UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

ÉCOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR DE LILLE

THÈSE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Arnaud BAILLY

TEST & VALIDATION DE COMPOSANTS LOGICIELS

Soutenue publiquement le 15 décembre 2005 devant la commission d'examen :

Président	: Pr. Laurence DUCHIEN	LIFL, Université de Lille I
Rapporteurs	: Pr. Ana CAVALLI	INT, Évry
	: M. Jean-Louis LANET, HDR	GEMPLUS LABS, La Ciotat
Examinateurs	: Pascal FLAMENT	NORSYS, Ennevelin
Directrice	: Pr. Mireille CLERBOUT	LIFL, Université de Lille I
Co-encadrante	: Dr. Isabelle SIMPLOT-RYL	LIFL, Université de Lille I

Remerciements

Merci tout d'abord à Marie-Aimée de m'avoir soutenu et supporté tout au long de ces sept années. C'est promis, je ne le ferai plus, ou alors j'attendrai la retraite.

Merci à Isabelle Ryl devenue Simplot-Ryl d'être venue me chercher en licence, de m'avoir gardé en DEA et de m'avoir proposé ce sujet de thèse. Merci à Mireille Clerbout d'avoir accepté de diriger et co-encadrer ce travail.

Merci à Pascal Flament, Sylvain Breuzard et au Conseil Régional Nord-Pas-de-Calais de m'avoir permis de manger et faire manger ma famille pendant ces trois ans et demi. Merci aussi aux ASSEDIC qui m'ont rémunéré pendant les trois années ayant précédé la thèse.

Merci à Ana Rosa Cavalli et à Jean-Louis Lanet d'une part, à Laurence Duchien d'autre part, d'avoir accepté respectivement de rapporter ce travail un peu particulier et de présider le jury de soutenance, malgré leur emploi du temps surchargé.

Merci à Yves Roos, Michel Latteux, Isabelle et Mireille encore, Jean-Marc Talbot, Sophie Tison, Jean Berstel, Jacques Sakarovitch et plein d'autres de m'avoir fait découvrir le monde merveilleux des automates et plus généralement de l'informatique théorique. Je ne manquerais pas de propager la bonne parole dans la mesure de mes moyens même si hélas, les temps sont durs pour la théorie.

Merci à Raphaël Marvie, Emmanuel Renaux, Philippe Merle, Olivier Barais, Lionel Seinturier, Dolorès Diaz, Jérémie Hattat, Renaud Pawlak pour les discussions souvent éclairantes sur la vraie nature des composants et l'existence d'un génie du logiciel.

Merci à Mirabelle Nebut, Yann Hodique, Iovka Boneva, Denis Debarbieux, encore Yves Roos et de manière générale toute l'équipe STC et une grande partie du LIFL pour leur agréable compagnonnage de bureau et leur indulgence envers mes sautes d'humeur.

Pour les mêmes raisons, merci à tous les collègues de Norsys avec lesquels j'ai eu l'occasion de discuter, travailler et même rigoler ces dernières années : Pascal bien sûr, Thomas Recloux, Christophe Cordennier, Xavier Farine, Éric Sillègue, Edwige Renaux et tous les autres.

Merci à Florence Mignard de m'avoir accompagné durant le *Nouveau Chapitre de Thèse* et à l'*Association Bernard Grégory* d'avoir permis que cela se fasse.

Merci à Douglas Hofstadter d'avoir écrit « *Gödel Escher Bach* ». C'est après la lecture de son livre que je me suis décidé à reprendre des études d'informatique. Merci donc aux enseignants et non-enseignants du FIL qui ont rendu ces années d'études souvent passionnantes, Patricia Caron, Bruno Bogaert, le SUDES, Jean-Paul Delahaye, Michel Petitot, Sophie Tison, Jean-Marc Geib, Philippe Mathieu, François Denis, David Simplot-Ryl et tous les autres.

Merci à Arnaud Fontaine, Damien Devigne et Lætitia Bonte d'avoir souffert en silence sur mon logiciel boursouflé durant leurs stages respectifs.

Merci enfin[1] à Isabelle Simplot-Ryl, Mireille Clerbout, Annie Audoux, Marie-Aimée Bailly et Jean-Louis Lanet pour leur vigilance dans la traque des fautes d'orthographe et de syntaxe innombrables de ce manuscrit.

[1] Si vous lisez ces lignes et que vous estimez mériter des remerciements que je ne vous ai pas accordé, envoyer moi un mail à `abailly@achilleus.net`

Table des matières

Introduction

Contexte

Ce travail de thèse s'est déroulée dans le cadre d'une collaboration entre la société NORSYS et le *Laboratoire d'Informatique Fondamental de Lille*, cofinancée par la Région Nord-Pas-de-Calais. NORSYS est une société de services informatiques créée en 1990 et spécialisée dans la réalisation de logiciels à façon pour des clients de taille et d'activité diverses : banque, assurance, secteur social, grande distribution. Les logiciels réalisés ont tous pour caractéristiques communes d'être des outils de gestion participant d'un système d'information d'entreprise, évoluant dans un environnement techniquement hétérogène, pour des utilisateurs de qualifications diverses. Il s'agit donc pour l'essentiel de traitement de l'information, autrement dit de l'*informatique de gestion*.

