la suite d'aboutir, non pas à des chemins sérialisés, mais, directement, à des chemins concurrents *partiellement* ordonnés ou à des chemins séquentiels *totalement* ordonnés. Les chemins concurrents sont constitués de noeuds qui seront représentés dans un ordre partiel, alors que les chemins séquentiels sont constitués de noeuds qui seront représentés dans un ordre total. Dans ce qui suit, nous définissons chaque chemin de test généré comme un ensemble de noeuds qu'on a muni d'une *relation d'ordre* partielle ou totale, dépendamment de ce que le chemin représenté est concurrent ou séquentiel.

5.2.1 Les relations d'ordre et les chemins de test

Dans un graphe CFG, un chemin de test commence toujours par le noeud d'entrée de ce graphe pour prendre fin à son noeud de sortie. Un tel chemin est généralement représenté par un sous-ensemble de l'ensemble des noeuds du graphe considéré. Les noeuds constituant ce sous-ensemble sont présentés dans l'ordre selon lequel ils ont été visités lors du parcours du graphe pour la génération du chemin. En effet, tout chemin de test étant extrait d'un graphe CFG ordinaire pourrait être représenté par un ensemble de noeuds qui sont totalement ordonnés.

En revanche, pour un programme concurrent (un processus BPEL), les chemins de test qui seraient générés peuvent être de nature séquentielle ou concurrente, dépendamment de ce que ces derniers décrivent un comportement concurrentiel ou pas. Pour une représentation qui siéra aux chemins des deux natures (concurrente et séquentielle) nous nous sommes, comme nous l'avons signalé plus haut, ressourcés de la notion de *relations d'ordre*. Un exposé concis de cette notion est fourni dans la section suivante.

5.2.1.1 Les relations d'ordre

En mathématique — comme dans la vraie vie — il y a des ensembles dont certains éléments se trouvent reliés les uns aux autres par des relations qui ne tiennent pas forcément pour le reste des éléments (58). L'une de ces relations les plus usuelles est celle qu'on appelle communément *relation d'ordre*. Une telle relation, lorsqu'elle est associée à un ensemble E, définit une relation *binaire*[1] sur cet ensemble et permet ainsi de comparer ses éléments entre eux d'une façon *cohérente*. Formellement, la définition de toute relation d'ordre est basée sur les trois propriétés données ci-dessous (21) :

- ***Réflexivité*** : une relation $\mathcal{R}$ définie sur l'ensemble E est dite *réflexive* si elle met en relation tout élément avec lui-même, autrement dit : $\forall\, x \in E,\ x\mathcal{R}x$.
- ***Antisymétrie*** : une relation $\mathcal{R}$ définie sur l'ensemble E est dite *antisymétrique* si elle ne met en aucun cas les éléments distincts de cet ensemble en relation mutuelle, autrement dit : $\forall (x, y) \in E^2$, $x\mathcal{R}y$ et $y\mathcal{R}x \Longrightarrow x = y$.
- ***Transitivité*** : une relation $\mathcal{R}$ définie sur l'ensemble E est dite *transitive* si elle met en relation deux éléments en transitant par un troisième, autrement dit : $\forall (x, y, z) \in E^3$, $x\mathcal{R}y$ et $y\mathcal{R}z \Longrightarrow x\mathcal{R}z$.

Définition 1 « Une relation $\mathcal{R}$ définie sur l'ensemble E est appelée une *relation d'ordre*, ou plus simplement un *ordre*, si elle est réflexive, antisymétrique et transitive » (58).

Dans notre cas, nous aurons particulièrement besoin de deux variantes qui découlent de cette notion de base de relation d'ordre, à savoir les *relations d'ordre strict* et les *préordres*. La première variante permet de restreindre une relation d'ordre aux couples d'éléments *distincts*, alors que la deuxième permet d'étendre, et donc de muter, une relation d'ordre en une où l'on autoriserait les cycles de plus d'un élément (i.e. les cycles non triviaux). La définition de chacune des deux variantes est donnée comme suit :

Définition 2 Un *préordre* défini sur un ensemble E est une relation binaire *réflexive* et *transitive*.

---

[1] Une relation binaire $\mathcal{R}$ d'un ensemble E vers un ensemble F est définie par une partie $\mathcal{G}$ de E×F. Si (x,y) $\in \mathcal{G}$ on dit que x est en relation avec y et on le note « x$\mathcal{R}$y ».

Définition 3 Une *relation d'ordre strict* $\mathcal{R}$ définie sur un ensemble E est une relation binaire *irréflexive* (i.e. $\forall x \in E, x\mathcal{R}x$) et *transitive*[2].

Outre les variantes définies ci-dessus, nous tenons aussi à souligner que les relations d'ordre sont scindées en deux catégories. Une première catégorie désigne les relations d'ordre dites *totales*, et une deuxième à qui sont attachées les relations d'ordre dites *partielles*. La définition de chacune des deux catégories est donnée comme suit :

Définition 4 Une relation d'ordre $\mathcal{R}$ définie sur un ensemble E est dite *totale* si tous les éléments de E sont *comparables*, autrement dit : $\forall (x, y) \in E^2$, $x\mathcal{R}y$ ou $y\mathcal{R}x$.

Définition 5 Une relation d'ordre $\mathcal{R}$ définie sur un ensemble E est dite *partielle* s'il existe deux éléments de E qui ne peuvent être mis en relation, ni dans un sens ni dans l'autre, autrement dit : $\exists (x, y) \in E^2$, $x\mathcal{R}y$ et $y\mathcal{R}x$.

Comme exemple, la relation « *est supérieur ou égal à* » est une relation d'ordre car elle est réflexive, antisymétrique et transitive. Par contre, la relation « *est strictement supérieur à* » est une relation d'ordre strict car elle est, certes, antisymétrique et transitive, mais irréflexive. Quant à la relation « *divise* », elle définit une relation d'ordre sur les entiers naturels qui ne peut être totale, mais partielle.

5.2.1.2 Représentation des chemins de test

Comme nous l'avons avancé dans ce qui précède, un chemin de test extrait d'un graphe CFG est généralement défini par un ensemble de noeuds qui sont présentés dans un ordre séquentiel total, i.e., ordre selon lequel ils ont été visités lors du parcours du graphe. En revanche, pour un graphe B-CFG traduisant un programme BPEL, chaque chemin de test qui en serait extrait correspondra à un ensemble de noeuds qui seraient totalement ou partiellement ordonnés, dépendamment de ce que le chemin généré serait séquentiel ou concurrent.

Pour une représentation convenable pour les chemins séquentiels et pour les chemins concurrents qui seraient extraits d'un graphe B-CFG, nous avons donc pensé à présenter chacun

[2]Les propriétés d'irréflexivité et de transitivité permettent de déduire qu'une relation d'ordre strict est aussi antisymétrique, voire antisymétrique en un sens plus fort qu'une relation d'ordre de base. Autrement dit, si un élément x est mis en relation avec un élément y par une relation d'ordre strict, cela implique que y ne peut l'être avec x par cette même relation : $\forall (x, y) \in E^2$, $x\mathcal{R}y \Longrightarrow y\mathcal{R}x$.

de ces chemins comme un ensemble de noeuds P auquel nous associons la relation « *est visité avant ou égal à* » que nous symboliserons par "$\leq$". Cette relation est en fait un préordre que nous définissons sur l'ensemble des noeuds P, en respect des propriétés de réflexivité et de transitivité qui sont requises pour la définition de tout préordre, en l'occurrence :

- $\forall\ n \in P,\ n \leq n$ (réflexivité);
- $\forall (n_1, n_2, n_3) \in\ P^3,\ n_1 \leq n_2$ et $n_2 \leq n_3 \Longrightarrow n_1 \leq n_3$ (transitivité).

La considération d'un préordre, au lieu d'une relation d'ordre de base, s'est vue imposée car, par définition, un préordre autoriserait la présence des cycles non triviaux (i.e. de plus d'un élément) dans un chemin — cycles qui seraient induits dans un chemin suite au parcours d'un arc de retour (e.g., dû à la traduction d'un `<while>`) qui serait présent dans le graphe **B**-CFG considéré. Au fait, c'est la propriété d'antisymétrie propre à toute relation d'ordre de base qui, n'étant requise pour un préordre, rend impossible la présence de ces cycles non triviaux.

Cela dit, que ce soit pour le cas d'un graphe **B**-CFG n'incluant aucun arc de retour ou pour celui où le critère *0-1 répétition*(voir 5.3.4) serait considéré, la question de présence de cycles ne sera plus à l'ordre du jour. En effet, dans ces deux cas, nous considérerons la relation d'ordre strict « *est visité avant* » que nous symboliseront par "$<$". Cette dernière permet de restreindre la relation d'ordre aux couples d'éléments (i.e. de noeuds) *distincts* en se conformant aux deux propriétés d'irréflexivité et de transitivité qu'elle définit ainsi :

- $\forall\ n \in P,\ n \not< n$ (irréflexivité);
- $\forall (n_1, n_2, n_3) \in\ P^3,\ n_1 < n_2$ et $n_2 < n_3 \Longrightarrow n_1 < n_3$ (transitivité).

Qu'il s'agisse d'un préordre ou d'une relation d'ordre strict, nous constatons que les deux partagent la propriété de *transitivité*. Pour notre travail, nous nous sommes intéressés à cette propriété plus qu'aux autres car, au fond, c'est cette dernière qui nous permettra de représenter les chemins de test en comparant et, donc, en ordonnant les éléments (i.e., les noeuds) les constituant, entre eux, d'une façon *cohérente*. Par défaut, c'est la relation d'ordre strict qui sera considérée, tandis que le préordre ne sera évoqué que dans le cas où le graphe **B**-CFG considéré inclut un arc de retour.

Finalement, nous voulons mentionner que pour la représentation d'un chemin de test séquentiel, le préordre ou la relation d'ordre strict, dont sera muni son ensemble de noeuds P, sera *total(e)*. En revanche, pour la représentation d'un chemin de test concurrent, le préordre

ou la relation d'ordre strict, dont sera muni son ensemble de noeud P, sera *partiel(le)*. Tout au long du reste du mémoire, et sauf indication contraire, nous n'emploierons que l'appellation « relation d'ordre strict (ou préordre) partielle »[3].

Pour des fins d'illustration, ci-dessous sont donnés des exemples de représentation de chemins de test. Ces chemins sont extraits du graphe B-CFG de la figure 5.1. Vu que ce graphe n'inclut aucun arc de retour, seule la relation d'ordre strict « *est visité avant* » sera considérée.

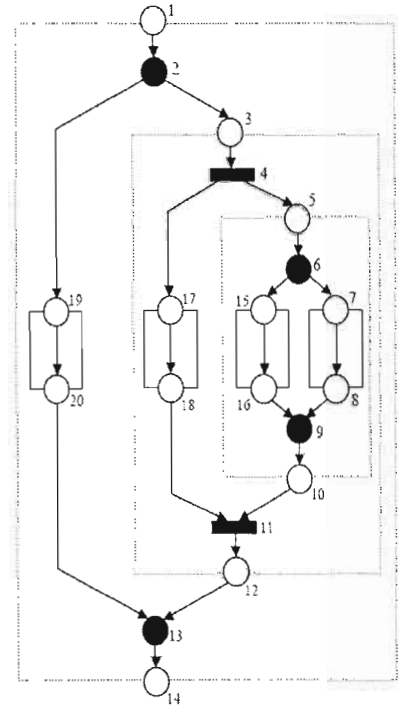


Figure 5.1 Exemple d'un graphe B-CFG dont on a numéroté les noeuds.

Un premier chemin C_1, constitué des noeuds regroupés par l'ensemble $P_1 = \{1, 2, 19, 20, 13, 14\}$, est un exemple de chemin de test séquentiel. Le chemin étant séquentiel, son ensemble P_1 est donc censé être totalement ordonné. Ainsi, pour sa représentation, ledit chemin verra son ensemble P_1 muni de la relation d'ordre strict « *est visité avant* », symbolisée par "$<$" et qui dans ce cas sera totale[4]. Cela donne lieu à la représentation $C_1 = [(1 < 2), (2 < 19), (19 < 20), (20 < 13), (13 < 14)]$.

---

[3] Certains mathématiciens définissent une relation d'ordre totale comme un cas particulier d'une relation d'ordre partielle où le nombre d'éléments non comparables est nul (21).

[4] La *totalité* pour une relation d'ordre strict définie sur un ensemble E s'exprime comme suit : $\forall (x, y) \in E^2$, $x < y$ ou $x = y$ ou $y < x$.

Un deuxième chemin C_2, constitué des noeuds regroupés par l'ensemble P_2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18}, est un exemple de chemin de test concurrent. Le chemin étant concurrent, son ensemble P_2 est donc censé être partiellement ordonné. Ainsi, pour sa représentation, ledit chemin verra son ensemble P_2 muni de la relation d'ordre strict « *est visité avant* », symbolisée par "$<$" et qui dans ce cas sera partielle. Cela donne lieu à la représentation : C_2 = [(1 < 2), (2 < 3), (3 < 4), (4 < 5), (5 < 6), (6 < 7), (7 < 8), (8 < 9), (9 < 10), (10 < 11), (11 < 12), (12 < 13), (13 < 14), (4 < 17), (17 < 18), (18 < 11)].

Dans ce qui suit, nous simplifions la représentation des chemins de test à générer en remplaçant chaque couple $(x < y)$ présent dans un chemin C_i par (x, y). Ainsi la représentation des chemins C_1 et C_2 changera d'allure pour :

- C_1 = [(1,2), (2,19), (19,20), (20,13), (13,14)];
- C_2 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11), (11,12), (12,13), (13,14), (4,17), (17,18), (18,11)].

De plus, pour le reste du mémoire, chaque couple (x, y) représentera, certes, la relation d'ordre entre les noeuds x et y présents dans un chemin, mais sera aussi considéré comme une incarnation de l'arc reliant ces deux noeuds.

5.2.2 L'algorithme de génération des chemins de test

5.2.2.1 Entrée/sortie de l'algorithme

L'*entrée* principale de notre algorithme de génération des chemins de test est un graphe **B**-CFG. Ce graphe doit être représenté d'une façon ou d'une autre de sorte à ce qu'il soit traitable par ledit algorithme. D'après Cormen (8), « il existe deux façons standard pour représenter un graphe : comme une collection de *listes d'adjacence* ou comme une *matrice d'adjacence* ». Dans notre cas, nous avons préféré une représentation en listes d'adjacence à une qui aurait été basée sur une *matrice d'adjacence* en raison de la nature *peu dense* du graphe **B**-CFG. La faible densité de ce dernier est due au fait qu'il s'agit d'un graphe *orienté* et *(quasi-) acyclique*; les cycles qui seraient présents dans un tel graphe sont généralement peu nombreux vu qu'ils ne devraient émanés que de la traduction des constructeurs itératifs comme le `<while>`.

Notre représentation d'un graphe **B**-CFG $< s, f, N, E, V >$ en listes d'adjacence repose sur un tableau Adj de $|N|$ éléments, un élément par noeud. Au fait, pour chaque nœud $u \in N$,

l'élément du tableau lui étant associé pointe vers une liste de noeuds. Cette liste est constituée par l'ensemble des nœuds adjacents au nœud u considéré. Autrement dit : $\forall\ u \in N, Adj[u] = \{v\ /\ v \in N \text{ et } (u,v) \in E\}$. Un exemple d'une telle représentation est donné par la figure 5.2.

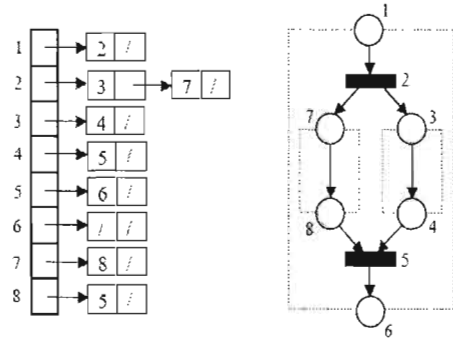


Figure 5.2 Exemple de représentation d'un graphe B-CFG en listes d'adjacence.

Quant à la *sortie* de l'algorithme, elle correspondra à une liste dont les éléments sont les chemins de test à générer lors du parcours du graphe. Conformément à ce qui a été souligné plus haut, chaque chemin de test C, qui ferait partie de cette liste, sera représenté par $C = [(n_1, n_2), (n_2, n_3), ..., (n_{k-1}, n_k)]$, où chaque couple (n_{i-1}, n_i) représente la relation d'ordre définie entre les noeuds n_{i-1} et n_i, en plus d'être l'incarnation de l'arc reliant ces deux noeuds. Représentés de la sorte, les chemins de test générés, en particulier ceux de nature parallèle, peuvent être facilement repérés et par suite marqués dans un graphe B-CFG. Au fait, on démontre que « tout ensemble ordonné fini (non vide) peut être représenté par un graphe » (58), et pour le cas d'un chemin de test, qui lui aussi est défini comme un ensemble ordonné fini (de noeuds), ce graphe correspondra à un sous-graphe du graphe B-CFG dont ce chemin a été dérivé.

5.2.2.2 Description de l'algorithme

Nous rappelons que le graphe B-CFG a été conçu de façon à réduire au minimum la quantité de données qui y serait présente, et cela en faisant croître au maximum la richesse de l'information *graphique* représentant le code. Ce choix a été entrepris pour éviter au maximum d'avoir à analyser les données présentes dans le graphe lors de son parcours pour la génération des chemins de test. Partant de ce fait, nous avons pu définir notre algorithme de génération des chemins de test de sorte à ce qu'il soit exclusivement basé sur la richesse de l'information graphique représentant le code — une richesse garantie par la syntaxe B-CFG.

L'algorithme que nous proposons adopte un parcours en *profondeur d'abord* (i.e., *DFS* pour *Depth-First Search*). Autrement dit, en commençant par le noeud d'entrée s, l'algorithme progresse en profondeur dans le graphe **B**-CFG considéré tant que cela est possible. Cette progression se fait en empruntant les arcs de ce graphe pour en explorer les noeuds à partir du noeud le plus récemment découvert et ayant des noeuds adjacents non encore visités. En outre, lors de son exécution, l'algorithme considère les types de noeuds définis par la syntaxe **B**-CFG en distinguant les noeuds de bifurcation des noeuds de décision ainsi que du reste (i.e., des noeuds de fusion, de jointure et normaux). Cette distinction donne lieu à trois types de traitement que nous associons aux différents types de noeuds comme suit :

1. Si le noeud u visité est un noeud de fusion, un noeud de jointure ou un noeud normal : on ajoute le couple (u, v), constitué du noeud u et de son noeud adjacent v, au chemin courant C — chemin en cours de génération. Cet ajout n'a lieu que si le couple (u, v) ne fait pas déjà partie du chemin C.
2. Si le noeud u visité est un noeud de bifurcation : on ajoute au chemin courant C les n couples (u, v) qui sont respectivement constitués du noeud u et de chacun de ses n noeuds v adjacents ($n = |Adj[u]|$). Outre cet ajout, l'algorithme cherche les n sous-chemins à ajouter au chemin courant C. Pour ce faire, chacun des n noeuds $v \in Adj[u]$ est gardé comme noeud à visiter (par la suite) pour la génération du sous-chemin dont il marque le début. Au fait, au chemin courant, nous associons une pile PB (Pile des Bifurcations) qui contiendra les noeuds adjacents aux noeuds de bifurcation faisant partie de ce chemin (e.g. chaque noeud $v \in Adj[u]$ est gardé dans la pile PB associée au chemin courant C).
3. Si le noeud u visité est un noeud de décision : on duplique le chemin courant C en n chemins, où n est le nombre des noeuds v adjacents au noeud u. À chacun de ces chemins on associe une copie de la pile PB étant associée au chemin courant C. De plus, les n couples (u, v), constitués du noeud u et de chacun de ses n noeuds v adjacents, sont respectivement ajoutés aux n chemins ainsi créés. Par la suite, ces n chemins, accompagnés de leurs piles PB respectives, seront contenus dans une pile PD (Pile des Décisions). Un des chemins de cette pile sera pris comme le chemin courant à traiter, alors que le reste sera laissé de côté pour être traité ultérieurement.

L'avancement en profondeur, dans un graphe **B**-CFG $< s, f, N, E, V >$, pourrait être suspendu suite à l'atteinte du noeud de sortie f ou d'un noeud (de jointure) dont le noeud adjacent a été déjà ajouté au chemin courant C. Ainsi, pour pouvoir relancer l'exploration du reste du

graphe, un retour aux noeuds adjacents, dont la suite n'a pas encore été explorée, est entrepris. Pour notre algorithme, ce retour se traduit, dans le cas où le chemin courant C reste incomplet, par la considération d'un noeud de la pile PB qui est associée à ce chemin. En revanche, si, arrivé à ce stade, le chemin courant C est complété (i.e., sa pile PB est vide), le retour se traduit par la considération d'un chemin de la pile PD comme le nouveau chemin courant à générer, i.e., considération du noeud le plus récemment découvert de ce dernier. Les chemins complétés sont joints à la liste LCT (Liste des Chemins de Test).

Notre algorithme ne commence la génération d'un nouveau chemin qu'une fois le traitement du chemin courant C est achevé. Autrement dit, l'algorithme donne la priorité au traitement des noeuds adjacents aux noeuds de bifurcation visités, et cela au détriment de ceux adjacents aux noeuds de décision découverts. Lorsque l'exploration de la suite de tous les noeuds adjacents aux noeuds de décision et aux noeuds de jointure découverts est accomplie, l'algorithme s'arrête — i.e. lorsque les piles PB et PD sont vides.

5.2.2.3 Pseudo-code de l'algorithme

Le pseudo-code fourni ci-dessous implémente l'algorithme de génération des chemins de test sous une forme itérative. Ce même algorithme aurait aussi pu être implémenté selon une forme récursive. Par ailleurs, ledit pseudo-code suppose que le graphe B-CFG à parcourir est acyclique, et si ce dernier ne l'est pas, nous verrons à la section 5.3.4 comment procéder pour le rendre comme tel. Ce pseudo-code fait appel aux méthodes décrites dans le tableau 5.1.

Méthode	Description
createPathCopy()	Crée une copie du chemin donné en paramètre et la retourne.
createPBCopy()	Crée une copie de la pile PB donnée en paramètre et la retourne.
push()	Empile un élément dans la pile PD ou PB.
pop()	Dépile un élément de la pile PD ou PB.
add()	Si le paramètre fourni est un couple (u, v), la méthode l'ajoute au chemin concerné. Cependant, si le paramètre correspond à un chemin, la méthode l'ajoute à la liste LCT.

Tableau 5.1 Les méthodes utilisées par l'algorithme de génération des chemins.

```
u <- 's'   // 's' est le nœud d'entrée du graphe
PB <- Null
PD <- Null
currentPath <- Null
WHILE (true) DO
  SWITCH(u)
      CASE a decison node :
           FOREVERY v IN Adj[u] DO
              tempCurrentPath <- createPathCopy(currentPath)
              tempPB <- createPBCopy(PB)
              tempCurrentPath.add((u,v))
              PD.push(v,tempCurrentPath,tempPB)
           ENDFOREVERY
           u,currentPath,PB <- PD.pop()
           CONTINUE   // Passer à la prochaine itération du 'while'
      CASE a bifurcation node :
           FOREVERY v IN Adj[u] DO
              PB.push(v)
              currentPath.add((u,v))
           ENDFOREVERY
           u <- PB.pop()
           CONTINUE
      DEFAULT :   // cas d'un nœud normal, de jointure ou de fusion
           IF Adj[u] not empty THEN
              Let v be the (unique) adjacent node of u
              IF (u,v) NOT IN currentPath THEN
                 currentPath.add((u,v))
                 u <- v
                 CONTINUE
              ENDIF
           ENDIF
  ENDSWITCH
  // Traitement à faire si l'on ne peut plus avancer en profondeur
  IF PB not empty THEN
      u <- PB.pop()
  ELSE
      LCT.add(currentPath)
      IF PD not empty THEN
           u,currentPath,PB <- PD.pop()
      ELSE
           BREAK     // Arrêt de l'algorithme
      ENDIF
  ENDIF
ENDWHILE
```

Pseudo code de l'algorithme de génération des chemins concurrents de test

Comme exemple d'une génération automatique des chemins de test, l'application de notre algorithme sur le graphe **B**-CFG décrit par la figure 5.1 donne lieu aux trois chemins de test :

- C_1 = [(1,2), (2,19), (19,20), (20,13), (13,14)];
- C_2 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11), (11,12), (12,13), (13,14), (4,17), (17,18), (18,11)];
- C_3 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,15), (15,16), (16,9), (9,10), (10,11), (11,12), (12,13), (13,14), (4,17), (17,18), (18,11)].

L'algorithme de génération des chemins de test proposé plus haut pourrait visiter plusieurs fois un même arc, voire toute une suite d'arcs, du graphe **B**-CFG parcouru. À titre d'exemple, pour le graphe **B**-CFG de la figure 5.1, l'arc incarné par le couple $(13, 14)$ sera visité à trois reprises, et la suite d'arcs $[(9, 10), (10, 11)]$ à deux reprises. À l'encontre de cela, nous proposons un deuxième algorithme qui ne visite qu'une seule fois les arcs d'un graphe **B**-CFG donné. La description et le pseudo-code de ce deuxième algorithme sont donnés à l'annexe C. Il est à signaler que l'*entrée* et la *sortie* de ce deuxième algorithme sont les mêmes que celles du premier, à savoir une liste d'adjacence comme entrée et une liste des chemins de test à générer comme sortie. Cela dit, nous tenons à préciser que si le premier algorithme ne commence la génération d'un nouveau chemin qu'une fois celui étant en cours de traitement est généré, le deuxième algorithme pourrait quant à lui générer plusieurs chemins en parallèle, en particulier si le graphe parcouru inclut des noeuds de bifurcation.

Au fait, les graphes incluant des noeuds de bifurcation (i.e., traduisant des comportements concurrentiels) pourraient éventuellement être constitués de structures *non-parenthésées*. En d'autres mots, une structure non-parenthésée qu'on pourrait trouver dans de tels graphes est une structure étant initiée par un noeud de bifurcation (ou un noeud de décision) qui n'a pas un noeud de jointure (*resp.* un noeud de fusion) associé — e.g., le graphe de la figure 5.3 inclut *quatre* noeuds de bifurcation et *cinq* noeuds de jointure, ce qui implique qu'*au moins* l'un des *cinq* noeuds de jointure ne sera associé à aucun noeud de bifurcation, d'où la présence d'au moins une structure non-parenthésée dans ce graphe. L'éventuelle présence de ces structures non-parenthésées dans un graphe rend difficile la génération des chemins de test car elle empêche à ce que le parcours des graphes **B**-CFG soit fait d'une façon modulaire qui serait plus simple à mettre en oeuvre. De même, cette éventuelle présence des structures non-parenthésée rend aussi impossible la représentation de nos chemins de test d'une façon qui serait similaire à la représentation informelle proposée par Yuan (donnée à l'annexe de (81)).

5.2.3 Élimination des chemins infaisables

La phase de génération des chemins de test vise à parcourir le graphe **B**-CFG considéré pour générer des chemins de contrôle (partant du noeud d'entrée du graphe à son noeud de sortie) dont chacun couvre une partie de ce graphe. Chacun de ces chemins est censé symboliser un comportement effectif du code BPEL traduit, i.e., symboliser l'un des scénarios effectifs ayant lieu lors de l'exécution de ce code. Or, une partie de ces chemins pourrait ne correspondre à aucun comportement effectif. C'est pour cela que lors de la génération des chemins de test, il faut faire la différence entre deux grandes catégories de chemins : les chemins dits « *exécutables* » ou « *faisables* » — chemins correspondant à des comportements effectifs du code testé —, et les chemins dits « *non exécutables* » ou « *infaisables* » — chemins qui ne seront jamais suivis lors de l'exécution du code testé.

Pour le cas de notre graphe **B**-CFG, nous avons recensé l'éventuelle présence de trois types de chemins infaisables. Ces trois types de chemins sont énumérés comme suit :

1. Un chemin C incluant un noeud de jointure v est censé inclure tous les noeuds u pour qui v est un noeud adjacent, et donc englober tous les couple (u, v) qui sont respectivement constitués de ces noeuds u et de leur noeud adjacent v. Autrement dit, si le nombre de ces noeuds u est k, le chemin C doit englober k sous-chemins parallèles qui tous rejoignent le noeud v et incluent les couples (u, v) respectivement. Si le chemin C n'inclut pas un de ces sous-chemins et donc un de ces couples (u, v), il est infaisable.

   Comme exemple, nous évoquons le chemin infaisable [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (4,8), (8,9), (9,16), (16,17), (17,18), (18,22), (22,23), (23,24), (24,25), (25,26), (26,27), (27,28), (28,29), (29,6), (2,11), (11,12), (12,13), (13,6), (12,14), (14,15), (15,17), (2,19), (19,20), (20,30), (30,35), (35,36), (36,34), (34,28), (30,31), (31,32), (32,33), (33,34)] extrait du graphe **B**-CFG de la figure 5.3. Ce chemin qui, incluant le noeud de jointure "22", n'inclut pas pour autant tous les noeuds pour qui "22" est un noeud adjacent, en particulier le noeud "21". Au fait, ledit chemin englobe non pas le sous-chemin [(2,19), (19,20), (20,21), (21,22)] incluant le noeud "21", mais l'autre sous-chemin qui diverge de ce dernier au noeud de décision "20" et qui exclut de sa composition ce noeud "21".

2. À la section 4.6.3, nous avons vu que chaque lien de contrôle l est traduit par deux arcs : un premier arc E_l représentant son état *exécuté* (true) et un deuxième E'_l représentant son état *dead path* (false). Nous avons aussi vu que chaque `<joinCondition>` est fonction des états des liens de contrôle qu'elle lie et donc des arcs qui représentent les états de ces liens.

De plus, si une activité X définit une `<joinCondition>`, sa traduction donne lieu aux deux arcs : E_X^{exe}, dont le prédicat correspond à cette `<joinCondition>`, et $E_A^{(s)jf}$ dont le prédicat correspond à la négation de cette dernière. Ainsi, un chemin C incluant un arc du type E_X^{exe} (ou $E_A^{(s)jf}$), dont le prédicat correspond à une `<joinCondition>` (*resp.* à sa négation), est donc censé inclure les arcs qui représentent les états des liens de contrôle qui permettront une évaluation *positive* de ce prédicat et donc de ladite `<joinCondition>` (*resp.* de sa négation). Si ce n'est pas le cas, le chemin est classé comme infaisable.

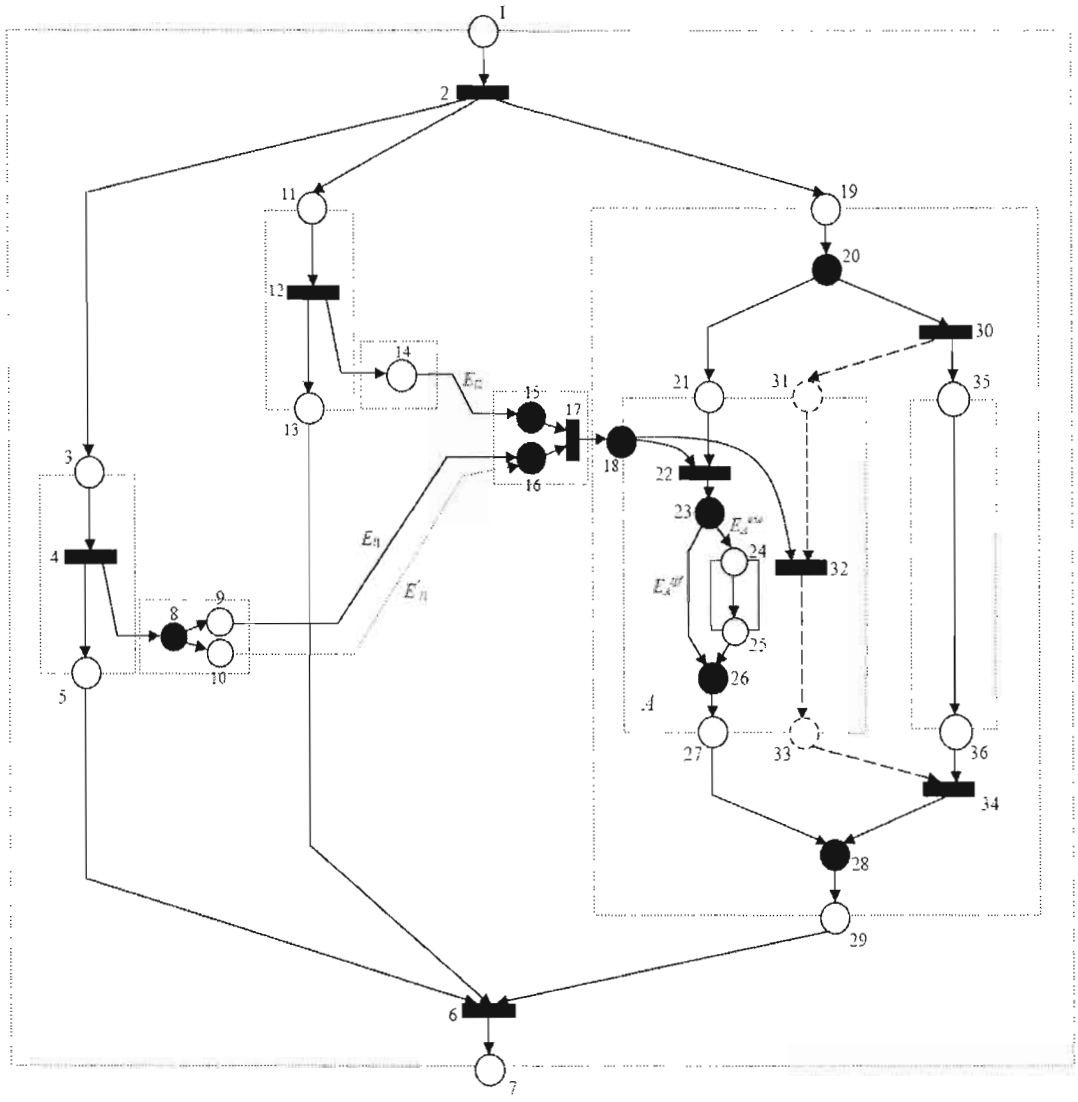


Figure 5.3 Un graphe B-CFG dont le parcours génère des chemins de test infaisables.

Comme exemple, dans le graphe **B**-CFG de la figure 5.3, la traduction d'une activité A qui est cible de deux liens de contrôle l_1 et l_2 (respectivement traduits par E_{l1}/E'_{l1} et E_{l2}) donne en particulier lieu à deux arcs : E_A^{exe} (i.e., (23,24)) dont le prédicat correspond à la `<joinCondition>` définie par cette activité, et à un deuxième E_A^{sjf} dont le prédicat correspond à la négation de cette `<joinCondition>`. Si cette `<joinCondition>` est définie comme une conjonction des états des deux liens l_1 et l_2, le chemin qui inclura l'arc E_A^{exe} doit aussi inclure les arcs E_{l1} et E_{l2} (représentant l'état *exécuté* de l_1 et l_2) pour que l'évaluation de ladite `<joinCondition>` qui correspond au prédicat de E_A^{exe} soit positive. Autrement, le chemin sera considéré comme infaisable[5]. De même, si un chemin de ce graphe inclut l'arc E_A^{sjf}, il est censé aussi inclure les arcs E'_{l1} et E_{l2} pour que l'évaluation de la négation de ladite `<joinCondition>` qui correspond au prédicat de E_A^{sjf} soit positive, sinon le chemin sera considéré comme infaisable.

3. Dans un graphe **B**-CFG traduisant un processus BPEL, il pourrait y avoir des chemins infaisables qui sont de la même famille des chemins non exécutables qu'on trouverait dans un graphe CFG traduisant un programme séquentiel. Autrement dit, dans un graphe **B**-CFG, on pourrait trouver un chemin qui sera jugé infaisable à cause d'une incohérence entre les intervalles de valeurs étant exigées, pour une variable et/ou plusieurs variables qui sont corrélées, par les conditions (prédicats des arcs) que ce chemin inclura. De fait, un tel chemin ne peut être faisable car il n'y aura aucun ensemble de données de test qui saura satisfaire tous les prédicats que ce chemin inclura. Dans ce cas, on dit qu'il n'y aura aucun ensemble de données de test qui *sensibilisera* ce chemin.

   Comme exemple, considérons un chemin qui inclut deux arcs dont les prédicats respectifs sont $pr_1 = (x > 1)$ et $pr_2 = (x < -3)$, et où la valeur de la variable x n'est jamais sujet à modification. Ce chemin sera jugé infaisable car il n'y aura pas une donnée de test pour x qui saura satisfaire les deux conditions $(x > 1)$ et $(x < -3)$.

Dans ce qui suit, nous décrivons comment chacun des chemins infaisables, correspondant à l'un des trois types énumérés ci-dessus, sera écarté de la liste LCT des chemins de test à générer par l'un des deux algorithmes proposés plus-haut.

---

[5] À l'exemple du chemin infaisable |(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (4,8), (8,10), (10,16), (16,17), (17,18), (18,22), (22,23), (23,24), (24,25), (25,26), (26,27), (27,28), (28,29), (29,6), (2,11), (11,12), (12,13), (13,6), (12,14), (14,15), (15,17), (2,19), (19,20), (20,21), (21,22)|.

5.2.3.1 Élimination des chemins infaisables du premier type

Ci-dessous, nous proposons un algorithme qui permettra d'éliminer les chemins infaisables du premier type décrit plus haut. Cet algorithme vérifie, pour chaque noeud de jointure X faisant partie d'un chemin C, si tous les noeuds pour qui ce noeud X est adjacent sont présents dans ce chemin C. Pour ce faire, au cours de la construction d'un graphe B-CFG, nous associons à chaque noeud de jointure X, qui serait défini par ce graphe, une propriété « *adjacencyNbr* » désignant le nombre des noeuds pour qui ce noeud X est un noeud adjacent. Dans un chemin donné, ce nombre sera comparé au nombre d'occurrence des couples (u, v) pour qui $v = X$ — i.e., le nombre de fois que le noeud X est cité comme noeud destination des couples faisant partie d'un chemin donné, et donc le nombre de fois que ce noeud a été lié aux noeuds qui l'ont comme noeud adjacent. Cette vérification a lieu avant l'ajout de tout chemin P à la liste LCT, lors de l'exécution de l'algorithme de génération des chemins de test.

```
feasible <- true
FOREVERY join node v IN P DO
    IF computeOccurrenceNbr(v,P) != v.adjacencyNbr THEN
       feasible <- false
       BREAK
    ENDIF
ENDFOREVERY
IF feasible THEN
    LCT.add(P)
ENDIF
```

L'algorithme donné ci-dessus fait appel à la méthode « $computeOccurrenceNbr(v, P)$ » qui permet de compter le nombre d'occurrence d'un noeud de jointure v comme noeud destination des couples faisant partie d'un chemin P.

5.2.3.2 Élimination des chemins infaisables du deuxième type

Tel que décrit plus haut, un chemin infaisable du deuxième type est un chemin qui inclut un arc dont le prédicat est une `<joinCondition>` (ou sa négation), et cela sans pour autant inclure les arcs représentant les états des liens de contrôle qui permettront une évaluation positive de cette `<joinCondition>` (*resp.* de sa négation). Par ailleurs, nous rappelons que les deux

états *exécuté* et *dead path* d'un lien de contrôle l, qui sont graphiquement représentés par les arcs respectifs E_l et E'_l, sont aussi décrits par la variable globale ***es*** associé à ce lien. Cette variable prend la valeur *true* si l'arc emprunté est l'arc E_l représentant l'état *exécuté* du lien l (i.e., l'élément *ac* de E_l vaut $(es = true)$), tandis que si l'arc emprunté est l'arc E'_l représentant l'état *dead path* de ce lien l, ladite variable prend la valeur *false* (i.e., l'élément *ac* de E'_l vaut $(es = false)$). Nous rappelons aussi qu'une <joinCondition> $JC = \$l_1\ op_1\ \$l_2\ op_2 ... op_n\ \$l_k$ est transformée dans un graphe **B**-CFG en une $JC = es_1\ op_1\ es_2\ op_2 ... op_n\ es_k$, où chaque es_i $(i = 1, \ldots, k)$ correspond à la variable associée au lien de contrôle l_i dont elle décrit l'état.

Ainsi, pour la détection et donc l'élimination des chemins infaisables du deuxième type, nous remplaçons pour toute <joinCondition> $JC = es_1\ op_1\ es_2\ op_2 ... op_n\ es_k$ (ou sa négation) étant présente dans un chemin P, chacun de ses termes es_i par la valeur *true* ou *false*. Le *true* ou *false* qui remplacera une variable es_i correspond à celui attribué à cette dernière par l'élément *ac* de l'un des arcs E_{li} ou E'_{li} inclut dans le chemin P et représentant le lien de contrôle l_i auquel est associée cette variable es_i. Si suite à cette substitution des termes es_i par des *true* et/ou des *false*, l'évaluation de la <joinCondition> JC (ou sa négation) qui lie ces termes ne donne pas un résultat positif, le chemin P est considéré infaisable et ne sera pas ajouté à la liste LCT. Cela se traduit par l'algorithme donné ci-dessous :

```
feasible <- true
FOREVERY join condition JC IN P DO //JC désigne une <joinCondition> ou sa négation
   FOREVERY es IN JC DO //es est la variable associée à un lien l concerné par JC
      IF El IN P THEN //El est l'arc représentant l'état exécuté du lien l auquel
          REPLACE es BY true IN JC //es est associée
      ELSE //cas où c'est l'arc El' qui est inclus par P et non pas l'arc El
          REPLACE es BY false IN JC
      ENDIF
   ENDFOREVERY
   IF !eval(JC) THEN
      feasible <- false
      BREAK
   ENDIF
ENDFOREVERY
IF feasible THEN
    LCT.add(P)
ENDIF
```

L'algorithme donné ci-dessus fait appel à la méthode « $eval(JC)$ » qui permet d'évaluer une condition JC donnée en paramètre et retourne le résultat booléen correspondant à son évaluation. Cet algorithme s'exécute avant l'ajout de tout chemin P à la liste LCT.

5.2.3.3 Élimination des chemins infaisables du troisième type

Tel que décrit plus haut, un chemin infaisable du troisième type est un chemin pour lequel on ne trouvera aucun ensemble de données de test qui saura satisfaire toutes les conditions (prédicats des arcs) qu'il inclut. D'après Xanthakis (74), une détection manuelle des chemins infaisables de ce type « est une pratique très difficile ». Cela nous conduit donc à penser à des méthodes automatiques pour pouvoir détecter ces derniers. Une de ces méthodes les plus usuelles est celle qui, en procédant à l'analyse des prédicats qui sont présents dans un chemin, est capable de déterminer si ce dernier pourrait être sensibilisé ou pas.

Pour ce faire, une telle méthode commence par collecter les prédicats présents dans un chemin donné pour former un système d'équations et/ou d'inéquations. Par la suite, ladite méthode fait appel à un solveur de contraintes (i.e., *constraint solver*) afin de résoudre le système venant d'être formé. Si une solution est trouvée, cela implique que le chemin est faisable, alors que si l'exécution du solveur ne résulte en aucun ensemble de données, ledit chemin est classé infaisable. En effet, l'adoption d'une telle méthode permet de faire d'une pierre deux coups de par son potentiel à vérifier la faisabilité des chemins et, le cas échéant, à générer les ensembles de données de test qui les sensibiliseront. Certains solveurs de contraintes qu'on trouve dans la littérature, et auxquels cette méthode pourrait faire appel, s'exécute en un intervalle de temps raisonnable, ce qui nous éviterait le problème d'indécidabilité pour la faisabilité des chemins. Une description de cette méthode[6] est donnée à la section 5.4.

5.2.4 La génération des chemins pour BPEL : nous et les autres

Principalement, deux approches ont été proposées pour la génération des chemins de test à partir d'un graphe représentant du code BPEL : la première étant proposée par Yuan (81) et Li(26), alors que la deuxième porte l'empreinte de Yan (76). Chacune de ces deux approches se

[6] Cette méthode aurait pu être utilisée pour l'élimination des chemins infaisables du deuxième type puisque dans le fond ces derniers ne sont qu'un cas particuliers des chemins infaisables du troisième type. Or, il se trouve que pour les chemins infaisables du deuxième type, tous les paramètres (i.e. variables es_i décrivant l'état des liens de contrôle) sont connus et que l'on n'aura donc pas besoin d'un solveur de contraintes pour détecter la présence de tels chemins.

base sur une variante du graphe CFG qui est différente de la nôtre.

En particulier, la première approche, proposée par Yuan et Li pour le parcours de leurs graphes BFG, se base et sur l'information graphique et sur les données associées aux arcs et aux noeuds (les post/pré-conditions) de ces graphes. Cela est à l'opposé de notre façon de faire qui, pour le parcours des graphes **B**-CFG, n'a recours qu'à l'information graphique représentée par ces derniers. En fait, si l'approche de Yuan et Li fait aussi usage des données présentes dans leurs graphes BFG, c'est que l'information graphique représentée par ces derniers n'est pas assez riche pour se passer desdites données.

Par ailleurs, cette première approche s'implémente par un algorithme manipulant des matrices. Chacune de ces matrices représente un chemin, et le nombre de ses lignes, ainsi que celui de ses colonnes, est égal au nombre des noeuds du graphe parcouru[7]. Sachant que les chemins pourraient être générés parallèlement par cette première approche, leur représentation en ce genre de matrice pourrait présenter une certaine lourdeur en termes de mémoire, surtout pour le cas des graphes de grandes tailles (e.g., graphe composé de centaines de noeuds ou plus) et abritant un grand nombre de chemin. Toutefois, on souligne que l'approche de Yuan et Li permet de générer les chemins de test tout en évitant de se retrouver avec des chemins infaisables qui seraient similaires à ceux du premier et deuxième type.

En ce qui concerne la deuxième approche proposée par Yan, elle commence par parcourir ses graphes XCFG comme s'ils étaient des graphes CFG ordinaires pour n'en extraire que des chemins séquentiels. Par la suite, ladite approche procède, s'il y a lieu, à la combinaison de ces derniers pour obtenir des chemins concurrents qui seraient faisables. Or, une fois qu'il s'agit de combiner des chemins pour en obtenir d'autres qui soient faisables, les choses se complique vu qu'il va falloir, d'une part, vérifier quels chemins seraient combinables, et d'autre part, fixer le nombre maximal des chemins qu'on peut combiner pour obtenir un nouveau. Potentiellement, cette deuxième approche aura à traiter un grand nombre de chemins infaisables.

Finalement, si Yuan et Li ont proposé une représentation formelle (en matrice) qui n'est pas adaptée au chemin de test à générer, Yan n'a proposé qu'une représentation informelle qui se trouvera limitée, voire inappropriée si jamais on pense l'adopter pour notre cas.

---

[7]Les indices des lignes (ou des colonnes), d'une matrice M_C représentant un chemin C, correspondent respectivement aux noeuds n_i du graphe BFG dont C a été extrait. Chaque élément $M_C[i][j]$ de la matrice M_C représente un arc $< n_i, n_j >$ qui serait présent dans le graphe BFG parcouru et qui ferait partie du chemin C (81).

5.3 Critères de couverture

Pour notre travail, nous avons adopté différentes techniques de test qui toutes se basent sur la couverture du flot de contrôle. Ces techniques diffèrent quant à l'ensemble des chemins qui devront être mobilisés pour la couverture de certaines sous-structures du graphe de contrôle. Au fait, chacune de ces techniques a comme objectif de proposer un critère de couverture donné. Chaque critère correspond à une méthode de sélection qui fournit un ensemble fini de chemins de contrôle permettant la couverture des sous-structures visées dans le graphe considéré. La satisfaction d'un critère correspond donc à trouver les données de test qui sensibilisent ces chemins de contrôle. Dans ce qui suit, nous énumérons et décrivons les critères de couverture pour lesquels nous avons opté et que nous avons dû adapter pour notre travail.

5.3.1 Couverture de tous-les-chemins

Le critère « tous-les-chemins » exige la couverture de tout chemin étant présent dans le graphe **B**-CFG traduisant le code à tester. Ainsi, pour la satisfaction d'un tel critère, il faut, en premier, extraire tous les chemins de contrôle du graphe considéré, i.e. extraire tous les chemins partant du noeud d'entrée au noeud de sortie de ce graphe. Cela se fait par l'application de l'algorithme de génération des chemins, donné à la section 5.2.2, sur le graphe en question. Par la suite, nous produisons les données de test qui permettront de sensibiliser les chemins de contrôle ainsi générés, et cela par l'application de la méthode décrite à la section 5.4. Si des données de test sont trouvées pour tous les chemins de contrôle étant générés, le critère « tous-les-chemins » est donc satisfait.

En revanche, et comme nous l'avons souligné précédemment, parmi les chemins de contrôle générés à partir d'un graphe donné, il pourrait y avoir des chemins infaisables. Par conséquent, un certain nombre des chemins présents dans ce graphe ne pourra être couvert, et le critère « tous-les-chemins » ne pourra ainsi être satisfait. Dû à l'éventuelle présence de ces chemins non-exécutables, la couverture de « tous-les-chemins » dans un graphe est généralement impossible (74; 75). C'est pour cela d'ailleurs qu'on parle souvent du critère de couverture de « tous-les-chemins **exécutables** » et non pas de « tous-les-chemins » dans l'**absolu**. L'impossibilité de réaliser un test avec une couverture de « tous-les-chemins » dans l'absolu nous amène à considérer, dans ce qui suit, des méthodes de couverture partielle que nous espérons, contrairement à cette couverture de « tous-les-chemins », pouvoir satisfaire dans l'absolu.

5.3.2 Couverture des chemins de base

Une des méthodes de couverture partielle, qu'on pourrait adopter comme alternative à un test de « tous-les-chemins » qui ne pourrait être réalisé dans l'absolu, est la méthode mettant en oeuvre le critère « *tous-les-chemins-indépendants* » (i.e. *chemins-de-base*). La considération de cette méthode de couverture pourrait aussi être due à des raisons budgétaires, en plus de cette éventuelle impossibilité de réaliser le test de « tous-les-chemins » dans l'absolu.

En effet, qu'un programme soit séquentiel ou concurrent, tester *tous* ses chemins consiste généralement en une tâche coûteuse en termes de temps et d'argent. McCabe (39) souligne le même problème quand il dixit : « malgré qu'il est possible d'exprimer tous les chemins d'un programme moyennant des expressions algébriques, considérer tous ces chemins s'est avéré non pratique ». En réaction à cela, nous pourrions donc faire de même que McCabe en ne considérant que les chemins de test dits de base. Ces derniers, mis en combinaison, génèrent tous les chemins de test possibles et forment ainsi ce qu'on appelle communément une *base de chemins*.

Pour tout programme séquentiel, tout chemin de test à générer est nécessairement de nature séquentielle. Par conséquent, la base de chemins à définir, qui n'est autre qu'un sous-ensemble de l'ensemble de tous les chemins présents dans un tel programme, ne peut être constituée que de chemins de test séquentiels. Le nombre de ces chemins de base, dits aussi indépendants, est égal au nombre de McCabe (voir 2.3). En revanche, pour un programme concurrent, en l'occurrence un processus BPEL, il y a de fortes chances qu'il y ait des chemins de test à générer qui ne soient pas séquentiels mais *parallèles*. Par conséquent, la base de chemins à définir, qui n'est autre qu'un sous-ensemble de l'ensemble des chemins présents dans un tel programme, peut être constituée de chemins de test séquentiels et/ou parallèles. Comme on le verra par la suite, cette nouvelle donne nous pousse à repenser le calcul du nombre de McCabe pour les programmes concurrents en général, et pour les processus BPEL en particulier.

Pour l'application de cette couverture des chemins de base à notre graphe **B**-CFG, nous essaierons donc en premier de re-définir le calcul du nombre de McCabe de façon à le rendre sensé pour des programmes concurrents. Cette re-définition sera élaborée en s'épaulant sur la base formelle présentée à la section 2.3. Par la suite, nous fournirons des méthodes pratiques pour l'identification des chemins de base qui permettent de satisfaire le critère « *tous-les-chemins-indépendants* » pour un processus BPEL donné. Dire que ce critère a été satisfait c'est dire qu'un test *structuré* a été réalisé (à ne pas confondre avec le test structurel).

5.3.2.1 Le nombre des chemins de base dans un graphe B-CFG

À tout processus BPEL, on peut associer un graphe **B**-CFG. De par sa définition, le graphe **B**-CFG pourra, à l'image d'un graphe CFG calqué d'un programme séquentiel, être rendu fortement connexe par l'adjonction d'un arc de retour reliant son nœud de sortie à son nœud d'entrée. Cependant, cela signifierait-il pour autant que le nombre de McCabe resterait valable pour le graphe résultant de cette adjonction? La réponse tient également en une question : dans un graphe **B**-CFG, l'*excédent* en arcs sortants, dont dépend le nombre de McCabe, serait-il dû juste aux nœuds de décision de ce graphe ou à d'autres facteurs aussi?

5.3.2.1.1 La non-applicabilité directe du nombre de McCabe pour le B-CFG

En se basant sur les trois faits énumérés ci-dessous, nous démontrerons par l'absurde que le nombre de McCabe, comme il a été défini, ne peut s'appliquer *directement* pour le cas des processus BPEL en particulier, et des programmes concurrents en général :

1. McCabe s'est basé sur la théorie des graphes pour définir sa mesure de la complexité, mais, pour ce faire, il s'en est distingué en ne considérant que les bases de circuits dont les composants (les circuits indépendants) correspondent à des chemins de test définis dans le programme.
2. Dans la définition d'un processus BPEL, il y a de fortes chances qu'il y ait des chemins de test à générer qui ne soient pas séquentiels mais parallèles. Par conséquent, si l'on veut rester conforme à la démarche de McCabe, la base de circuits à définir pour le processus considéré devra être constituée de circuits qui pourraient correspondre à des chemins de test séquentiels et/ou parallèles. En particulier, la base de circuits, d'un **B**-CFG qui est calqué sur un processus BPEL définissant des comportements concurrents, ne peut être constituée que de chemins de test séquentiels car sinon, on aura une partie des circuits de base qui représenteront des pseudo-chemins ne correspondant à aucun scénario d'exécution du processus considéré!
3. Pour un programme séquentiel, le *corollaire 1* donné à la section 2.3 affirme que le nombre de McCabe n'est fonction que de l'excédent en arcs sortants généré par les nœuds de décision du graphe de contrôle étant associé à ce programme. Or, dans le cas d'un graphe **B**-CFG, on constate que l'excédent en arcs sortants pourrait être généré par ses noeuds de bifurcation en plus des nœuds de décision qui y figurent.

Démonstration

Hypothèse : Supposons que le nombre de McCabe s'applique directement pour le cas des processus BPEL (i.e. des graphes **B**-CFG qui leur sont associés).

Dire que le nombre de McCabe s'applique directement pour le cas des processus BPEL, c'est dire que le calcul de ce nombre pour un graphe **B**-CFG se fait tout simplement en comptant le nombre d'arcs e et de nœuds n dans ce graphe et en appliquant par la suite la formule : V(G) = e − n + 2. Cela revient donc à calculer l'excédent en arcs sortants qui est généré par chacun des nœuds du graphe **B**-CFG considéré.

De plus, le fait (3.) précise que cet excédent est en effet égal à l'excédent en arcs sortants des nœuds de décision et de bifurcation du graphe **B**-CFG considéré.

Donc, en définitive, les nœuds de bifurcation seront confondus aux noeuds de décision et ainsi le graphe **B**-CFG sera traité comme étant un graphe CFG calqué d'un programme séquentiel. Il paraît alors que cette confusion (i.e. traiter le nœud de bifurcation comme s'il était un nœud de décision) nous ramène naturellement au calcul du nombre cyclomatique, et vide ainsi le nombre de McCabe de son sens (i.e. nombre des chemins de base à tester et non pas de n'importe quel circuit (chemin) indépendant).

Donc, puisque les nœuds de bifurcation ont été vidés de leur sémantique concurrente, tous les circuits qui seront considérés dans le graphe **B**-CFG représenteront des chemins séquentiels, i.e. des chemins de test complets séquentiels qui, à la base, n'incluent aucun comportement concurrent, ou des chemins séquentiels *incomplets* qui ne sont dans le fond que des composantes de chemins de test complets mais parallèles.

Donc, forcément, la base de circuits se trouvera formée exclusivement de circuits représentant des (pseudo)chemins séquentiels, et ce quelle que soit la nature du processus considéré.

Or, d'après le fait (2.), cette base de circuits, surtout dans le cas d'un processus BPEL définissant des comportements concurrents, ne peut être constituée de circuits ne représentant que des (pseudo)chemins séquentiels.

Suite à cette contradiction, on conclut la *non-applicabilité directe* du nombre de McCabe pour les processus BPEL.

□

Toutefois, il faut noter que l'application directe du nombre de McCabe pour le cas des graphes **B**-CFG mène à la définition d'une base de circuits qui, d'un point de vue vectoriel (grossier), reste valable. Dans ce cas, on devrait plutôt parler de nombre cyclomatique et non pas de celui de McCabe. Cependant, notre objectif ne se restreint pas à définir une base qui ne soit valable que d'un point de vu vectoriel (grossier), mais s'étale au fait que la base escomptée devra regrouper des circuits qui correspondent à des chemins de test à travers le processus BPEL considéré. C'est de cette façon que l'on respectera l'idée qui était à la source de la définition du nombre de McCabe. L'idée qui tient sa légitimité de par le fait qu'elle cherche à travers la définition d'une base de circuits, à calculer le nombre maximal des chemins de test indépendants qu'on doit considérer.

Pour ce faire, nous essaierons dans ce qui suit de re-définir le calcul du nombre de McCabe pour le cas des processus BPEL de sorte à ce que ce nombre soit *sensible* à la sémantique des différents types de nœuds du graphe **B**-CFG. Une attention particulière sera portée aux deux types de nœuds : de décision et de bifurcation, de sorte à les différentier et à associer à chacun d'eux un traitement adéquat.

5.3.2.1.2 Le nombre concurrent de McCabe

Considérons les deux graphes G et G' donnés ci-dessous :

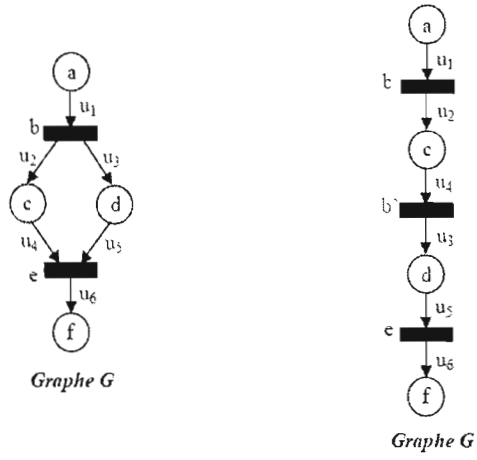


Figure 5.4 Exemple de deux graphes « vectoriellement équivalents ».

Principalement, chacun des deux graphes ne définit qu'un seul chemin de test. Le graphe G définit le chemin μ = [(a,b), (b,c), (c,e), (b,d), (d,e) (e,f)], tandis que le graphe G' définit

le chemin β = [(a,b), (b,c), (c,b'), (b',d), (d,e) (e,f)]. Cependant, par l'application directe du nombre de McCabe sur les deux graphes, on trouve : V(G) = 6 (arcs) - 6 (noeuds) + 2 = 2, et V(G') = 6 (arcs) - 7 (noeuds) + 2 = 1. Le résultat du calcul pour le graphe G' est normal, tandis que pour le graphe G le résultat est contradictoire vu qu'il dicte que le nombre des chemins de base dans le graphe devrait être égal à 2, alors qu'en réalité le graphe n'en définit qu'un seul!

Par ailleurs, on constate que le chemin $\mu = [u_1, u_2, u_4, u_3, u_5, u_6] = (1,1,1,1,1,1)$ de même que le chemin $\beta = [u_1, u_2, u_4, u_3, u_5, u_6] = (1,1,1,1,1,1)$, et donc vectoriellement parlant les deux chemins sont équivalents. Cela est dû, comme on l'avait vu à la section 2.3, au fait que la représentation vectorielle est « grossière ».

A priori, on peut dire que les deux graphes G et G' sont « vectoriellement équivalents ». Au fait, on peut facilement remarquer que pour passer de G à G', on n'a qu'à éliminer l'effet de concurrence engendré par le nœud de bifurcation b du graphe G en le dupliquant dans le graphe à produire, en l'occurrence dans le graphe G' (i.e. le nœud b du graphe G est dupliqué en b et b' dans le graphe G'). Ainsi, on transforme le graphe G en un graphe séquentiel G'. Dans ce cas, on peut parler d'une « *séquentialisation* » du graphe G.

Principalement, cette « séquentialisation » a permis de diminuer l'excédent en arcs sortants généré par le nœud de bifurcation présent dans le graphe G. En dupliquant ce nœud dans le graphe G', on a alors établi un certain équilibre entre le nombre total des arcs et celui des nœuds présents dans le graphe. Cela permet d'une part de résoudre le problème de la contradiction qui s'est manifestée entre le nombre de McCabe et le nombre réel des chemins de test dans le graphe G, et d'autre part d'expliquer pourquoi ce nombre était sans contradiction aucune dans le cas du graphe G'.

Dans ce qui suit, nous allons essayer de tirer profit du concept de « séquentialisation » des graphes pour re-définir le calcul du nombre de McCabe pour le cas des processus BPEL. Cette re-définition donnera lieu à un nouveau nombre s'appliquant pour de tels processus et que nous baptisons « *nombre concurrent de McCabe* ». Commençons d'abord par généraliser le concept de « séquentialisation » pour tout graphe **B**-CFG.

Séquentialisation La séquentialisation d'un graphe **B**-CFG G consiste à éliminer l'effet de concurrence engendré par tout nœud de bifurcation figurant dans ce graphe. Cette élimination se fait en dupliquant, dans le graphe G' à produire, tout nœud de bifurcation b du graphe G en un nombre de fois égal à $d^+(b) - 1$ (i.e. nombre des branches que b engendre dans le

graphe G, moins l'unité). Cette séquentialisation maintient l'ordre des parties séquentielles du graphe G, mais prend en **bloc** chacune des branches d'un nœud de bifurcation b et la rend en ordonnancement séquentiel avec le reste des branches du même nœud b. Le noeud source de chacune de ces branches ne sera plus le nœud b mais le nœud résultant de la duplication de b.

La séquentialisation est à ne pas confondre avec la « sérialisation » car cette dernière ne prend pas comme unité de travail une branche *en entier* pour l'ordonnancer séquentiellement avec le reste des branches du même nœud de bifurcation, mais va au-delà de cela en définissant l'ordre entre les branches, **activité par activité**.

La propriété de la représentation vectorielle, décrite dans la section 2.3, nous permet d'affirmer que tout graphe G' résultant de la séquentialisation d'un graphe G est vectoriellement équivalent à ce dernier. Cela signifie que pour tout chemin de test P, issu du graphe originel G, il existe un chemin de test P', issu du graphe G', et qui est vectoriellement équivalent à P. On en déduit que trouver une base de chemins pour le graphe G revient à trouver une pour le graphe G'. Au fait, vu que le graphe G' est par définition séquentiel (contrairement au graphe G qui pourrait être parallèle), on a donc intérêt à définir une base de chemins pour G', puisque le nombre de McCabe s'y applique directement, et par la suite associer la base résultante au graphe originel G. D'où la définition :

Définition 6 Le « *nombre concurrent de McCabe* » d'un graphe **B**-CFG G correspond au nombre de McCabe du graphe G' résultant de la séquentialisation du graphe G. Ce nombre, noté par **VC**(G), est égal au nombre maximal des chemins de base du graphe G.

5.3.2.1.3 Exemple

Considérons les deux graphes G et G' relatés par la figure 5.5. Le graphe G' correspond à la séquentialisation du graphe G. D'après la *définition 6* donnée ci-dessus, le nombre concurrent de McCabe du graphe G est égal au nombre de McCabe du graphe G' :

$$\mathrm{VC(G)} = \mathrm{V(G')} = 15\ \text{(arcs)} - 15\ \text{(noeuds)} + 2 = 2$$

Le nombre concurrent de McCabe du graphe G est égal à 2, tandis que si l'on avait appliqué directement le nombre de McCabe pour ce même graphe G, le résultat aurait été 4 !

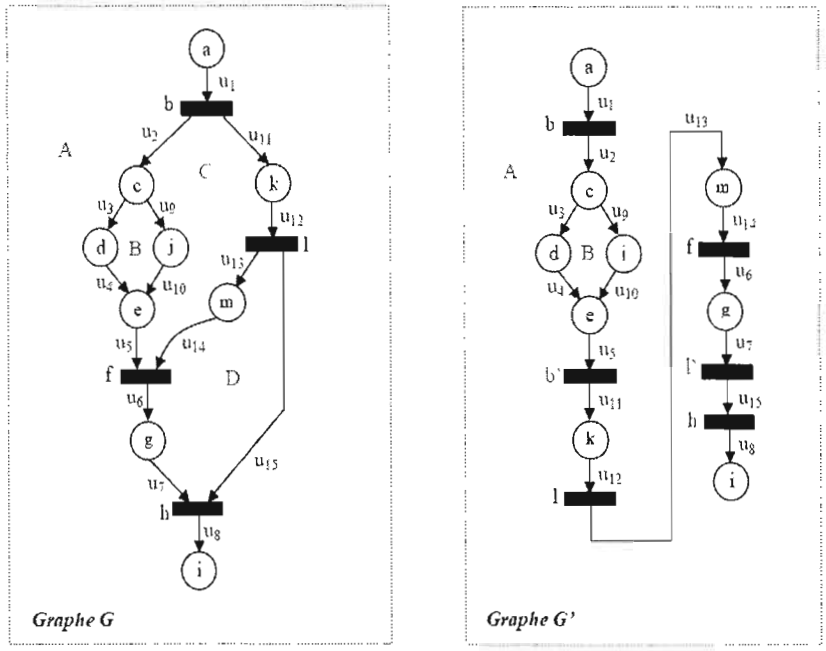


Figure 5.5 Exemple de séquentialisation d'un graphe.

5.3.2.1.4 Simplification du calcul du nombre concurrent de McCabe

Le calcul du « nombre concurrent de McCabe » pour un graphe **B**-CFG passe par la séquentialisation de ce dernier. Or, il se trouve qu'en pratique le passage par cette séquentialisation alourdit la procédure de calcul de ce nombre. Pour pallier à cette lourdeur, nous proposons deux nouvelles procédures qui permettrons de calculer ledit nombre d'une façon plus simple, sans passer par la séquentialisation.

Première procédure Selon la *définition 6* donnée ci-dessus, le « nombre concurrent de McCabe » d'un graphe **B**-CFG G correspond au nombre de McCabe du graphe G' résultant de sa séquentialisation, i.e. VC(G) = V(G'). Étant donné que ce graphe G' est séquentiel, le calcul de son nombre de McCabe peut donc se faire, comme le souligne le *corollaire 1* vu à la section 2.3, en terme de l'excédent en arcs sortants des noeuds de décision qui y sont présents.

Par ailleurs, nous constatons que, dans les deux graphes G et G', on retrouve les mêmes noeuds de décision, en forme et en nombre, car la séquentialisation n'affecte pas les branches engendrées par ces noeuds. En effet, puisque le nombre V(G'), qui est égal à VC(G), n'est fonction que de l'excédent en arcs sortants des noeuds de décision présents dans le graphes G',

noeuds qu'on retrouve aussi dans le graphe G, il vaut donc mieux calculer cet excédent dans ce graphe G et se passer de sa séquentialisation G'. Autrement dit, le « nombre concurrent de McCabe » pourra être directement calculé en termes de l'excédent en arcs sortants des noeuds de décision présents dans le graphe G, et cela tout en excluant l'excédent en arcs sortants de ses éventuels noeuds de bifurcation.

Ainsi, pour un graphe **B**-CFG G et un congénère G' résultant de sa séquentialisation, on a : $VC(G) = V(G') = \sum(d^+(\delta_i) - 1) + 1$, avec $i = 1, \ldots, k$ et k est le nombre des nœuds de décision présents dans le graphe G (les mêmes que ceux qui sont présent dans G'). Ce faisant, cela nous évitera de « séquentialiser » le graphe G pour le calcul de son nombre concurrent de McCabe.

Deuxième procédure À la section 2.3, nous avons vu que tout graphe séquentiel planaire divise le plan le contenant en un nombre de régions r, et que calculer le nombre de McCabe pour ce graphe revient à calculer ce nombre r, i.e. calcul par *inspection visuelle*. Ces r régions qu'on dénombre sont tracées par les branches sortant des noeuds de décision du graphe considéré.

Vu que tout graphe G', résultant de la séquentialisation d'un graphe **B**-CFG G, est un graphe séquentiel, le calcul de son nombre de McCabe pourrait donc être basé sur le nombre des régions que les branches de ses noeuds de décision tracent dans le plan le contenant. Outre cela, nous constatons que, dans les deux graphes G et G', on retrouve les mêmes noeuds de décision, en forme et en nombre, car la séquentialisation n'affecte ni ces noeuds ni les branches qui en sortent.

Ainsi, sachant que le nombre V(G'), qui est égal à VC(G), n'est fonction dans le cas d'une inspection visuelle que du nombre des régions tracées par les branches des noeuds de décision présents dans le graphes G', noeuds qu'on retrouve aussi dans le graphe G, il vaut mieux donc compter ces régions dans ce graphe G et se passer de sa séquentialisation G'. Autrement dit, le « nombre concurrent de McCabe » pourra être directement calculé en terme du nombre des régions tracées par les branches des noeuds de décision présents dans le graphe G, et cela tout en excluant les régions tracées par les branches de ses éventuels noeuds de bifurcation.

Comme exemple à cela, la figure 5.5 montre que dans le graphe G et sa séquentialisation G', on retrouve les deux régions A et B qui sont tracées par les branches du noeud de décision "c" présent dans les deux graphes. Ainsi, le nombre concurrent de McCabe peut être directement calculé à partir du graphe G, en ne tenant compte que des deux régions A et B et tout en excluant

les régions C et D qui sont tracées par les branches des noeuds de bifurcation "b" et "l" présents dans ce graphe. Ce faisant, on obtiendra : VC(G) = V(G') = 2, et non pas 4!

5.3.2.1.5 Comparaison avec d'autres travaux

Pour autant que nous sachions, les travaux qui ont voulu étendre le nombre de McCabe, pour des programmes/processus concurrents, ont re-défini ce nombre pour refléter le niveau de complexité de ces derniers mais sans pour autant que cela puisse refléter l'effort requis pour les tester (i.e. sans pour autant que cela puisse refléter le nombre des chemins de base à travers de tels programmes ou processus).

Comme exemples de tels travaux, celui de Benwell (2) ainsi que celui de De Paoli (49) évoquent une même formule qui étend l'application du nombre de McCabe pour tout réseau de Petri PN étant l'image d'un programme séquentiel ou concurrent. Ladite formule se définit en considérant le réseau de Petri concerné comme un graphe orienté ayant comme noeuds ses p places et ses t transitions, et comme arcs ses e arêtes, ainsi : $V(PN) = e - (p + t) + 2$. Ce faisant, les deux auteurs ne différencient pas entre les types de noeuds que les places et transitons pourraient incarner dans un réseau de Petri modélisant un programme concurrent, et par suite leur formule ne peut aboutir à une complexité qui se veut le reflet du nombre des chemins de test indépendants présents dans un tel programme.

De son côté, Cardoso (7) a étendu le nombre de McCabe pour définir sa complexité à l'égard des processus métier — processus qui pourraient inclure des comportement concurrents. Au fait, Cardoso a défini sa propre complexité suite au constat que le nombre de McCabe ne peut être directement appliqué pour le calcul d'une complexité étant basée sur les graphes de contrôle qu'il a défini comme représentation des processus métiers — ces graphes définissent des types de noeuds dont le calcul du nombre de McCabe ne tient pas compte, tandis que la nouvelle complexité en est fonction. En revanche, cette nouvelle complexité a été définie en terme des états d'exécution (états mentaux) possibles dans un processus, et non pas en terme du nombre des chemins de test indépendants qui y sont présents. Ce faisant, Cardoso a voulu une complexité qui a vocation de mesurer la difficulté à analyser, à comprendre ou à expliquer un processus métier, mais non pas de mesurer la difficulté à le tester.

Les deux travaux qui se rapprochent le plus du nôtre, de par leur considération du nombre de McCabe comme mesure pour le nombre des chemins indépendants, sont ceux de Lertphumpanya (24) et de Yan (76). Or, il se trouve que ces derniers ont considéré le nombre de McCabe

comme il est, et n'ont pas cherché à le re-définir. Cette considération est inappropriée car un graphe traduisant du code BPEL inclurait des noeuds ayant des sémantiques différentes dont il faut tenir compte lors du calcul dudit nombre. En particulier, Yan aurait dû re-définir ce nombre car il envisage le traitement du code BPEL à comportement séquentiel et/ou concurrent, contrairement à Lertphumpanya qui ne traite que les constructeurs séquentiels, conditionnels et répétitifs du BPEL et exclut toute structure à comportement concurrent.

5.3.2.2 La génération des chemins de base

Pour faciliter la tâche à un testeur souhaitant réaliser une couverture de « tous-les-chemins indépendants », il est nécessaire de lui proposer une façon de faire qui génère *automatiquement* l'ensemble des chemins de base à exécuter pour mener à bien cette couverture, et ainsi lui éviter d'avoir à tâtonner jusqu'à trouver ces chemins. Généralement, une telle façon de faire consiste en une méthode qui permet d'identifier l'ensemble des chemins linéairement indépendants en parcourant le graphe de contrôle traduisant le programme à tester.

Une des méthodes les plus simples, qui permet cette identification des chemins indépendants dans un graphe CFG séquentiel, est celle proposée par Poole (51). L'idée principale derrière cette méthode est de choisir, parmi tous les arcs sortant de tout noeud de décision u, un seul arc qui sera nommé « *arc par défaut* » — cet arc sera toujours le seul à être choisi à chaque fois que le noeud u est revisité, tandis que les autres arcs sortant de ce même noeud ne seront traversés qu'une seule fois lors du parcours du graphe G considéré. Procédant de la sorte, cette méthode génère exactement V(G) chemins de test indépendants.

Pour notre cas, nous avons retouché le pseudo-code donné au paragraphe 5.2.2.3 de façon à pouvoir adapter la méthode de Poole pour son application sur les graphes **B**-CFG — une adaptation qui permettra la génération automatique des chemins de base présents dans de tels graphes. Cette adaptation, qui passe donc par la mutation de notre pseudo-code, s'est faite assez naturellement en raison de la simplicité de la méthode et son fondement sur un parcours en profondeur tout comme notre pseudo-code à muter.

Étant donné que la méthode de Poole ne vise que les noeuds de décision du graphe à parcourir, son adaptation pour le cas des graphes **B**-CFG se fera donc en n'agissant que sur la partie de notre pseudo-code qui traite le cas où le noeud visité correspond à un noeud de décision. Ainsi, au niveau de cette partie, nous agissons de sorte que les arcs sortant du noeud de décision traité soient tous traversés si ce noeud n'a pas été déjà visité, alors que s'il l'était déjà, seul

son « arc par défaut » sera emprunté. Au fait, tout noeud de décision u, figurant dans un graphe **B**-CFG G, aura comme « arc par défaut » l'arc incarné par le couple (u, v), où v est le noeud figurant comme *premier* élément de la liste d'adjacence de u. Ce faisant, le pseudo-code résultant nous permettra de générer exactement VC(G) chemins indépendants. Ces derniers constitueront une base qui saurait satisfaire le critère « tous-les-chemins indépendants ».

Pour concrétiser tout cela, nous associons à chaque noeud de décision la propriété « *visited* » qui, prenant pour valeur *true* ou *false*, permet d'indiquer si, lors du parcours du graphe, le noeud en question a été déjà visité ou pas. Outre cela, le pseudo-code résultant fera usage de la méthode « *firstAdjNode()* » qui, appliquée sur un noeud de décision u, retourne le noeud figurant comme premier élément de la liste d'adjacence de ce noeud u. Ci-dessous, nous présentons le pseudo-code résultant en ne mentionnant que la partie que nous avons dû modifier par rapport à la version originale. À la fin d'exécution de ce pseudo-code, les VC(G) chemins indépendants seront tous générés et contenus dans la liste LCT.

```
... ...
WHILE true DO
  SWITCH(u)
    CASE a decison node :
        IF u.visited THEN
          v <- firstAdjNode(u)
          currentPath.add((u,v)) //Ajout de l'arc par défaut
          u <- v
        ELSE
          u.visited <- true
          FOREVERY v IN Adj[u] DO
            tempCurrentPath <- createPathCopy(currentPath)
            tempPB <- createPBCopy(PB)
            tempCurrentPath.add((u,v))
            PD.push(v,tempCurrentPath,tempPB)
          ENDFOREVERY
          u,currentPath,PB <- PD.pop()
        ENDIF
        CONTINUE
    CASE ...
        ... ...
  ENDSWITCH
  ... ...
ENDWHILE
```

Certes, le pseudo-code donné ci-dessus permet de générer d'une façon simple et automatique une base de chemins pour tout processus BPEL à tester, néanmoins il se pourrait qu'un ou plusieurs des chemins de cette base soient infaisables. Dans la pratique, cela pourrait compromettre la satisfaction du critère « tous-les-chemins indépendants ». Afin donc d'éviter le risque de présence de tels chemins dans la base à générer et ainsi assurer une satisfaction à 100% dudit critère, on pourrait envisager une méthode qui, en cours de génération de la base, tient compte de la faisabilité des chemins.

En effet, dans ce qui suit, nous présentons une méthode qui permet de générer, pour tout processus BPEL à tester, une base de chemins qui ne sera constituée que de chemins faisables. Cette méthode est une variante de celle étant proposée par Yan et Zhang (77) et qui, contrairement à d'autres[8], assure la génération *automatique* d'un nombre *suffisant* de chemins pour construire une base; ces chemins sont sélectionnés *à la volée* de sorte à ce qu'ils soient tous faisables et linéairement indépendants. Comme elle l'est décrite ci-dessous, la nouvelle méthode que nous proposons procède en deux étapes principales :

1. Évaluer VC(G) du graphe G représentant le code BPEL à tester. Cette évaluation pourrait se faire par application de la première procédure de calcul fournie au paragraphe 5.3.2.1.4.
2. Générer un nouveau chemin C à l'aide de l'algorithme donné à la section 5.2.2, et cela tant que le nombre de chemins contenus dans la liste LCT est inférieur à VC(G) :
   (a) Tester si le chemin C est faisable. Si c'est le cas, passer à la sous-étape (b), sinon revenir à (2). On rappelle qu'un chemin est dit infaisable s'il est d'un des trois types de chemins qui sont détectables par la méthode ou l'un des deux algorithmes donnés à la section 5.2.3.
   (b) Tester si la liste LCT est vide. Si c'est le cas, ajouter le chemin C à cette liste et revenir à (2).
   (c) Tester si le chemin C est linéairement indépendant des chemins de la liste LCT. Si c'est le cas, ajouter C à cette liste. Ce test pourrait se faire par une méthode qui fait appel à un outil de programmation linéaire comme celui ayant été utilisé par Yan et Zhang (77) pour la même fin.

[8] D'autres méthodes dont celle (18) qui emboîte le pas à Poole en ne prenant pas compte de la faisabilité des chemins, celle (69) qui considère ce critère de faisabilité mais procède d'une façon semi-automatique (i.e., choix des chemins se fait en interaction avec le testeur), et celle (70) qui, certes, procède d'une façon automatique mais n'assure pas la génération d'un nombre suffisant de chemins indépendants pour construire une base.

5.3.3 Couverture de toutes-les-branches

Comme c'était le cas pour la génération des chemins de base, dans cette sous-section nous présentons une méthode pour générer automatiquement l'ensemble des chemins qui permettront une couverture de « *toutes-les-branches* » pour un processus BPEL donné — réaliser une couverture de « toutes-les-branches », revient à couvrir « *tous-les-arcs* » du graphe **B**-CFG traduisant le processus à tester. La méthode proposée ne considère que les chemins faisables en procédant comme suit :

1. Générer un nouveau chemin C à l'aide de l'algorithme donné à la section 5.2.2, et cela tant qu'il y aura au moins un arc qui n'a pas été visité dans le graphe **B**-CFG considéré :
   (a) Tester si le chemin C est faisable. Si c'est le cas, passer à la sous-étape (b), sinon revenir à (1).
   (b) Tester si le chemin C contient au moins un arc du graphe **B**-CFG considéré qui n'a pas été déjà visité. Si c'est le cas, ajouter C à la liste LCT, sinon revenir à (1).
   (c) Marquer tous les arcs, qui sont nouvellement découverts par le chemin C dans le graphe considéré, comme étant visités.

Le nombre maximum des chemins de test que cette méthode est censée générer ne dépassera jamais le « nombre concurrent de McCabe » du graphe **B**-CFG traduisant le processus BPEL à tester. Au fait, selon la hiérarchie établie dans la littérature (75), on classe le test de « tous-les-chemins-indépendants » comme étant "*plus fort*" que le test de « toutes-les-branches » — on dit aussi que le premier inclut le deuxième, et que satisfaire le premier implique la satisfaction du deuxième, mais l'inverse n'est pas toujours vrai.

5.3.4 Critères de couverture 0-1 répétition et 0-2 instances

Lors de la génération des chemins, la présence d'une structure répétitive (comme celle du `<while>`) dans un graphe **B**-CFG peut causer des chemins de taille infinie, de même que l'algorithme de génération des chemins risque de boucler infiniment. Ainsi, il est impératif de penser à un critère qui limite la taille des chemins en limitant le nombre de fois que le corps d'une structure répétitive peut être parcouru — i.e. adopter un critère de couverture qui limite ce nombre de fois à une valeur n. En effet, l'une des pratiques les plus commune pour pallier à ce problème est de considérer le critère « *0-1 répétition* » lors de la génération des chemins de

test. Ce critère limite à *deux* le nombre des scénarios d'exécution pour une structure répétitive : le premier scénario correspond à celui où le corps de la structure n'est pas exécuté (i.e. *zéro-répétition*), alors que le deuxième correspond au scénario où ce corps n'est exécuté qu'une seule fois (i.e. *une-répétition*).

Pour l'adoption du critère « *0-1 répétition* », nous avons choisi de procéder, à l'image des travaux de Yuan (81) et Yan (76), à l'élimination des arcs de retour présents dans les patrons de traduction des constructeurs répétitifs BPEL. Ci-dessous est donné le patron de traduction à considérer pour toute activité `<while>` et toute activité `<forEach>` de nature séquentielle, dans le cas d'adoption dudit critère. De même, pour l'activité `<repeatUntil>`, le patron la traduisant sera privé de son arc de retour, mais seul le deuxième scénario (i.e. *une-répétition*) sera considéré dans ce cas car, par définition, le corps d'un `<repeatUntil>` doit être exécuté *au moins* une fois.

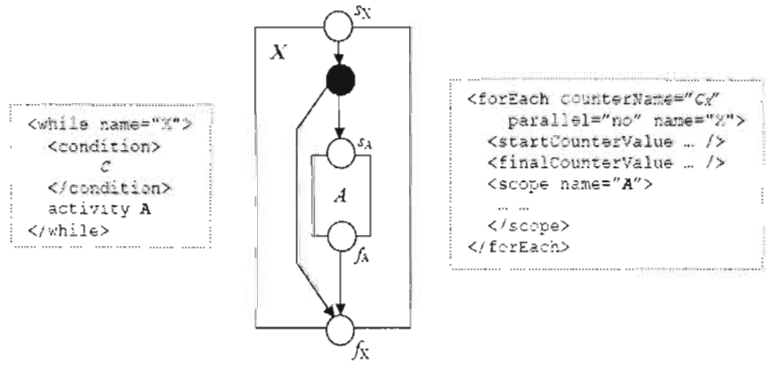


Figure 5.6 Adoption du critère «*0-1 répétition*» pour le `<while>` et le `<forEach>` séquentiel.

Le critère « *0-1 répétition* » pourrait aussi s'appliquer à un eventHandler qui se voit invoqué plusieurs fois d'une façon itérative, i.e. un *eventHandler* résultant en plusieurs instances qui s'exécutent en série. Outre cela, un *eventHandler* pourrait aussi être invoqué plusieurs fois pour résulter en des instances qui s'exécutent, non pas en série, mais en parallèle. Or, vu que cela nécessitera la duplication du sous-graphe B-CFG représentant ledit *eventHandler* autant de fois que ce dernier sera invoqué, et que ce nombre de fois ne peut être prévisible, nous avons donc choisi de limiter ce nombre à *deux* et ainsi éviter d'avoir à manipuler des graphes de taille illimitée. Autrement dit, cela revient à définir le nouveau critère « *0-2 instances* » qui permettra cette limitation. Le nombre "2" a été choisi comme une limite car il représente le nombre minimal d'instances à considérer pour pouvoir tester le comportement concurrent d'un *eventHandler*, et éventuellement tester l'accès aux variables qui seraient partagées par ses instances.

La figure 5.7 dépeint le patron de traduction d'un eventHandler *EH* étant associé à un <scope> *P*. Ce patron matérialise l'adoption du critère « *0-2 instances* » en ne dupliquant qu'une seule fois le sous-graphe représentant l'*eventHandler* traduit. Ledit critère limite à *trois* le nombre des scénarios d'exécution possibles pour un *eventHandler* : (a) aucune instance n'est exécutée (*zéro-instance*), (b) une seule instance est exécutée (i.e., *une-instance*), (c) deux instances sont exécutées en parallèle (i.e., *deux-instances*). En plus, nous soulignons que ce même patron matérialise aussi l'adoption du critère « *0-1 répétition* » vu qu'il n'inclut aucun arc de retour, contrairement au patron décrit par la figure 4.25.

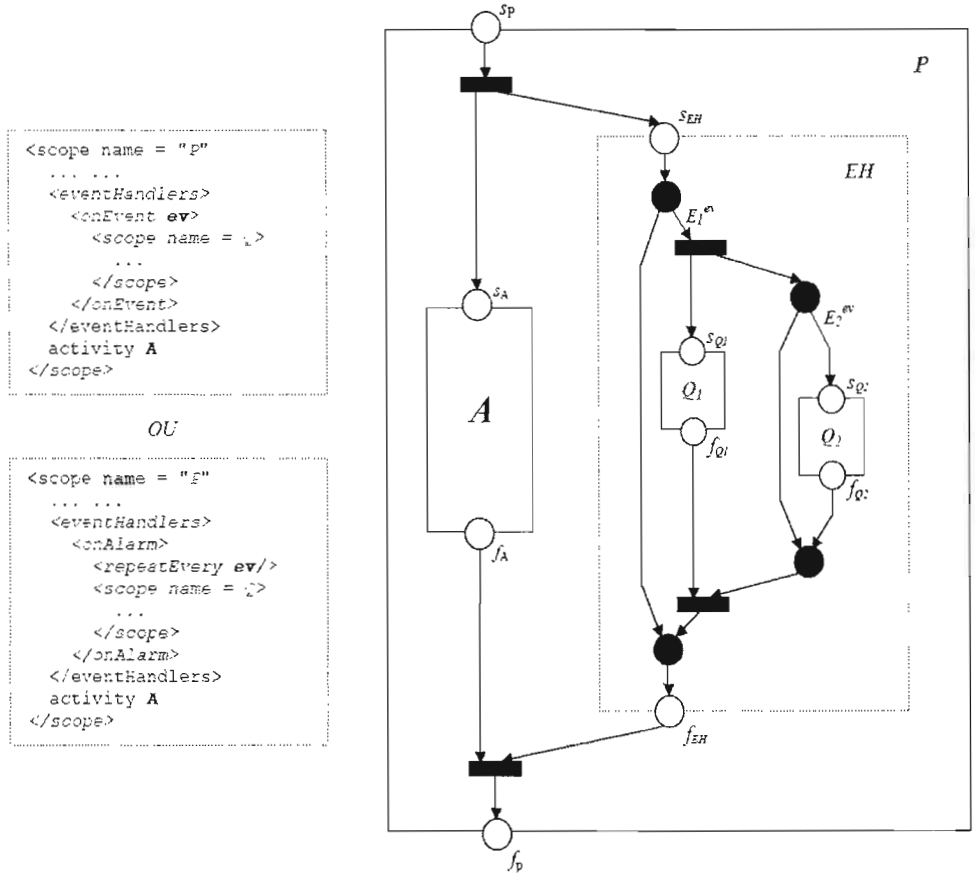


Figure 5.7 Adoption des critères «*0-1 répétition*» et «*0-2 instances*» pour les *eventHandlers*.

Le critère « *0-2 instances* » pourrait aussi s'appliquer aux activités <forEach> qui sont de nature concurrente car leur introduction du *parallélisme dynamique* empêche de prévoir le nombre des branches parallèles qui constitueraient chacune d'elles, i.e. un parallélisme dynamique qui brouille l'estimation de la taille des patrons traduisant de telles activités. Comme le mon-

tre la figure 5.8, l'adoption du critère « *0-2 instances* » limite à *deux* le nombre des scénarios d'exécution possibles pour tout `<forEach>` de nature concurrente : soit que le corps entier du `<forEach>` est contourné, soit que ce corps est exécuté mais avec la contrainte limitant à *deux* le nombre des branches parallèles qui constituent ce corps; chacune de ces branches est une instance du `<scope>` A inclus dans le `<forEach>` traduit.

```
<forEach counterName="Cx"
    parallel="yes" name="X">
  <startCounterValue … />
  <finalCounterValue … />
  <scope name="A">
   … …
  </scope>
</forEach>
```

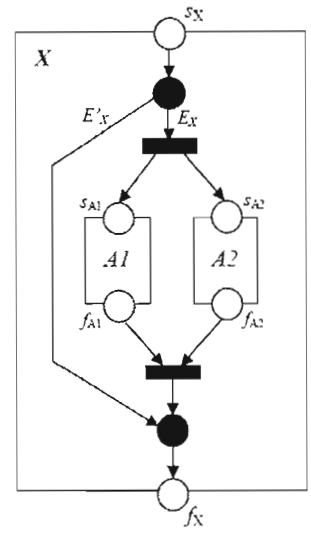


Figure 5.8 Adoption du critère «*0-2 instances*» pour le `<forEach>`.

5.4 Génération des données de test

Nous rappelons que l'approche de génération des cas de test (abstraits) que nous adoptons est une « *approche orientée chemins* » (i.e., *path-oriented approch*). Après les phases de traduction en **B**-CFG et de génération des chemins, ladite approche dicte de passer à une prochaine et dernière phase qui consiste à générer les données de test pour habiller les chemins extraits et de fait constituer les cas de test abstraits; ces derniers sont à transformer, ultérieurement, en des cas de test exécutables. Ainsi, à chaque fois que nous générons un chemin et que nous vérifions qu'il ne correspond pas à un chemin infaisable du premier ou du deuxième type, il faut penser à générer les données de test qui le sensibiliseront. Si de telles données ne peuvent être trouvées, le chemin est classé comme chemin infaisable. Dans ce qui suit, nous nous limitons à l'analyse de cette question de la génération des données de test que nous envisageons de mieux élaborer dans le cadre de futurs travaux.

5.4.1 Une solution à envisager

Principalement, la génération des données de test, qui correspond à cette dernière phase de « l'approche orientée chemins », pourrait être : (a) soit de nature *statique*, et dans ce cas on se contente d'analyser et d'examiner le graphe de contrôle et l'information qu'il inclut, i.e. analyse et examen passif du code à tester, (b) soit de nature *dynamique*, et dans ce cas il y a nécessité d'exécuter le code à tester (41). Pour notre travail, nous pourrions procéder par une génération de données de nature statique vu qu'elle est moins coûteuse que celle de nature dynamique (12). Pour ce faire, cette génération commencera par collecter les prédicats présents dans un chemin donné pour former un système d'équations et/ou d'inéquations qu'elle tâchera par la suite de résoudre pour ainsi aboutir aux données de test qui sensibiliseront ce chemin.

Deux des façons de faire les plus usuelles qui permettent la collection des prédicats sont (82) : la « *forward expansion* » qui, en partant du noeud d'entrée au noeud de sortie du chemin considéré, résulte en une/des expression(s) symbolique(s), et la « *backward substitution* » qui, contrairement à la « *forward expansion* », collecte les prédicats en partant du noeud de sortie au noeud d'entrée du chemin considéré, et cela tout en veillant à substituer les variables touchées par les opérations d'affectation qui sont traitées dans l'ordre selon lequel elles ont été visitées.

Pour de futurs travaux, nous pourrions emboîter le pas à Yan (76) et procéder par une « *backward substitution* » en adaptant son algorithme de collection des prédicats pour notre graphe **B**-CFG — algorithme dont l'exécution résulte en un système de contraintes. De même, pour résoudre un tel système de contraintes et ainsi générer les données de test, nous pourrions faire comme Yan en faisant appel à un solveur de contraintes (e.g., *LP_Solve* (4)). Finalement, nous soulignons que la génération des données de test, pour les variables qui ne sont pas utilisées dans le systéme de contraintes résultant (i.e., une sorte de variables libres), pourrait être accomplie en faisant appel à un outil de génération aléatoire de données (81).

5.4.2 Le problème des variables partagées

La définition de certains processus BPEL pourrait exprimer une relation d'ordre partiel entre les activités qu'elle met en jeu, surtout si ces activités sont définies de sorte à s'exécuter en parallèle. Si de telles activités accèdent, en mode lecture et écriture, à des variables qu'elles partagent, il se pourrait qu'un certain non-détermisme de l'exécution soit causé, de même que

l'opération de génération des données de test soit brouillée. Pour remédier à un tel effet, il faut entreprendre l'une des deux démarches suivantes :

1. Définir au préalable l'ordre d'exécution de ces activités partageant les mêmes variables, et cela en agissant sur le code à tester ou bien sur le graphe le traduisant — i.e. prédéfinir l'ordre d'accès en lecture et écriture aux variables partagées au niveau du code ou du graphe considéré.
2. Prendre compte de cet ordre lors du déroulement de la génération des données de test, et cela sans avoir à le prédéfinir au niveau du code ou du graphe considéré.

En ce qui a trait à la première démarche, elle pourrait être implémentée sous deux formes. La première forme consiste à agir directement sur le code BPEL à tester, et cela en enrichissant ce dernier par de nouveaux liens de contrôle qu'on définit entre ses activités accédant concurremment à des variables communes. Ces nouveaux liens régissent l'ordre d'exécution de telles activités, et arrivent ainsi à prédéfinir l'ordre d'accès aux variables partagées par ces dernières. Contrairement à la première forme, la deuxième parvient à la pré-définition de l'ordre en agissant non pas sur le code BPEL à tester mais sur le graphe B-CFG le traduisant. Concrètement[9], cette deuxième forme consiste à doter le graphe B-CFG considéré de nouveaux arcs prédéfinissant l'ordre d'exécution des actions qui, étant liées à des arcs partiellement ordonnés, correspondent à des opérations de lecture et écriture sur des variables partagées. Sous ses deux formes, la première démarche est surtout à appliquer si l'ordre à prédéfinir est implicitement connu à l'avance[10] et est unique[11], car sinon il va falloir considérer plusieurs ordres et donc plusieurs variantes du processus ou du graphe B-CFG à traiter.

En fait, dans le cas où l'ordre à prédéfinir ne pourrait être connu à l'avance ou ne pourrait être unique, la deuxième démarche est à adopter. Pour cette deuxième démarche, il ne s'agit pas de prédéfinir un ordre au niveau du code BPEL ou du graphe B-CFG considéré, mais plutôt de tenir compte d'un tel ordre pour pouvoir générer des données de test sensées. Par analogie

---

[9] À l'image de ce que fait Yan (76) pour son graphe XCFG.

[10] Comme ce serait le cas pour deux « *isolated scopes* » S_1 et S_2 dont des activités accèdent concurremment à des variables communes en mode lecture et écriture; cas dans lequel la sémantique du « *isolated scopes* » dicte implicitement à ce que l'ordre d'accès aux variables communes soit prédéfini comme si toutes les activités concernées de S_1 allaient s'exécuter avant celle de S_2 (ou vice-versa) pour finalement aboutir au même résultat (1).

[11] Dans le sens où il n'y aurait pas d'autres ordres d'exécution qui aboutiront à un résultat différent.

à ce qu'on propose pour le test du code concurrent Ada (80), pour chaque chemin C qui sera extrait d'un graphe **B**-CFG et qui inclura des activités accédant concurremment à des variables communes, il faut énumérer différents ordres possibles d'exécution de ces activités. Par la suite, il faut générer différentes données de test pour ce même chemin C en respect de chacun de ces ordres. Dans ce cas on pourrait parler du *test concurrent* comme il était formalisé par Tai (59) – et dans un premier temps par Weiss (71), sans oublier les différents défis qui s'y rapportent (e.g., la *sélection* des ordres d'exécution et le jugement de leur *faisabilité*).

Éventuellement, on pourrait aussi penser à une troisième démarche qui consiste en une *sérialisation* complète du processus à tester. En d'autres mots, il s'agit de définir un *graphe de concurrence* (une sorte de graphe d'accessibilité) à partir du graphe **B**-CFG traduisant le processus considéré, et cela à l'image de ce que Taylor (60) propose pour le test du code concurrent Ada. Toutefois, le problème d'explosion d'états pourrait miner cette éventuelle démarche vu qu'elle consiste à tout sérialiser — même les arcs (du graphe de contrôle) dont les actions n'accèdent pas concurremment à des variables communes ne seront épargnés de cette sérialisation.

Certes, un accès *incontrôlé* à des variables partagées relève d'un problème de mauvaise programmation (76), mais cela n'implique pas pour autant que le partage des variables, en tant que tel, doive être banni dans toutes circonstances. En fait, ce partage pourrait à titre d'exemple servir à l'implémentation d'une forme de synchronisation (54). Ainsi, il est souvent nécessaire de considérer et de traiter la présence des variables partagées dans un processus BPEL à tester, chose que nous envisageons d'étudier en profondeur dans de futurs travaux.

5.5 Les cas de test exécutables

5.5.1 La spécification

Nous rappelons que notre travail entre dans le cadre du test unitaire des processus BPEL. Nous rappelons aussi qu'un processus BPEL pourrait interagir avec plusieurs partenaires et que ces derniers pourraient correspondre soit à des processus complexes ou à de simples services Web sans état (i.e., *stateless*). Ainsi, pour le test unitaire d'un processus BPEL donné, nous avons besoin d'avoir accès au code qui définit sa logique interne, mais aussi d'avoir au moins une idée sur le comportement de ses partenaires, à son égard. En effet, le test unitaire, selon l'approche « *modules isolés* » que nous adoptons pour le cas du BPEL, préconise l'usage des bouchons et des lanceurs qui simulent le comportement des composantes applicatives entourant

l'unité de code à tester. Cela évitera toute dépendance à la disponibilité du code implémentant ces composantes. Pour BPEL, les bouchons et lanceurs à définir, qu'on appelle dans ce cas des *mocks* (34), simuleront les comportements des partenaires interagissant avec le processus BPEL à tester.

Pour un scénario de test donné, l'ensemble des comportements décrits par les *mocks* simulant les partenaires d'un processus P à tester constitue ce qu'on appelle la *logique de test.* Cette logique, combinée aux données de test que nous générons pour le scénario de test considéré, forme un *cas de test exécutable* pour ce processus P à tester. Comme le soulignent Li et Sun (27), la logique de test à spécifier pour un tel cas de test devrait être définie à partir du chemin de test décrivant le scénario de test à simuler.

Au fait, tout chemin de test étant extrait du processus à tester est un ensemble d'activités qui sont tournées soit vers l'intérieur soit vers l'extérieur de ce processus. En particulier, les activités qui sont tournées vers l'extérieur (i.e., liées aux invocations de services, comme le `<receive>`/`<reply>` et `<invoke>`) forment une séquence d'interactions avec les partenaires du processus à tester. Cette séquence peut être séparée pour définir le comportement de chaque *mock* partenaire. Par exemple, pour chaque *mock* on spécifie la série des appels que le partenaire qu'il simule est supposés recevoir de la part du processus à tester, et cela en se référant à ladite séquence. Les *mocks* incluront aussi des données de test et des assertions pour ainsi constituer des cas de test exécutable en bonne et due forme. Les cas de test résultant sont souvent regroupés en une *suite de test* pour rendre le test automatique (exécuter plusieurs cas de test d'un trait) et surtout permettre l'obtention de statistiques d'ensemble (e.g. calcul de la couverture de code).

5.5.2 L'exécution

Après l'étape de spécification des cas de test, l'étape qui suit consiste en l'exécution de ces derniers. Pour mener à bien cette étape, on peut choisir entre l'un des trois types de *frameworks* d'exécution qui tous se rapportent au test unitaire BPEL, à savoir :

1. *Les frameworks B2B (use BPEL to test BPEL).* À la lumière de la famille xUnit, ce type de frameworks utilise BPEL comme langage de spécification des tests en définissant les *mocks* partenaires comme des processus BPEL à code allégé. À titre d'exemple, on peut citer le framework *BUnit*, développé et commercialisé par le groupe *Active Endpoints* (14), ainsi que le *framework* prototype réalisé par Li (28).

2. *Les specialized-language frameworks.* Ce type de *frameworks* utilise un mini-langage XML spécialisé pour la spécification des tests, i.e. définition des *mocks*, des données et des assertions. À titre d'exemple, on peut citer le framework BPELUnit (46). Les grands atouts de ce dernier sont son calcul de la couverture de code suite à l'exécution d'une suite de test, ainsi que sa prise en charge du test des processus de longue durée — traiter des durées d'exécution qui risquent de s'étaler sur plusieurs heures, voire plusieurs jours.
3. *Les object-oriented frameworks.* Ce type de *frameworks* se base sur la programmation orientée objet pour la spécification des tests. Principalement, on peut citer le framework BPEL-Unit (27) qui étend son congénère JUnit. L'atout majeur dudit *framework* est sa simplification de la tâche d'écriture de test pour les développeurs qui sont déjà familiarisés avec JUnit et la programmation orientée objet, et cela en leur épargnant la manipulation de documents XML et d'autres détails de définition de services durant le test.

Finalement, une des choses intéressante qui reste à faire, pour une automatisation complète du processus du test unitaire BPEL, est de développer un module qui permet de convertir automatiquement les chemins et données de test générés (i.e., les cas de test abstraits) en des cas de test exécutables regroupés en une suite de test. Cela pourrait se faire par analogie à ce que propose Lertphumpanya (24), mais tout en respectant le format et le langage de spécification de test qui sont adoptés par le *framework* d'exécution à utiliser — e.g., faut-il générer les *mocks* en BPEL ou en Java?

5.6 Conclusion

À travers ce cinquième chapitre, nous avons pu exposer notre représentation formelle des chemins concurrents de test, nos algorithmes pour la génération de ces chemins, notre adaptation de certains critères de couverture que nous empruntons de la programmation séquentielle, ainsi que les pistes que nous proposons pour la génération des données et l'obtention de cas de test exécutables. Dans le chapitre qui suit, nous présenterons les résultats d'application de notre méthode de génération de cas de test sur deux exemples différents de processus BPEL qui seront décrits avant d'être traités.

CHAPITRE VI

TEST DES PROCESSUS BPEL : ÉTUDES DE CAS

6.1 Introduction

Dans ce chapitre, nous allons appliquer notre méthode de génération de cas de test sur deux exemples de processus BPEL. En particulier, cette application concernera notre traduction du BPEL en **B**-CFG de même que nos algorithmes de génération de chemins de test. Ainsi, après sa description, chacun des deux exemples sera en premier traduit en un graphe **B**-CFG. Après cela, un tableau, regroupant des données statistiques portant sur les propriétés du graphe et des chemins de test générés, sera dressé pour chacun des deux exemples. Finalement les données des deux tableaux seront comparées et des conclusions seront tirées.

6.2 Le processus *Travel*

Le premier exemple de processus BPEL que nous allons aborder est un processus permettant l'achat, en ligne, de billets d'avion pour des voyages d'affaire. Ce processus est une adaptation de celui proposé par Juric et *al.* (19)

7.2.1 Description

Le processus *Travel* qui sera étudié décrit un scénario qui a été relativement simplifié. Son exécution est amorcée suite à la réception d'un message. Ce dernier est envoyé par un client cherchant à acheter un billet d'avion pour un employé donné. Le message reçu inclut le nom de l'employé concerné, la destination, la date de départ ainsi que la date d'arrivée. Après cette réception, le processus cherche avant tout à vérifier le statut dudit employé. Pour ce faire, le processus fait appel au service Web « *Employee Travel Status* ». Suite à cette

vérification, le processus procède à la consultation du prix du billet d'avion auprès des deux compagnies aériennes : *American Airlines* et *Delta Airlines*. Cette consultation se fait à travers deux services Web dont chacun est offert par une des deux compagnies, à savoir le « *American Airlines web service* » et le « *Delta Airlines web service* ». L'appel à ces deux services Web se fait d'une façon concurrente. En dernier lieu, le processus choisit le prix le moins cher entre ceux fournis par les deux compagnies pour finalement retourner le plan de voyage au client.

Ci-dessous, une représentation graphique du processus *Travel* est fournie. Cette représentation correspond à un diagramme d'activité UML où des stéréotypes sont utilisés pour indiquer les opérations BPEL définies par ledit processus. Le code BPEL de ce dernier est fourni à la section D.1 de l'annexe D.

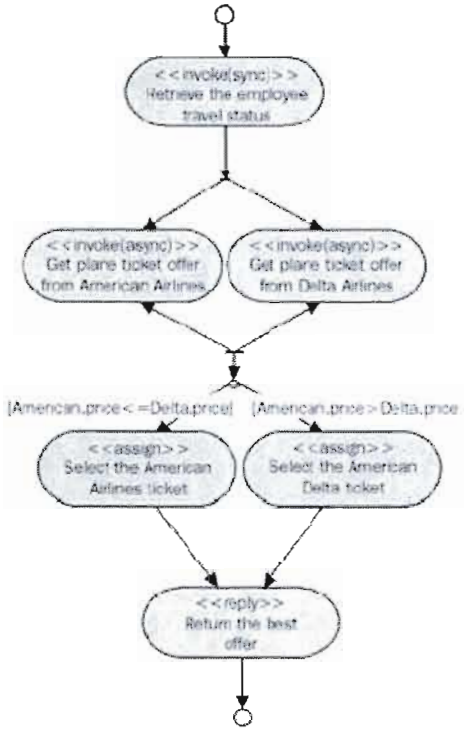


Figure 6.1 La représentation graphique du processus *Travel* (19).

7.2.2 Traduction et propriétés

Le processus *Travel* a été traduit en un graphe **B**-CFG en respect des patrons et règles définies aux chapitres précédents. Cette traduction est donnée par la figure 7.2.

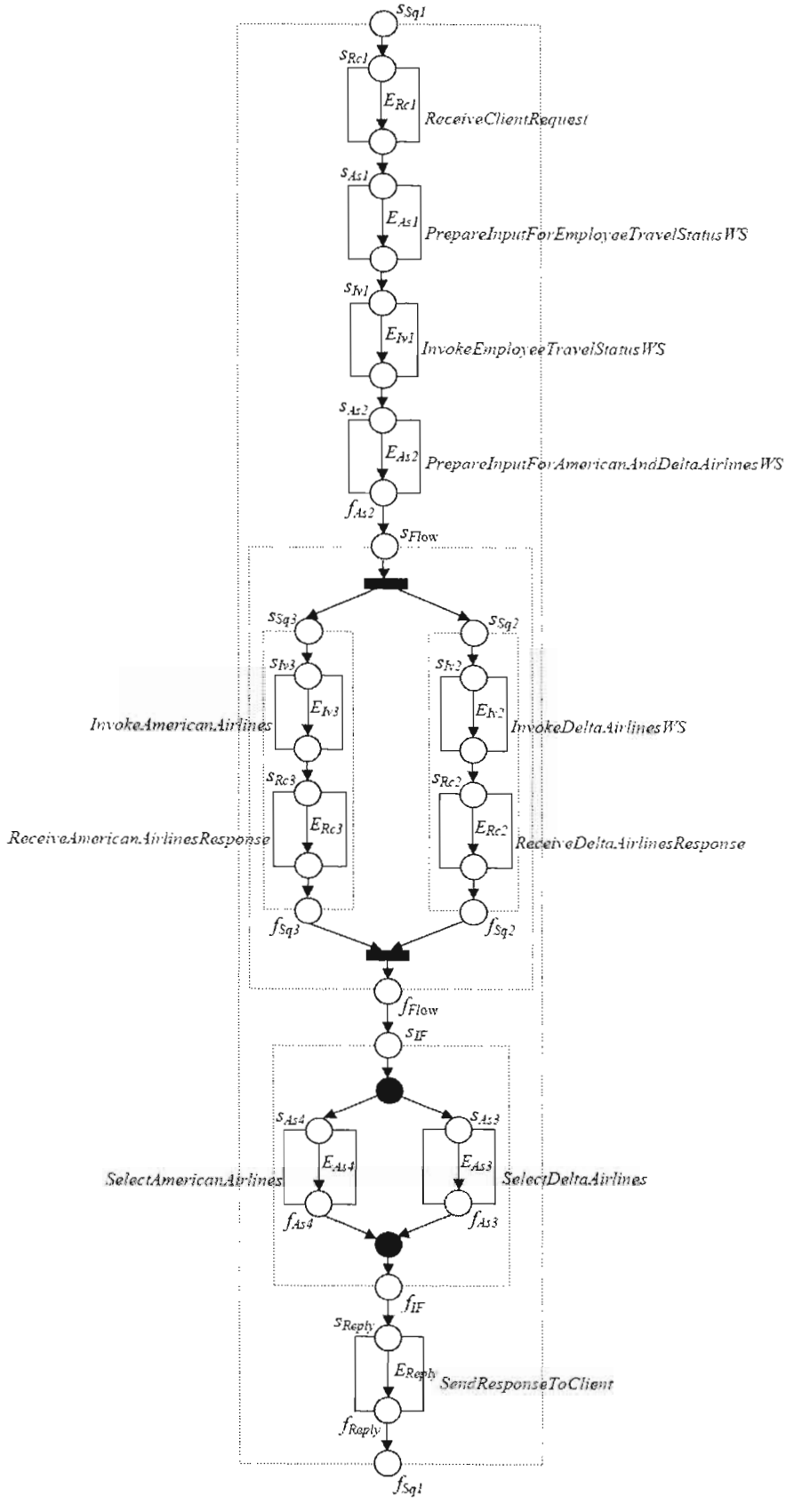


Figure 6.2 La traduction du processus *Travel* en B-CFG.

Quelques propriétés du processus *Travel* sont exposées en chiffres par le tableau 6.1. Ces propriétés concernent aussi bien la spécification du processus que sa traduction en **B**-CFG. Le tableau expose aussi quelques résultats qui ont été obtenus suite à l'application de nos algorithmes de génération de chemins de test sur cet exemple de processus BPEL. La génération a concerné les critères « tous-les-chemins », « toutes-les-branches » et « chemins-de-base ».

	Propriétés du processus		Propriétés du graphe		Chemins générés			
	Activités	Liens de contrôle	Nœuds	Arcs	Tous	Infaisables	Branches	Base
Nbr	16	0	36	36	2	0	2	2

Tableau 6.1 Les propriétés liées au processus *Travel*.

Les données présentées par le tableau 6.1 seront comparées à celles du tableau se rapportant à notre deuxième exemple de processus BPEL : le processus *loan Approval*.

6.3 Le processus *loan Approval*

Le deuxième exemple de processus BPEL que nous allons aborder est un processus dédié au traitement de demandes d'approbation de prêts. Ce processus est une variante de celui proposé par la norme BPEL (1), et dont le code est fourni à la section D.2.

7.3.1 Description

Le processus *loan Approval* voit son exécution amorcée suite à la réception d'une demande de prêt d'un de ses clients. Si le montant du prêt est supérieur à 10000, on fait appel au service Web *Approver* qui permet d'approuver ou non le prêt demandé. En revanche, si le montant est inférieur à 10000, le service Web *Assessor* est sollicité. Ce service permet d'évaluer le risque que présente le prêt demandé. Si ce risque est élevé, un appel au service *Approver* est requis pour l'approbation ou non du prêt. Cependant, si ce dernier présente un risque jugé faible, le prêt est directement accordé. Dans tous les cas, une réponse, négative ou positive, est envoyée au client demandeur. La représentation graphique du processus a été fournie à la section 1.4.5.

7.3.2 Traduction et propriétés

Le processus *loan Approval* a été à son tour traduit en un graphe **B**-CFG en respect des règles et patrons définis précédemment. Cette traduction est fournie par la figure 6.3.

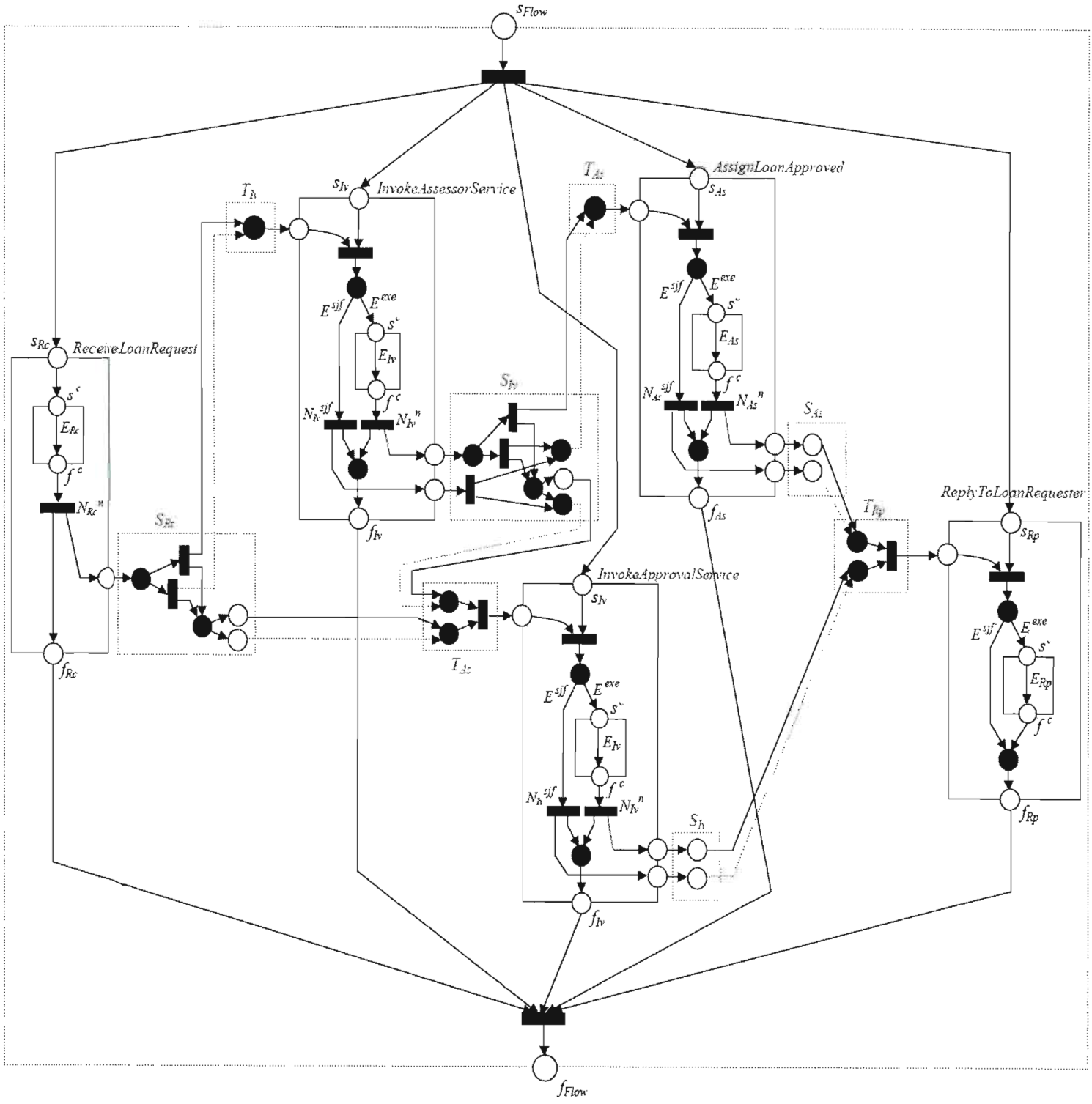


Figure 6.3 La traduction du processus *loan Approval* en graphe B-CFG.

Comme c'était le cas pour le processus *Travel*, quelques propriétés sur la spécification et la traduction du processus *loan Approval* sont exposées en chiffres par un tableau, en l'occurrence par le tableau 6.2. Ce dernier expose aussi quelques résultats qui ont été obtenus suite à l'application de nos algorithmes de génération de chemins de test sur ce deuxième exemple.

	Propriétés du processus		Propriétés du graphe		Chemins générés			
	Activités	Liens de contrôle	Nœuds	Arcs	Tous	Infaisables	Branches	Base
Nbr	6	6	80	101	160	157	3	3

Tableau 6.2 Les propriétés liées au processus *loan Approval*.

6.4 Comparaison

Après une comparaison des résultats qui sont dressés par les tableaux liés aux deux exemples *Travel* et *loan Approval*, nous avons pu tirer les constats suivants :

- Le nombre de nœuds et d'arcs du graphe **B**-CFG du processus *loan Approval* est presque trois fois plus important que de celui du graphe **B**-CFG du processus *Travel*, alors que l'on sait que ce dernier définit quand même plus d'activités BPEL que son congénère (16 pour *Travel* et 6 pour *loan Approval*). Cela pourrait s'expliquer par la présence de liens de contrôle dans l'un (6 liens pour *loan Approval*) et leur absence dans l'autre (aucun lien pour *Travel*). Au fait, nous pouvons dire que (potentiellement) de tels liens ajoutent une certaine complexité aux processus les définissant, et en particulier au résultat de traduction de ces derniers en **B**-CFG.
- Le même constat que nous avons évoqué pour le nombre de nœuds et d'arcs est valable aussi pour le nombre des chemins de test générés. Ce nombre de chemins, pour le processus *loan Approval*, est beaucoup plus élevé que celui ayant été trouvé pour le processus *Travel* (160 pour *loan Approval* et 3 pour *Travel*). Le même facteur de présence ou d'absence de liens de contrôle semble être aussi plausible pour expliquer ce deuxième fait.
- Les critères « toutes-les-branches » et « chemins-de-base » ont été satisfaits pour le processus *Travel*, alors qu'ils ne l'ont pas été pour le processus *loan Approval*. Deux chemins de test faisables et indépendants ont pu être générés pour l'exemple *Travel* dont le graphe **B**-CFG ne définit que deux branches et a un nombre concurrent de McCabe valant 2. Cependant, pour le *loan Approval*, dont le graphe **B**-CFG a un nombre VC = 9, on ne

peut générer plus que 3 chemins de test faisablcs qui restent insuffisants pour satisfaire le critère « chemins-de-base » ou même le critère « toutes-les-branches ».

Par ailleurs, nous constatons que le nombre des chemins faisables qui ont été générés pour l'exemple *loan Approval* est faible par rapport au nombre de leurs rivaux infaisables (3 chemins faisables contre 157 infaisables). Cela pourrait être dû à la nature même du processus et à la façon dont il a été défini : un processus dont les branches ne sont que des choix exclusifs (un montant soit inférieur soit supérieur à 10000, et un risque soit faible soit élevé) et où il n'y a pas besoin d'un flot concurrent aurait dû être défini à base d'activités conditionnelles comme le `<if>` au lieu d'être défini à base d'un `<flow>` enrichi par des liens de contrôle.

Les chemins infaisables qui ont dû être générés pour l'exemple *loan Approval* ont été presque tous détectés par le deuxième algorithme d'élimination des chemins infaisables (153 des chemins infaisables détectés sont du deuxième type, seuls 4 sont du troisième type et aucun du premier). Ce constat montre l'utilité de ce deuxième algorithme qui peut rendre cette détection moins coûteuse en termes de temps d'exécution. Au fait, si nous n'avons pas pensé à un tel algorithme, nous aurions été contraints (pour pouvoir détecter ces 153 chemins infaisables du deuxième type) de faire appel à la méthode proposée pour la détection des chemins infaisables du troisième type. Ladite méthode est basée sur un solveur de contraintes qui pourrait compromettre les performances en termes de temps d'exécution, voire mener à un problème d'indécidabilité (à propos de la faisabilité des chemins).

Finalement, nous soulignons que le nombre des chemins infaisables, pour l'exemple *loan Approval*, aurait pu être réduit si la détection de tels chemins se faisait à la volée et en exploitant l'information de faisabilité des chemins déjà générés. Autrement dit, si lors de la génération d'un chemin donné on a su qu'il est devenu infaisable à un certain nœud X, la génération de ce chemin devrait être interrompue, de même qu'il n'y aura plus besoin d'explorer les autres éventuelles branches dérivant de ce nœud X. La prise en compte d'une telle détection est envisagée dans le cadre de futurs travaux.

6.5 Conclusion

Dans ce cinquième et dernier chapitre, nous avons exposé les résultats que nous avons eus suite à l'application de notre méthode de génération de cas de test sur deux exemples de processus BPEL de natures différentes, et cela tout en décortiquant le pourquoi de chaque

résultat obtenu. Pour clore ce mémoire, nous dresserons dans ce qui suit un bilan de notre travail et une liste de ce qui reste à faire pour l'améliorer et le compléter.

CONCLUSION

En guise de conclusion, nous présenterons une récapitulation des principales contributions de ce mémoire, de même que nous exposerons les pistes qui restent à explorer comme perspectives de travail dans cette arène du test unitaire des processus BPEL, et en particulier dans le cadre de son volet « génération des cas de test ».

Contributions

Pour notre génération des cas de test pour tout processus BPEL, nous avons proposé une variante de l'approche orientée chemin. Cette variante s'effectue en plusieurs étapes. Elle exige à ce que le code BPEL à tester soit en premier lieu traduit en un graphe de flot de contrôle (CFG). À partir de ce dernier des chemins et données de test sont extraits, selon des critères bien établis, pour finalement donner naissances à des cas de test exécutables. À chaque étape de ladite variante, nous avons pu proposer des solutions qui tiennent compte de la nature assez particulière et assez complexe du langage BPEL. Ces solutions consistent en :

- Une traduction du code BPEL en une version augmentée du graphe de flot de contrôle, baptisée **B**-CFG. Cette version a été définie comme une collection de patrons et de règles qui rendent la traduction du BPEL en CFG quasi-systématique. Formellement définie, la nouvelle version reproduit fidèlement les particularités comportementales de tout processus BPEL à tester, et cela que ce dernier inclut ou pas des comportements concurrentiels, aussi complexes soient-ils. Par ailleurs, la traduction que nous avons envisagée concerne aussi bien le comportement normal d'un processus BPEL qu'une partie de son comportement exceptionnel (gestion d'événements et d'erreurs).

- Une nouvelle représentation des chemins de test à générer qui a été élaborée d'une façon formelle à base de la notion de relation d'ordre. Cette nouvelle représentation sied aux deux types de chemins concurrents et séquentiels.

- Des algorithmes pour la génération de chemins de test faisables ainsi que pour l'élimination de leurs congénères infaisables.

- Une prise en compte de plusieurs types de couvertures lors de cette phase de génération de chemins, à savoir : la couverture « tous-les-chemins » avec les critères « 0-1 répétition » et « 0-2 instances », la couverture « tous-les-chemins-indépendants » de même que la couverture « toutes-les-branches ».

- Une re-définition du calcul du nombre de McCabe de façon à le rendre sensé et pouvoir l'utiliser dans le processus du test logiciel appliqué à des langages concurrents comme le langage BPEL.

- Des propositions pour la génération de données de test ainsi que pour la spécification et l'exécution des cas de test.

Au cours de la réalisation du présent travail, quelques importantes décisions de conception ont été prises. Parmi ces dernières, nous citons la décision visant à réduire la taille des graphes **B**-CFG en minimisant la taille du code mort représenté. Cette minimisation est basée sur l'algorithme de sélection que nous proposons. Un autre exemple de décisions de conception ayant été prises correspond à celui du choix d'une représentation **B**-CFG qui est riche en information graphique. Cette richesse en ce type d'information nous évite (au maximum) d'avoir à analyser les données présentes dans le graphe et facilite ainsi le parcours de ce dernier pour une génération plus simple des chemins de test.

Finalement, il faut noter que la sémantique du **B**-CFG que nous proposons ne permet de représenter qu'une seule instance d'un processus BPEL à tester. De plus, l'aspect *temporel* dans un processus BPEL (e.g., `<wait>`) ne peut être représenté dans un graphe **B**-CFG — graphe qui ne peut être qu'une représentation *statique* de ce dernier.

Travaux futurs

Dans la perspective de faire suite au travail réalisé, nous envisageons dans le cadre de futurs travaux de traiter certaines parties inachevées ainsi que de donner suite à d'autres pistes qui restent à explorer. Entre autres, nous envisageons de :

- Compléter la traduction du BPEL en **B**-CFG pour d'une part, prendre compte des éventuelles transmissions du traitement d'erreurs d'un `<scope>` à un autre, et d'autre part, englober les `<compensationHandlers>` et les `<terminationHandlers>` afin de tenir compte du processus de gestion d'erreurs dans sa globalité.

- Développer un outil pour une traduction automatique du code BPEL en **B**-CFG. Un tel outil nous permettra d'éviter une traduction manuelle pouvant s'avérer laborieuse, surtout pour le cas de processus BPEL de grande taille.
- Appliquer notre traduction et nos algorithmes sur d'avantage d'exemples de processus BPEL, en particulier sur des processus de grande taille qui soient des cas d'application émanant de l'industrie. Cela nous aidera à mieux évaluer notre solution et à tirer des conclusions qui nous permettront, éventuellement, d'ouvrir les yeux sur ce qui serait à améliorer.
- Mieux élaborer la solution que nous proposons pour la phase de génération de données de test. Cette élaboration devrait être faite de façon à pouvoir mettre en oeuvre ladite solution et à prendre en considération le problème des variables partagées.
- Développer un module qui permet de convertir automatiquement les chemins et données de test générés (i.e., les cas de test abstraits) en des cas de test exécutables regroupés en des suites de test. Ces cas de test exécutables devraient être générés dans un format qui soit similaire à celui exigé par le *framework* d'exécution de test utilisé.

APPENDICE A

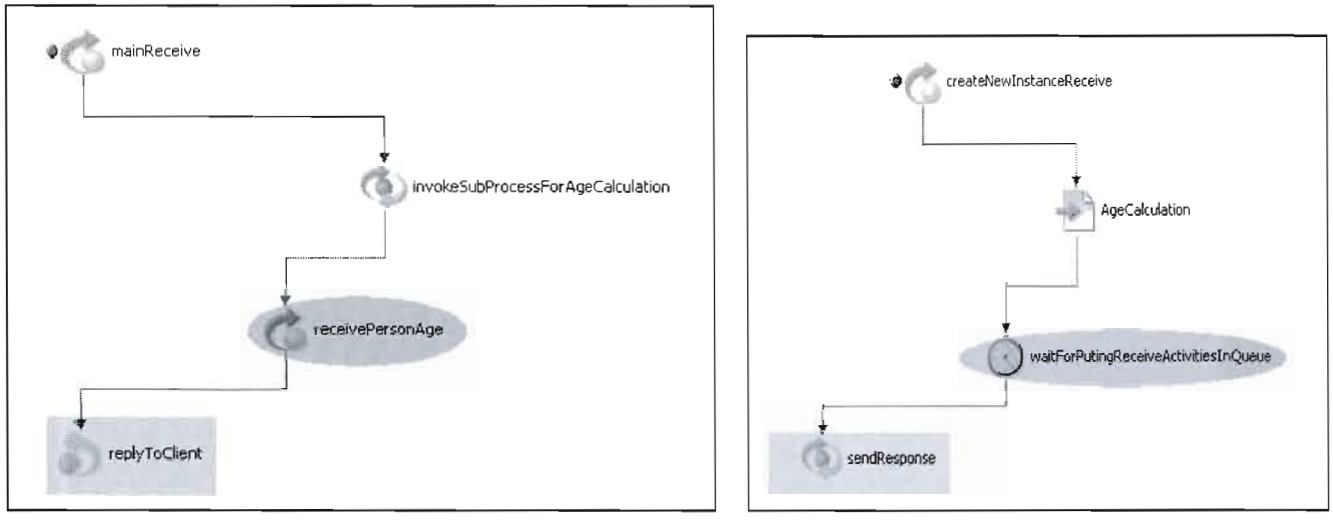
LES CHOIX D'IMPLÉMENTATION D'ACTIVEBPEL ENGINE

A.1 Premier point et premier choix

La gestion des files d'attente des messages en BPEL peut différer d'une implémentation à une autre. En particulier, dans le cas où l'on a affaire à deux instances d'un même processus métier, comment une implémentation donnée (*activeBPEL Engine* dans notre cas) fait en sorte que les messages envoyés par un processus partenaire aboutissent à la bonne instance étant à leur attente. Certes, cela est rendu possible par le biais des `<correlationSets>`, mais ce qui serait intéressant à vérifier c'est le comportement des files d'attente vis-à-vis d'une situation où l'on a des messages envoyés, par deux instances différentes, sur le même *partnerLink*, le même port, la même opération et en utilisant la même *correlationSet* portant les mêmes valeurs?

Les files d'attente utilisées au niveau d'*activeBpel Engine* sont au nombre de deux : une première file permet de mémoriser les activités `<receive>` qui sont déclenchées et en attente d'un message, et une deuxième utilisée pour la mémorisation des messages qui sont reçus mais en attente du déclenchement de leurs activités `<receive>` respectives. Ainsi, pour répondre à la question posée ci-dessus, nous avons pensé à deux scénarios que nous avons implémentés au niveau d'*activeBPEL Engine*. Le premier a été implémenté dans l'optique d'examiner le comportement de la file d'attente utilisée pour les activités `<receive>`, tandis que le deuxième a été plutôt envisagé pour le test de celle utilisée pour les messages reçus. Les figures A.1 et A.2 exposent les représentations graphiques des deux scénarios.

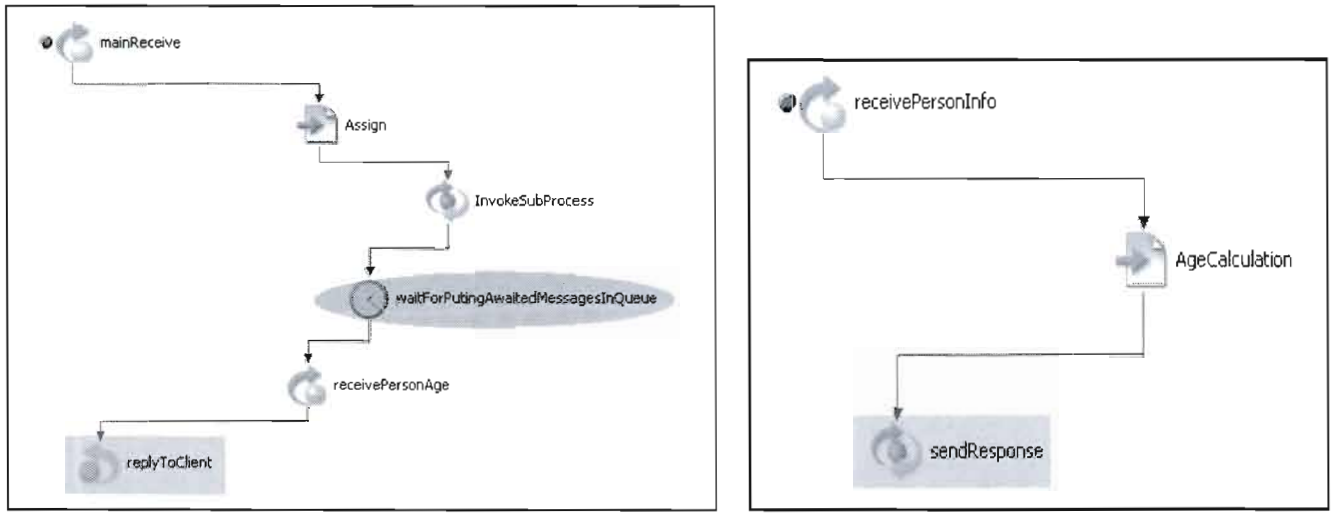
En ce qui concerne les activités `<receive>`, nous avons constaté et conclu, suite à l'exécution du premier scénario (figure A.1), que le serveur activeBPEL Engine utilise une file ***LIFO*** pour leur mémorisation.



Le processus appelant Le processus appelé

Figure A.1 Le scénario adopté pour le test de la file d'attente des activités `<receive>`.

Quant aux messages reçus, nous avons constaté et conclu, suite à l'exécution du deuxième scénario (figure A.2), que l'activeBPEL Engine utilise une file ***FIFO*** pour leur mémorisation.



Le processus appelant Le processus appelé

Figure A.2 Le scénario adopté pour le test de la file d'attente des messages reçus.

A.2 Deuxième point et deuxième choix

Dans le cas où une instance d'un processus donné reçoit un message attendu par une activité `<receive>` qui n'a pas été encore exécutée par cette instance, est ce que le message sera mémorisé, rejeté ou une erreur sera déclenchée?

Pour répondre à cette question, nous avons pensé à un scénario que nous avons implémenté au niveau d'*activeBPEL Engine* (figure A.3) et qui nous a permis de tirer les constats suivants : dans le cas où les échanges de messages sont corrélés, les messages reçus sont mémorisés et l'exécution se termine normalement, alors que si ces échanges ne sont pas corrélés, une erreur du type *correlationViolation* pourrait avoir lieu.

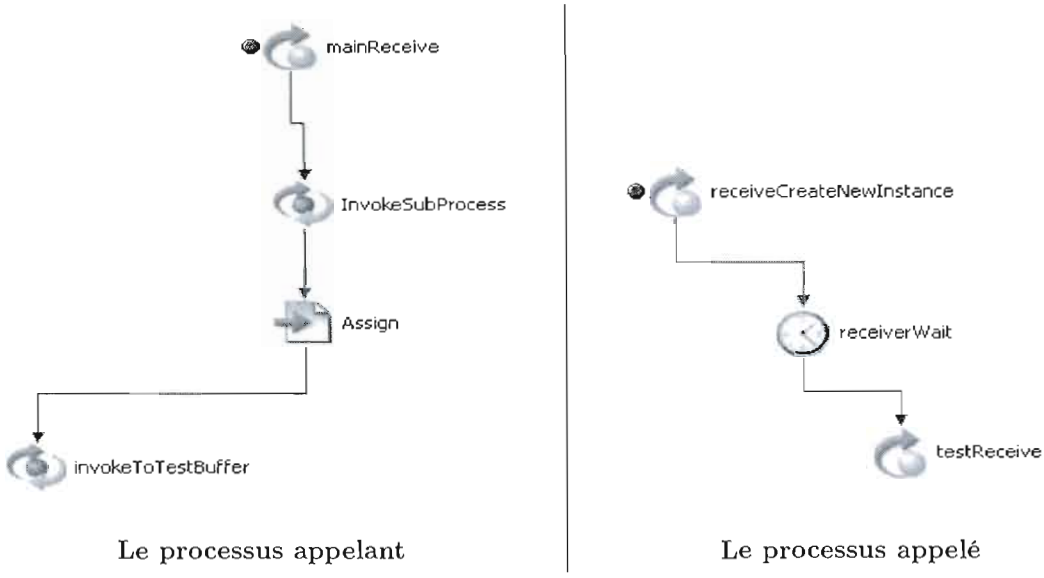


Figure A.3 Le scénario adopté pour le test des messages échangés avec ou sans corrélation.

APPENDICE B

LES PATRONS DE TRADUCTION CONNEXES

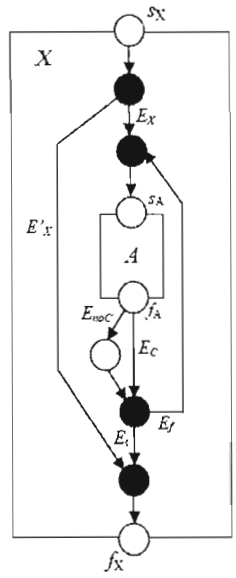
B.1 L'activité <forEach> et l'élément <completionCondition>

Dans le cas où la déclaration d'un `<forEach>` X inclut l'élément `<completionCondition>`, cela sous-entend la définition d'une condition de type "*at least m out of n*" qui s'applique sur le nombre des instances exécutées du *scope* A attaché à X, avec $n = (Fcv\text{-}exp - Scv\text{-}exp) + 1$ et m correspond à la valeur d'évaluation de l'expression définie par l'élément `<branches>` (e.g., l'expression *Brc-exp* de la figure B.1). Autrement dit, si, lors de la performance du `<forEach>` X, l'exécution d'au moins m instances du `<scope>` A est accomplie, sa `<completionCondition>` tiendra, et dans ce cas aucune autre instance ne sera créée s'il s'agit d'un `<forEach>` séquentiel, tandis que pour le cas d'un `<forEach>` parallèle, toutes les instances qui sont en cours d'exécution verront leur exécution interrompue. La comparaison du nombre des instances exécutées du scope A avec le nombre m se fait à la fin d'exécution de chaque instance dudit `<scope>`.

En fait, la valeur du nombre des instances exécutées qui est comparé avec m dépend de l'attribut optionel `<successfulBranchesOnly>` de l'élément `<branches>`. Si cet attribut est mis à "no", toutes les instances exécutées du scope A seront comptées sans avoir à vérifier si ces instances se sont exécutées avec succès ou pas. Cependant, si l'atribut `<successfulBranchesOnly>` est à "yes", on ne compte que les instances du scope A qui se sont exécutées avec succès.

La figure B.1 présente le patron de traduction d'un `<forEach>` séquentiel qui est muni d'un élément `<completionCondition>` définissant un attribut *successfulBranchesOnly* à "yes". Tel que mentionné ci-dessus, pour un pareil `<forEach>`, il ne faut considérer que les itérations où l'exécution du `<scope>` A a réussi. Autrement dit, l'incrémentation du compteur C_X ne sera plus systématique comme c'était le cas pour le patron de la figure 4.6(a), mais dépendra de

l'état d'exécution du <scope> A.




```
<forEach counterName="CX"
    parallel="no" name="X">
  <startCounterValue Sav-exp />
  <finalCounterValue Fav-exp />
  <completionCondition>
    <branches successfulBranchesOnly = "yes">
      Brc-exp
    </branches>
  </completionCondition>
  <scope name="A">
    … …
  </scope>
</forEach>
```

Figure B.1 L'activité <forEach> et son élément <completionCondition>.

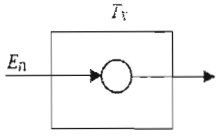
Au fait, si pour une itération donnée, l'exécution du <scope> A réussit, i.e. la variable ss associée à ce scope était mise à *true* (voir 4.6.5), dans ce cas le compteur C_X sera incrémenté via l'arc E_C dont $ac = (C_X{+}{+})$ et $pr = (ss == true)$. Dans le cas contraire, i.e. une exécution erronée du <scope> A et donc une variable ss étant mise à *false*, le compteur C_X ne sera pas incrémenté, mais c'est un autre compteur C'_X qui le sera via l'arc E'_C dont $ac = (C'_X{+}{+})$ et $pr = (ss == false)$; le compteur C'_X est une variable qui sera, comme l'est la variable C_X, ajoutée à l'ensemble des variables globales $\boldsymbol{V}$ dans le but de garder trace sur le nombre de fois où l'exécution du <scope> A a échoué. Finalement, pour l'arc E_t, son prédicat $pr = (C_X > m$ *Ou* $(C_X + C'_X) > n)$, alors que pour l'arc E_f, son prédicat $pr = (C_X \leq m$ *Et* $(C_X + C'_X) \leq n)$[1].

B.2 Les conteneurs <targets> et <sources>

Ci-dessous sont fournis des patrons de traduction correspondant à des cas où le conteneur <targets> n'accueillera qu'un seul lien de contrôle.

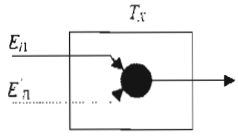
[1]Pour le cas où l'attribut *successfulBranchesOnly* serait mis à "no", c'est le même patron dépeint par la figure 4.6(a) qui sera adopté (car on n'aura pas à vérifier l'état d'exécution du scope enchâssé), et le seul changement qu'il va falloir noter concernera la définition des deux arcs E_t et E_f : pour le premier, son prédicat pr vaudra $(C_X > m)$, tandis que pour le deuxième, son pr vaudra $(C_X \leq m)$.

```
<activity name = "A"
 suppressJoinFailure = "no">
   <sources>
    <source l1 />
  </sources>
   … …
</activity>
…. ….
<activity name = "X">
   <targets>
    <joinCondition>
      bool-exp
    <joinCondition>
    <target l1 />
  </targets>
   … …
</activity>
```



Conteneur <targets> simple sans transitionCondition et dont l'activité source n'est pas à sauter et/ou a un sjf = "no"

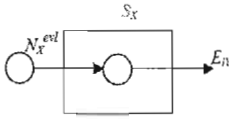
```
<activity name = "A"
    suppressJoinFailure = "yes">
   <sources>
    <source l1>
     <transitionCondition C1/>
    </source>
  </sources>
   … …
</activity>
…. …
<activity name = "X">
   <targets>
    <joinCondition>
      bool-exp
    <joinCondition>
    <target l1 />
  </targets>
   … …
</activity>
```



Conteneur <targets> simple avec transitionCondition ou dont l'activité source est à sauter et/ou a un sjf = "yes"

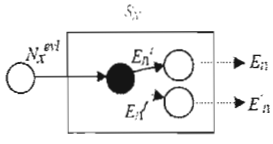
Ci-dessous sont fournis des patrons de traduction correspondant à des cas où le conteneur <sources> n'enveloppera qu'un seul lien de contrôle.

```
<activity name = "X"
    suppressJoinFailure = "no">
   <sources>
    <source l1 />
  </sources>
   … …
</activity>
```



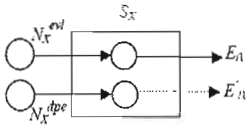
Conteneur <sources> simple avec un sjf = "no" et sans transitionCondition

```
<activity name = "X"
    suppressJoinFailure="no">
   <sources>
    <source l1>
     <transitionCondition C1/>
    </source>
  </sources>
   . .
</activity>
```



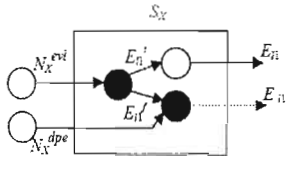
Conteneur <sources> simple avec transitionCondition et un sjf = "no"

```
<activity name = "X"

suppressJoinFailure="yes">
   <sources>
    <source l1 />
  </sources>
   … …
</activity>
```



Conteneur <sources> simple, sans transitionCondition, d'une activité X à sauter et/ou dont sjf = "yes" avec des links entrants

```
<activity name = "X"
    suppressJoinFailure="yes">
   <sources>
    <source l1>
     <transitionCondition C1/>
    </source>
  </sources>

</activity>
```



Conteneur <sources> simple, avec transitionCondition, d'une activité X à sauter ou dont sjf = "yes" avec des links entrants

APPENDICE C

UN DEUXIÈME ALGORITHME POUR LA GÉNÉRATION DES CHEMINS

C.1 Description

Pour qu'il arrive à ne visiter qu'une seule fois les arcs d'un graphe **B**-CFG à parcourir, ce deuxième algorithme adopte une approche de parcours qui est différente de celle du premier (donné à la section 5.2.2). Ladite approche se distingue, d'une part, par sa façon de traiter les sous-chemins parallèles découlant d'un même noeud de bifurcation, et d'autre part, par l'optimisation du traitement des chemins se croisant à un même noeud de fusion. Dans la suite, nous expliquons par l'exemple les deux traitements ainsi énumérés, ce qui revient donc à exposer l'approche de ce deuxième algorithme.

C.1.1 Traitement des chemins se croisant à un même noeud de fusion

Les chemins qui sont en cours de génération et qui se croisent à un même noeud de fusion X sont traités par le deuxième algorithme de façon à ce que la suite d'arcs (ou éventuellemet les suites d'arcs) découlant de ce noeud X ne soit pas explorée plusieurs fois pour venir compléter ces chemins — i.e. une seule exploration de cette suite d'arcs suffit pour venir compléter tous les chemins se croisant à ce même noeud de fusion X. Comme exemple à cela, si nous avions à utiliser le deuxième algorithme pour générer les chemins C_2 et C_3, énumérés au paragraphe 5.2.2.3 et se croisant au noeud de fusion "9" du graphe dépeint par la figure 5.1, la suite d'arc $[(9,10),(10,11)]$ qui découle donc de ce noeud "9" ne devrait être explorée qu'une seule fois pour venir compléter les deux chemins C_2 et C_3 lors de leur génération.

Au fait, durant la génération d'un chemin C, à chaque fois que ce deuxième algorithme atteint un noeud de fusion X, on teste si ce noeud reste non exploré. Si c'est le cas, ce noeud

est joint au chemin C et l'exploration d'une suite d'arcs qui découle de ce noeud se poursuit normalement. En revanche, si l'on trouve que le noeud X a été déjà exploré, cela signifie que, arriver à ce stade, au moins un (sous)chemin C' a été généré et a déjà fait adhérer à son sein le noeud X et une suite d'arcs qui en découle. Dans ce cas, cette dernière suite d'arcs est ajoutée au chemin C qui est en cours de génération et cela sans que l'on soit amené à explorer de nouveau les arcs qui constituent cette suite. Éventuellement, cette même suite d'arcs serait aussi ajoutée aux autres chemins qui croiseraient le chemin C' à ce noeud de fusion X.

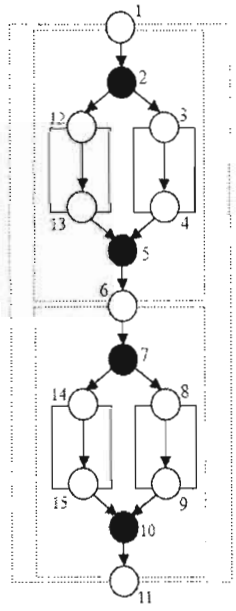


Figure C.1 Un graphe **B**-CFG où les chemins se croisent à différents noeuds de fusion.

Comme exemple à un tel traitement, nous appliquons le deuxième algorithme sur le graphe **B**-CFG dépeint à la figure C.1. Étant donné que ce deuxième algorithme sera, à l'image du premier, basé sur un parcours en profondeur d'abord, on commence par générer un premier chemin allant du noeud d'entrée "1" au noeud de sortie "11". Supposons que ce chemin correspondrait au chemin C_1 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11)]. L'algorithme considère par la suite un deuxième chemin C_2 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,14), (14,15), (15,10)] qui est à compléter[1]. Vu que ce chemin croise son congénère C_1 au noeud de

[1] Le chemin C_2 diverge du premier chemin C_1 au noeud de décision le plus récemment découvert lors de la génération de ce dernier, à savoir : le noeud "7" Au fait, à l'image du premier algorithme, à chaque fois que ce deuxième algorithme atteint un noeud de décision X, on duplique le chemin courant en n chemins, où n est le nombre des noeuds adjacents au noeud X.

fusion "10" étant déjà visité et joint à ce dernier, la suite [(10,11)] à arc unique qui découle de ce noeud ne sera donc pas explorée de nouveau pour venir compléter le chemin C_2. En effet, cette suite sera copiée du chemin C_1 pour être ajoutée au chemin C_2 de sorte à le compléter.

De même, le chemin $C = [(1,2),(2,12),(12,13),(13,5)]$, qui croise et le chemin C_1 et le chemin C_2 au noeud de fusion "5", résultera en deux chemins C_3 et C_4 qui seront respectivement complétés par les suites d'arcs [(5,6), (6,7), (7,8), (8,9), (9,10), (10,11)] et [(5,6), (6,7), (7,14), (14,15), (15,10), (10,11)]. Ces deux suites sont respectivement copiées des chemins C_1 et C_2 et non pas générées en explorant de nouveau les arcs les constituants. Ainsi, le parcours du graphe par ce deuxième algorithme donnera lieu à quatre chemins sans pour autant être amené à explorer une partie de ses arcs une multitude de fois. Les quatre chemins résultants sont :

- C_1 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11)];
- C_2 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,14), (14,15), (15,10), (10,11)];
- C_3 = [(1,2), (2,12), (12,13), (13,5), (5,6), (6,7), (7,8), (8,9), (9,10), (10,11)];
- C_4 = [(1,2), (2,12), (12,13), (13,5), (5,6), (6,7), (7,14), (14,15), (15,10), (10,11)].

C.1.2 Traitement des sous-chemins parallèles

Dans un autre registre, les sous-chemins parallèles découlant des noeuds de bifurcation sont traités de façon à ce qu'ils soient tous générés une seule fois pour résulter en des chemins complets qui seront le fruit de leur combinaison. Pour concrétiser cela, nous considérons que chaque noeud de bifurcation définit plusieurs branches parallèles. Le nombre de ces branches parallèles est égal au nombre des noeuds adjacents au noeud de bifurcation considéré. Potentiellement, chacune de ces branches pourrait envelopper un ou plusieurs sous-chemins, e.g. le noeud "2" du graphe B-CFG de la figure C.2 a deux noeuds adjacents, "3" et "13", marquant la naissance de deux branches parallèles dont chacune enveloppe deux sous-chemins. Les sous-chemins d'une branche, d'un noeud de bifurcation X, sont déstinés à être combinés avec leurs congénères qui leur sont parallèles et qui sont enveloppés par les autres branches du même noeud de bifurcation X.

Pour illustrer notre propos, nous appliquons le deuxième algorithme, qui adopte le traitement des sous-chemins parallèles décrit ci-dessus, sur le graphe de la figure C.2. Comme résultat de son fondement sur un parcours *DFS*, cet algorithme commence par générer un premier chemin allant du noeud d'entrée "1" au noeud de sortie "10". Supposons que ce chemin correspondrait au chemin incomplet C_{11} = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10)]. Ce dernier,

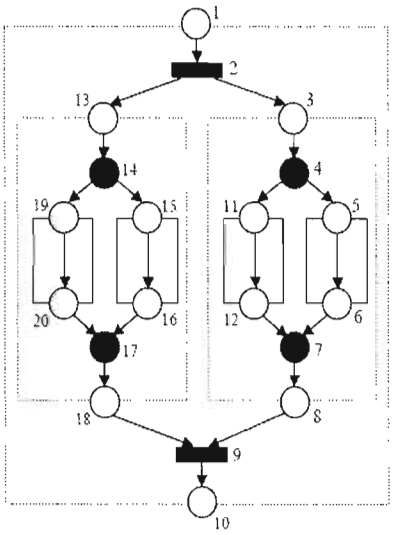


Figure C.2 Un graphe B-CFG où des sous-chemins parallèles sont définis.

et en particulier sa sous-composante [(2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9)], est considéré comme le premier sous-chemin extrait d'une première branche découlant du noeud de bifurcation "2". Par la suite, l'algorithme extrait le deuxième sous-chemin C_{12} = [(1,2), (2,3), (3,4), (4,11), (11,12), (12,7), (7,8), (8,9), (9,10)] enveloppé par cette même première branche[2].

Ayant extrait l'ensembe des sous-chemins enveloppés par la première branche du noeud de bifurcation "2", l'algorithme passe à la deuxième branche de ce noeud pour extraire les deux sous-chemins C_{21} = [(2,13), (13,14), (14,15), (15,16), (16,17), (17,18), (18,9)] et C_{22} = [(2,13), (13,14), (14,19), (19,20), (20,17), (17,18), (18,9)] qui y sont enveloppés. Finalement, les sous-chemins C_{11} et C_{12} de la première branche sont combinés avec ceux de la deuxième (i.e. C_{21} et C_{22}) pour ainsi résulter aux quatre chemins de test donnés comme suit :

- C_1 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (2,13), (13,14), (14,15), (15,16), (16,17), (17,18), (18,9)], i.e. combinaison des sous-chemins C_{11} et C_{21}.
- C_1 = [(1,2), (2,3), (3,4), (4,5), (5,6), (6,7), (7,8), (8,9), (9,10), (2,13), (13,14), (14,19), (19,20), (20,17), (17,18), (18,9)], i.e. combinaison des sous-chemins C_{11} et C_{22}.
- C_{12} = [(1,2), (2,3), (3,4), (4,11), (11,12), (12,7), (7,8), (8,9), (9,10), (2,13), (13,14), (14,15), (15,16), (16,17), (17,18), (18,9)], i.e. combinaison des sous-chemins C_{12} et C_{21}.

[2]Conformément à ce qui est souligné plus haut, la suite d'arc [(7,8), (8,9), (9,10)] n'est exploré qu'une seule fois pour venir compléter les deux sous-chemins C_{11} et C_{12}, se croisant au noeud de fusion "7"

- C_{12} = [(1,2), (2,3), (3,4), (4,11), (11,12), (12,7), (7,8), (8,9), (9,10), (2,13), (13,14), (14,19), (19,20), (20,17), (17,18), (18,9)], i.e. combinaison des sous-chemins C_{12} et C_{22}.

Ainsi, et par sa façon de traiter les sous-chemins parallèles découlant d'un même noeud de bifurcation et par son optimisation du traitement des chemins se croisant à un même noeud de fusion, le deuxième algorithme arrive à n'explorer les arcs d'un graphe **B**-CFG qu'une seule fois. Pour ce faire, ce deuxième algorithme, à l'image du premier, distingue les noeuds de décision de ceux de bifurcation, et surpasse même ce dernier en distinguant aussi les noeuds de fusion et de jointure des noeuds normaux. Ainsi, le deuxième algorithme tire profit de la sémantique de tous les types de noeuds fournis par la syntaxe des graphes **B**-CFG, i.e. tire profit d'une bonne partie de la richesse de l'inforamtion graphique présentée par ces derniers. Le pseudo code implémentant ce deuxième algorithme est donné par le paragraphe suivant.

C.2 Pseudo-code

Le pseudo-code fourni ci-dessous implémente le deuxième algorithme de génération des chemins de test selon une forme récursive, et cela contrairement au premier algorithme qui est implémenté selon une forme itérative. Nous rappelons aussi que ce deuxième algorithme admet comme *entrée* le graphe **B**-CFG à parcourir — un graphe qui est représenté en listes d'adjacence des noeuds le composant. À chacun de ces noeuds, on associe la propriété « *visited* » qui, prenant pour valeur *true* ou *false*, permet d'indiquer si, lors du parcours du graphe, le noeud en question a été déjà visité ou pas. Cependant, comme *sortie*, l'exécution de ce deuxième algorithme résulte en une liste des chemins de test à générer (i.e. *LCT*). Outre cette liste, l'algorithme fait usage d'une deuxième liste et de deux piles, à savoir :

- La *LCI* : *L*iste des *C*hemins *I*ncomplets où l'on sauvegarde les (sous)chemins qui sont liés à un noeud de bifurcation donné et qui sont en cours de construction.
- La *PCB* : *P*ile des *C*hemins d'une *B*ranche où l'on sauvegarde les sous-chemins enveloppés par une *branche* d'un noeud de bifurcation donné.
- La *PNBD* : *P*ile des *N*oeuds de *B*ifurcation *D*écouverts où l'on sauvegarde les noeuds de bifurcation dans l'ordre selon lequel ils sont découverts lors du parcours du graphe. Chacun de ces nœuds sera retiré de cette pile une fois toutes ses branches sont explorées.

Au fait, à chaque (sous)chemin, faisant partie de la liste *LCI* ou de la pile *PCB*, on associe les proporiétés « *node* » et « *branch* ». La première (i.e. *node*) permet, éventuellement,

d'indiquer à quel noeud de bifurcation le (sous)chemin est lié, alors que la deuxième (i.e. *branch*) est plus spécifique car elle permet, éventuellement, d'indiquer de quelle branche d'un noeud de bifurcation le (sous)chemin fait partie. En revanche, à chaque noeud X de la pile *PNBD*, on associe les deux propriétés « *branchNbr* » et « *currentBranch* ». La première (i.e. *branchNbr*) désigne le nombre de branches définies par un noeud X de ladite pile, alors que la deuxième (i.e. *currentBranch*) désigne la branche de ce noeud qui est en cours de traitement.

Le pseudo-code implémentant le deuxième algorithme fait appel à des méthodes qu'on décrit dans le tableau C.1.

Méthode	Description
currentBranchToNext()	Permet de passer à la branche suivante du noeud de bifurcation considéré en incrémentant d'un pas sa propriété *currentBranch*.
createPathCopy()	Crée une copie du chemin donné en paramètre et la retourne.
newPath()	Crée un chemin vierge et le retourne.
nextNode()	Retourne le noeud adjacent correspondant.
adjNodeNbr()	Retourne le nombre des noeuds adjacents au noeud donné en paramètre.
push()	Empile un élément dans la pile PCB ou PNBD.
pop()	Dépile un élément de la pile PCB ou PNBD.
add()	Si le paramètre fourni est un couple (u, v), la méthode l'ajoute au chemin concerné. Cependant, si le paramètre correspond à un chemin, la méthode l'ajoute à la liste LCT ou LCI.
get()	Retourne le dernier élément étant empilé dans la PNBD et cela sans pour autant le dépiler de cette dernière.
delete()	Supprime de la liste LCI le chemin donné en paramètre.
concat(p1,p2)	Concatène les chemins *p1* et *p2* pour former un seul chemin qui constitue sa valeur de retour.
copyRest(p,n)	Crée une copie du *reste* du chemin p — reste qui correspond à la suite des couples (u,v) faisant partie de p et qui, commençant par le couple (u,v) dont u = n, se termine par le dernier couple de p.

Tableau C.1 Les méthodes utilisées par le deuxième algorithme de génération des chemins.

```
      // ****************** ALGORITHME PRINCIPAL ******************/
BEGIN
   LCT = Null
   LCI = Null
   PCB = Null
   PNBD = Null
   initialPath = newPath()
   Generate-Path(s, initialPath) // 's' est la racine du graphe
END
      // ***************** PROCÉDURE Generate-Path() *****************/
Generate-Path (u, currentPath)
BEGIN
  SWITCH(u)
  CASE a normal node :
       IF Adj[u] not empty THEN
            v = Adj[u].nextNode()
            currentPath.add( (u,v) )
            Generate-Path(v, currentPath)
       ELSE   // Cas où l'on arrive au noeud final du graphe
            IF PNBD is empty THEN
                 LCT.add(currentPath)
            ELSE
                 nbd = PNBD.get()
                 // Lier le chemin courant au nœud de bifurcation consulté nbd
                 currentPath.node = nbd.node
                 currentPath.branch = nbd.currentBranch
                 PCB.puch(currentPath)
            ENDIF
       ENDIF
       BREAK
  CASE a decision node :
       FOREVERY v ∈ Adj[u] DO
            tempPath = createPathCopy(currentPath)
            tempPath.add( (u,v) )
            Generate-Path(v, tempPath)
       ENDFOREVERY
       BREAK
  CASE a join node :
       IF ( !u.visited ) THEN
            u.visited = true
            v = Adj[u].nextNode()
            currentPath.add( (u,v) )
            Generate-Path(v, currentPath)
       ELSE
            nbd = PNBD.get()
            currentPath.node = nbd.node
            currentPath.branch = nbd.currentBranch
            PCB.puch(currentPath)
       ENDIF
       BREAK
```

```
CASE a fork node :
     branchCounter = adjNodeNbr(u)
     nbd = (node = u, branchNbr = branchCounter, currentBranch = 0)
     PNBD.puch(nbd)
     FOREVERY v ∈ Adj[u] DO
          nbd = PNBD.pop()
          nbd.currentBranchToNext()
          PNBD.puch(nbd)
          IF (nbd.currentBranch < 2) THEN // Si l'on est à la 1ère branche de 'u'
               currentPath.add( (u,v) )
          ELSE
               currentPath = newPath()
               currentPath.add( (u,v) )
          ENDIF
          Generate-Path(v, currentPath)
             /* Une fois le(s) sous-chemin(s) enveloppé(s) par une branche du noeud 'u'
             est(sont) généré(s), on le(s) traite par la procédure toProcessSubPath() */
          toProcessSubPath(nbd)
     ENDFOREVERY
        /*Une fois toutes les branches de 'u' (tous ses nœuds adjacents) ont été traitées pour
        générer les sous-chemins qu'elles enveloppent, on invoque concludeProcessing()*/
     concludeProcessing()
     BREAK
CASE a merge node
     IF ( !u.visited ) THEN
          u.visited = true
          v = Adj[u].nextNode()
          currentPath.add( (u,v) )
          Generate-Path(v, currentPath)
     ELSE   // Le currentPath croise d'autres chemins au nœud 'u' qu'ils incluent déjà
          IF PNBD is empty THEN
               FOREVERY path p ∈ LCT WITH (u ∈ p) DO
                    tempPath = copyRest(p,u)
                    tempPath = concat(currentPath, tempPath)
                    LCT.add(tempPath)
               ENDFOREVERY
          ELSE
               nbd = PNBD.get()
               FOREVERY path p ∈ (LCT or PCB) WITH (u ∈ p) DO
                    tempPath = copyRest(p,u)
                    tempPath = concat(currentPath, tempPath)
                    tempPath.node = nbd.node
                    tempPath.branch = nbd.currentBranch
                    PCB.puch(tempPath)
               ENDFOREVERY
          ENDIF
     ENDIF
     BREAK
ENDSWITCH
END
```

```
  /********************** PROCÉDURE toProcessSubPath() **********************/
toProcessSubPath( nbd )
BEGIN
   IF (nbd.currentBranch == 1) THEN // Si l'on est à la 1ère branche de nbd
        // Déplacer dans la LCI les chemins de la PCB qui sont liés au noeud nbd
        FOREVERY path p ∈ PCB WITH (p.node == nbd.node) DO
             LCI.add(p)
             PCB.pop() // Éliminer le chemin p
        ENDFOREVERY
   ELSE
        /* Concaténer chacun des sous-chemins de la PCB liés au noeud nbd avec chacun des
        chemins de la LCI liés à ce même noeud et plus précisément à sa branche précédente */
        FOREVERY path p ∈ PCB WITH (p.node == nbd.node) DO
             FOREVERY path r ∈ LCI WITH ( (r.node == nbd.node)
                       AND (r.branch == nbd.currentBranch -1) )
                  tempPath = concat(r, p)
                  tempPath.node = nbd.node
                  tempPath.branch = nbd.currentBranch
                  LCI.add(tempPath)
             ENDEVERY
             PCB.pop() // Éliminer le chemin p venant d'être traité
        ENDFOREVERY
        FOREVERY path p ∈ LCI WITH ( (p.node == nbd.node)
                  AND (p.branch == nbd.currentBranch -1) )
             LCI.delete(p)
        ENDFOREVERY
   ENDIF
END
  /********************** PROCÉDURE concludeProcessing() **********************/
BEGIN
   nbd = PNBD.pop()
   IF PNBD is empty THEN
        // Vider les chemins de la LCI dans la LCT
        FOREVERY path p ∈ LCI DO
             LCT.add(p)
             LCI.delete(p) // Supprimer le chemin p de la LCI
        ENDFOREVERY
   ELSE
        /* Déplacer les chemins de la LCI qui sont liés au noeud nbd (venant d'être retiré de la
        PNBD) dans la PCB et cela après les avoir liés au noeud de bifurcation suivant */
        nextNbd = PNBD.get()
        FOREVERY chemin p ∈ LCI AVEC (p.node == nbd.node) FAIRE
             p.node = nextNbd.node
             p.branch = nextNbd.currentBranch
             PCB.puch(p)
             LCI.delete(p) // Supprimer le chemin p de la LCI
        ENDFOREVERY
   ENDIF
END
```

APPENDICE D

LA SPÉCIFICATION BPEL DES PROCESSUS TRAVEL ET LOAN APPROVAL

D.1 Le processus *Travel*

Ci-dessous, la spécification BPEL du processus *Travel* est fournie. Cette spécification a été adaptée de celle proposée par Juric et *al.* (19) de façon à la rendre conforme à la nouvelle version de la norme BPEL (i.e. BPEL2.0).

```
<process name="Travel"
    targetNamespace="http://packtpub.com/bpel/travel/"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:trv="http://packtpub.com/bpel/travel/"
    xmlns:emp="http://packtpub.com/service/employee/"
    xmlns:aln="http://packtpub.com/service/airline/" >
<partnerLinks>
    <partnerLink name="client"
      partnerLinkType="trv:travelLT"
      myRole="travelService"
      partnerRole="travelServiceCustomer"/>
   <partnerLink name="employeeTravelStatus"
      partnerLinkType="emp:employeeLT"
      partnerRole="employeeTravelStatusService"/>
   <partnerLink name="AmericanAirlines"
      partnerLinkType="aln:flightLT"
```

```
      myRole="airlineCustomer"
      partnerRole="airlineService"/>
   <partnerLink name="DeltaAirlines"
      partnerLinkType="aln:flightLT"
      myRole="airlineCustomer"
      partnerRole="airlineService"/>
</partnerLinks>
<variables>
    <!- input for this process ->
    <variable name="TravelRequest"
        messageType="trv:TravelRequestMessage"/>
    <!- input for the Employee Travel Status web service ->
    <variable name="EmployeeTravelStatusRequest"
        messageType="emp:EmployeeTravelStatusRequestMessage"/>
    <!- output from the Employee Travel Status web service ->
    <variable name="EmployeeTravelStatusResponse"
        messageType="emp:EmployeeTravelStatusResponseMessage"/>
    <!- input for American and Delta web services ->
    <variable name="FlightDetails"
        messageType="aln:FlightTicketRequestMessage"/>
    <!- output from American Airlines ->
    <variable name="FlightResponseAA" messageType="aln:TravelResponseMessage"/>
    <!- output from Delta Airlines ->
    <variable name="FlightResponseDA" messageType="aln:TravelResponseMessage"/>
    <!- output from BPEL process ->
    <variable name="TravelResponse" messageType="aln:TravelResponseMessage"/>
</variables>
<sequence>
    <!- Receive the initial request for business travel from client ->
    <receive partnerLink="client"
       portType="trv:TravelApprovalPT"
       operation="TravelApproval"
       variable="TravelRequest"
       createInstance="yes" />
```

```
<!- Prepare the input for the Employee Travel Status Web Service ->
 <assign>
    <copy>
       <from variable="TravelRequest" part="employee"/>
       <to variable="EmployeeTravelStatusRequest" part="employee"/>
    </copy>
 </assign>
 <!- Synchronously invoke the Employee Travel Status Web Service ->
 <invoke partnerLink="employeeTravelStatus"
    portType="emp:EmployeeTravelStatusPT"
    operation="EmployeeTravelStatus"
    inputVariable="EmployeeTravelStatusRequest"
    outputVariable="EmployeeTravelStatusResponse" />
 <!- Prepare the input for AA and DA ->
 <assign>
    <copy>
       <from variable="TravelRequest" part="flightData"/>
       <to variable="flightDetails" part="flightData"/>
    </copy>
    <copy>
       <from variable="EmployeeTravelStatusResponse" part="travelClass"/>
       <to variable="flightDetails" part="travelClass"/>
    </copy>
 </assign>
 <!- Make a concurrent invocation to AA in DA ->
 <flow>
    <sequence>
       <!- Async invoke of the AA web service and wait for the callback ->
       <invoke partnerLink="AmericanAirlines"
          portType="aln:FlightAvailabilityPT"
          operation="FlightAvailability"
          inputVariable="FlightDetails" />
       <receive partnerLink="AmericanAirlines"
          portType="aln:FlightCallbackPT"
          operation="FlightTicketCallback"
          variable="FlightResponseAA" />
    </sequence>
```

```
      <sequence>
          <!- Async invoke of the DA web service and wait for the callback ->
          <invoke partnerLink="DeltaAirlines"
             portType="aln:FlightAvailabilityPT"
             operation="FlightAvailability"
             inputVariable="FlightDetails" />
          <receive partnerLink="DeltaAirlines"
             portType="aln:FlightCallbackPT"
             operation="FlightTicketCallback"
             variable="FlightResponseDA" />
       </sequence>
    </flow>
   <if>
       <condition>
          $FlightResponseAA.confirmationData/aln:Price
          &lt;= $FlightResponseDA.confirmationData/aln:Price
       </condition>
      <!- Select American Airlines ->
      <assign>
          <copy>
             <from variable="FlightResponseAA" />
             <to variable="TravelResponse" />
          </copy>
      </assign>
      <else>
         <!- Select Delta Airlines ->
         <assign>
             <copy>
                <from variable="FlightResponseAA" />
                <to variable="TravelResponse" />
             </copy>
         </assign>
      </else>
   </if>
   <!- Send a response to the client ->
   <reply partnerLink="client"
       portType="trv:TravelApprovalPT"
       operation="TravelApproval"
       variable="TravelResponse"/>
 </sequence>
</process>
```

D.2 Le processus *loan Approval*

Ci-dessous, la spécification BPEL du processus *loan Approval* est fournie. Cette spécification est une variante de celle proposée par la norme BPEL (1). Cette variante exclut le traitement d'erreur prévue dans la version d'origine.

```
<process name="loanApprovalProcess"
   targetNamespace="http://example.com/loan-approval/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:lns="http://example.com/loan-approval/wsdl/"
   suppressJoinFailure="yes">
   <import importType="http://schemas.xmlsoap.org/wsdl/"
      location="loanServicePT.wsdl"
      namespace="http://example.com/loan-approval/wsdl/" />
   <partnerLinks>
      <partnerLink name="customer"
         partnerLinkType="lns:loanPartnerLT"
         myRole="loanService" />
      <partnerLink name="approver"
         partnerLinkType="lns:loanApprovalLT"
         partnerRole="approver" />
      <partnerLink name="assessor"
         partnerLinkType="lns:riskAssessmentLT"
         partnerRole="assessor" />
   </partnerLinks>
   <variables>
      <variable name="request"
         messageType="lns:creditInformationMessage" />
      <variable name="risk"
         messageType="lns:riskAssessmentMessage" />
      <variable name="approval"
         messageType="lns:approvalMessage" />
   </variables>
```

```
<flow>
    <links>
       <link name="receive-to-assess" />
       <link name="receive-to-approval" />
       <link name="approval-to-reply" />
       <link name="assess-to-setMessage" />
       <link name="setMessage-to-reply" />
       <link name="assess-to-approval" />
    </links>
    <receive partnerLink="customer"
       portType="lns:loanServicePT"
       operation="request"
       variable="request"
       createInstance="yes">
       <sources>
          <source linkName="receive-to-assess">
             <transitionCondition>
                $request.amount &lt; 10000
             </transitionCondition>
          </source>
          <source linkName="receive-to-approval">
             <transitionCondition>
                $request.amount >= 10000
             </transitionCondition>
          </source>
          </sources>
    </receive>
    <invoke partnerLink="assessor"
       portType="lns:riskAssessmentPT"
       operation="check"
       inputVariable="request"
       outputVariable="risk">
       <targets>
          <target linkName="receive-to-assess" />
       </targets>
       <sources>
          <source linkName="assess-to-setMessage">
             <transitionCondition>
                $risk.level='low'
             </transitionCondition>
          </source>
```

```
        <source linkName="assess-to-approval">
            <transitionCondition>
              $risk.level!='low'
            </transitionCondition>
          </source>
        </sources>
      </invoke>
      <assign>
        <targets>
          <target linkName="assess-to-setMessage" />
        </targets>
        <sources>
          <source linkName="setMessage-to-reply" />
        </sources>
        <copy>
          <from> <literal>yes</literal> </from>
          <to variable="approval" part="accept" />
        </copy>
      </assign>
      <invoke partnerLink="approver"
        portType="lns:loanApprovalPT"
        operation="approve"
        inputVariable="request" outputVariable="approval">
        <targets>
          <target linkName="receive-to-approval" />
          <target linkName="assess-to-approval" />
        </targets>
        <sources>
          <source linkName="approval-to-reply" />
        </sources>
      </invoke>
      <reply partnerLink="customer"
        portType="lns:loanServicePT"
        operation="request" variable="approval">
        <targets>
          <target linkName="setMessage-to-reply" />
          <target linkName="approval-to-reply" />
        </targets>
      </reply>
    </flow>
</process>
```

BIBLIOGRAPHIE

(1) Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. Technical report, OASIS, April 2007.

(2) George L. Benwell and Stephen G. MacDonell. Assessing the graphical and algorithmic structure of hierarchical coloured Petri nets. Technical Report 94/3, P.O. Box 56, Dunedin, New Zealand, 1994.

(3) Claude Berge. Graphes et hypergraphes, 1970. Paris, Dunod.

(4) M. Berkelaar. LP_SOLVE. `http://www.w3.org/TR/ws-cdl-10/`.

(5) Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society.

(6) Antonio Bucchiarone, Hernan Melgratti, and Francesco Severoni. Testing service composition. In *Proceedings of the 8th Argentine Symposium on Software Engineering (ASSE'07)*, Mar del Plata, Argentina, August 2007.

(7) Jorge Cardoso. Control-flow complexity measurement of processes and weyuker's properties. *Enformatika*, 8:213–218, October 2005.

(8) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

(9) Francisco Curbera, Yaron Goland, Johannes Klein, Frank Leymann, Dieter Roller, Satish Thatte, and Sanjiva Weerawarana. Business process execution language for web services, version 1.1. Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation, May 2003.

(10) Edsger W. Dijkstra. Notes on structured programming : Technical University Eindhoven, 1970.

(11) Yves Drothier. BPEL ou business process execution language, 2006. Le Journal du Net, `http://www.journaldunet.com/solutions/0607/060712-qr-bpel.shtml`.

(12) Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

(13) Michael Ellims, James Bridges, and Darrel C. Ince. Unit testing in practice, November 2004. 15th International Symposium on Software Reliability Engineering, ISSRE'04. IEEE Computer Society.

(14) Active Endpoints. `http://www.activevos.com`.

(15) Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Abstract operational semantics of the business process execution language for web services. Technical report, Simon Fraser University, Burnaby B.C. Canada, April 2004.

(16) Andrea Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.

(17) Gartner Group. `http://www.gartner.com`.

(18) Zhang Guangmei, Chen Rui, Li Xiaowei, and Han Congying. The automatic generation of basis set of path for path testing. In *ATS '05: Proceedings of the 14th Asian Test Symposium on Asian Test Symposium*, pages 46–51, Washington, DC, USA, 2005. IEEE Computer Society.

(19) Matjaz Juric, Benny Mathew, and Poornachandra Sarang. *Business Process Execution Language for Web Services 2nd Edition.* Packt Publishing, 2006.

(20) Tetsuro Katayama, Eisuke Itoh, Zengo Furukawa, and Kazuo Ushijima. Test-case generation for concurrent programs with the testing criteria using interaction sequences. pages 590–597, Takamatsu, Japan, December 1999. IEEE Computer Society.

(21) Arnold Kaufmann and Guy G.Boulaye. *Théorie des treillis en vue des applications.* Masson, Paris France, 1978.

(22) Bartek Kiepuszewski, Arthur H. M. ter Hofstede, and Wil M. P. van der Aalst. Fundamentals of control flow in workflows. *Acta Inf.*, 39(3):143–209, 2003.

(23) Pramod V. Koppol, Richard H. Carver, and Kuo-Chung Tai. Incremental integration testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 28(6):607–623, 2002.

(24) Theerapong Lertphumpanya and Twittie Senivongse. Basis path test suite and testing process for WS-BPEL. *W. Trans. on Comp.*, 7(5):483–496, 2008.

(25) Frank Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, May 2001.

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

(27) Zhong Jie Li and Wei Sun. BPEL-unit: Junit for BPEL processes. volume 4294, pages 415–426. Springer Berlin / Heidelberg, November 2006.

(28) Zhongjie Li, Wei Sun, Zhong Bo Jiang, and Xin Zhang. BPEL4WS unit testing: Framework and implementation. *Web Services, IEEE International Conference on*, 0:103–110, 2005.

(29) Hehui Liu, Zhongjie Li, Jun Zhu, and Huafang Tan. Business process regression testing. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 157–168, Berlin, Heidelberg, 2007. Springer-Verlag.

(30) Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0. Brisbane, Australia, September 2007. Springer-Verlag.

(31) Niels Lohmann. A feature-complete Petri net semantics for WS-BPEL 2.0 and its compiler BPEL2oWFN. Informatik-Berichte 212, Humboldt-Universität zu Berlin, August 2007.

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

(33) Niels Lohmann, H.M.W. Verbeek, Chun Ouyang, and Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *IJBPIM*, 2008.

(34) Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. pages 287–301, 2001.

(35) Didier Maquin. Eléments de théorie des graphes, Mai 2003. Institut National Polytechnique de Lorraine.

(36) Eric A. Marks and Michael Bell. *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology.* John Wiley & Sons, June 2006.

(37) Philip Mayer. Design and implementation of a framework for testing BPEL compositions. Master's thesis, University of Hannover, September 2006.

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

(39) Thomas J. McCabe. A complexity measure. *IEEE Transactions On Software Engineering*, 2(4):308–320, December 1976.

(40) Simon Moser, Axel Martens, Katharina Gorlach, Wolfram Amme, and Artur Godlinski. Advanced verification of distributed WS-BPEL business processes incorporating CSSA-based data flow analysis. *Services Computing, IEEE International Conference on*, 0:98–105, 2007.

(41) M.Prasanna, S.N. Sivanandam, R.Venkatesan, and R.Sundarrajan. A survey on automatic test case generation. *Academic Open Internet Journal*, 15, July 2005.

(42) Ranjit Mulye. Meteor-s process design and development tool. Master's thesis, University of Georgia, Athens Georgia, 2005.

(43) Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

(44) Yefim V. Natis. Service-oriented architecture scenario, 2003. `http://www.gartner.com/DisplayDocument?doc\_cd=114358`.

(45) OASIS. Organization for the advancement of structured information standards, 2009. `http://www.w3.org/TR/wsa-reqs/`.

(46) The University of Hannover. BPELUnit - the open source unit testing framework for BPEL. `http://www.se.uni-hannover.de/forschung/soa/bpelunit`.

(47) Oracle. BPEL Process Manager.

(48) Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

(49) Flavio De Paoli and Sandro Morasca. Extending software complexity metrics to concurrent programs. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, Chicago, IL, USA, November 1990.

(50) Chris Peltz. Web services orchestration: a review of emerging technologies, tools, and standards, January 2003. Hewlett Packard, Co.

(51) Joseph Poole. A method to determine a basis set of paths to perform program testing (nistir 5737). Technical report, U.S. Department of Commerce/National Institute of Standards and Technology, November 1995.

(52) Leonard Richardson and Sam Ruby. *RESTful Web Services.* O'Reilly, 2007.

(53) Maurice Rozenberg. *Test logiciel.* Eyrolles, 1998.

(54) Philippe Schnoebelen, Béatrice Bérard, Michel Bidoit, François Laroussinie, and Antoine Petit. *Vérification de logiciels : techniques et outils du model-checking.* Vuibert, April 1999.

(55) Frédéric Servais. Verifying and testing BPEL processes. Master's thesis, Université Libre de Bruxelles, 2006-2007.

(56) Christian Stahl. A Petri net semantics for BPEL. Technical report, Humboldt University of Berlin, 2005.

(57) Zoran Stojanovic and Ajantha Dahanayake. *Service-oriented Software System Engineering Challenges And Practices.* IGI Publishing, Hershey, PA, USA, 2005.

(58) Gabor Szasz. *Théorie des treillis.* Dunod, Paris France, 1971.

(59) Kuo-Chung Tai. Testing of concurrent software. Orlando, FL, USA, September 1989. IEEE.

(60) R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

(61) S. Thatte. Xlang, 2001. `http://www.gotdotnet.com/team/xml\_wsspecs/xlang-c/default.htm`.

(62) Jorgen Thelin. A comparison of service-oriented, resource-oriented, and object-oriented architecture styles, 2003. Cape Clear Software.

(63) W. M. P. van der Aalst, Ter, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

(64) H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL processes using Petri nets. In *Proceedings of the Second International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, pages 59–78, Florida International University, Miami, Florida, USA, June 2005. D. Marinescu.

(65) W3C. Web service architecture. `http://www.w3.org/TR/ws-arch/`.

(66) W3C. Web services architecture requirements. `http://www.w3.org/TR/wsa-reqs/`.

(67) W3C. WS-CDL (web services choreography description language), 2005. `http://www.w3.org/TR/ws-cdl-10/`.

(68) W3C. The world wide web consortium, 2009. `http://www.w3.org/`.

(69) Arthur H. Watson and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical report, Computer Systems Laboratory, NIST, September 1996.

(70) Arthur Henry Watson. *Structured testing: analysis and extensions.* PhD thesis, November.

(71) Stewart N. Weiss. A formal framework for the study of concurrent program testing. Banff, Alta., Canada, July 1988. IEEE.

(72) Petia Wohed, Wil M.P. van der Aalst, Wil M. P, Marlon Dumas, and Arthur H.M. ter Hofstede. Analysis of web services composition languages: The case of BPEL4WS. In *Proc. of ER'03, LNCS 2813*, pages 200–215. Springer Verlag, 2003.

(73) W. Eric Wong, Yu Lei, and Xiao Ma. Effective generation of test sequences for structural testing of concurrent programs. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 539–548, Washington, DC, USA, 2005. IEEE Computer Society.

(74) Spyros Xanthakis, Michel Maurice, Antonio de Amescua, Olivier Houri, and Luc Griffet. *Test et contrôle des logiciels. Méthodes, techniques et outils.* EC2, 269, rue de la Garenne, 92024 Nanterre Cedex France, 1994.

(75) Spyros Xanthakis, Pascal Régnier, and Constantinos Karapoulios. *Le test des logiciels.* HERMES Science Publications, Paris, France, 2000.

(76) Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun, and Jian Zhang. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. *issre*, 0:75–84, 2006.

(77) Jun Yan and Jian Zhang. An efficient method to generate feasible paths for basis path testing. *Inf. Process. Lett.*, 107(3-4):87–92, 2008.

(78) R.-D. Yang and C.-G. Chung. A path analysis testing of concurrent programs. *Inf. Softw. Technol.*, 34(1):43–56, 1992.

(79) Ren-Dar Yang and Chyan-Goei Chung. A path analysis approach to concurrent program testing. pages 425–432, Scottsdale, AZ, USA, march 1990. IEEE Computer Society.

(80) Ren-Dar Yang and Chyan-Goei Chung. A path analysis approach to concurrent program testing. Scottsdale, AZ, USA, March 1990. IEEE.

(81) Yuan Yuan, Zhongjie Li, and Wei Sun. A graph-search based approach to BPEL4WS test generation. *icsea*, 0:14, 2006.

(82) Jian Zhang and Xiaoxu Wang. A constraint solver and its application to path feasibility analysis. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):139–156, 2001.