Le démarrage de cette thèse s'est fait à peu près conjointement avec l'apparition au sein de la société de ce qu'il était convenu d'appeler les *Nouvelles Technologies de l'Information et de la Communication*. Ce terme désigne l'ensemble des techniques — langages, environnements, réseaux, Internet, systèmes — qui ont transformé l'informatique à la fin du XX^{e} siècle. Les applications ont évolué depuis des systèmes centralisés — client lourd-site central — vers des technologies décentralisées dites *réparties* accessibles depuis des *clients légers*. Les systèmes sont devenus ouverts et communiquants. Ce changement technologique ne s'est pas fait et ne se fait pas sans heurts et il a nécessité la mise en œuvre de méthodes d'analyse, de conception et de développement différentes. Une partie non négligeable de mon travail au sein de l'entreprise a donc consisté à accompagner ce changement par la réalisation de prototypes, le conseil dans la conception d'architectures logicielles, la mise en œuvre de techniques de développement inspirées des pratiques de l'*open-source* et la formation des personnels.

Une autre « révolution », dont le terme n'a pas encore été atteint, est la généralisation du phénomène d'externalisation des développements ou développement *off-shore*. L'arrivée sur le marché des services informatiques de concurrents issus de pays émergents — Inde, Chine, ex-Pays de l'Est, Maghreb et Moyen-Orient — et proposant les services d'une main d'œuvre qualifiée à des salaires nettement inférieurs à ceux pratiqués dans les pays développés entraîne une baisse des prix généralisée dans les métiers du développement à façon de logiciels. La question se pose donc pour les sociétés de services et de manière plus générale les « professionnels de la profession » des pays développés de la stratégie à adopter face à cette évolution.

Parmi les solutions possibles, il en est une qui concerne directement mon activité au sein de l'entreprise, c'est la mise en place d'un processus de développement « industriel ». L'*intensité capitalistique* est l'une des barrières d'entrée à la concurrence les plus efficaces sur un marché : plus les investissements seront lourds pour espérer obtenir des parts de marché, moins les concurrents seront tentés de s'y intéresser. Le concept, initialement forgé pour l'industrie classique, peut parfaitement s'appliquer au secteur des services avec une modification de taille : ici le capital est essentiellement humain. Industrialiser les développements consistera donc à créer des procédures, des normes, des contrôles, bref une chaîne de production, qui permette simultanément de réduire le temps de développement et d'accroître la qualité de la production.

Problèmes

Cet objectif se traduit sur le plan scientifique en termes de problèmes de *génie logiciel*. Les grandes questions qui se posent sont donc : comment automatiser le processus de développement ? Comment

améliorer le contrôle de la qualité de ce processus ? Comment accroître la fiabilité — mesurée en taux de défauts résiduels — des applications produites ?

Cette thèse cherche à contribuer à la résolution de ces problèmes, à partir des trois hypothèses suivantes :

1. les *composants logiciels* sont la bonne réponse à la complexité croissante des applications, si on ne se limite pas à leur utilisation dans le cadre de plate-formes technologiques *ad hoc* mais si on s'en sert aussi comme un outil pour concevoir les applications et leur architecture ;
2. la vérification et la validation automatisées des composants produits doivent être au cœur du processus de développement pour assurer un niveau de qualité élevé et constant ;
3. des contrats comportementaux formalisés supportant de multiples niveaux d'abstraction sont le carburant dont peut se nourrir un tel processus.

Il est bien connu que le test représente une part très importante du coût d'un projet de développement, les évaluations se trouvant généralement dans une fourchette de 30 à 50% du coût total. Ce que l'on entend par *test* dans ces évaluations est le *test système* ou dans notre contexte le *test de recette* — ou plus communément *la recette* — c'est-à-dire la phase succédant au développement et durant laquelle le logiciel produit est testé par des utilisateurs. Ce coût comprend non seulement le temps nécessaire pour exécuter les plans de test en fonction des exigences initiales de l'application, mais aussi le temps nécessaire à l'analyse des résultats de ces tests, le temps pris à corriger les erreurs détectées et bien entendu le temps nécessaire à la correction des erreurs introduites par les corrections. Dans le meilleur des cas, ce processus converge jusqu'à ce que plus aucune erreur ne soit signalée ou jusqu'à ce qu'un certain seuil — de temps ou de niveau d'exigences — soit franchi.

Le coût de cette phase de recette est non seulement corrélé à la qualité des développements proprement dits mais aussi à la qualité du processus d'analyse et de modélisation des exigences, donc à l'ensemble des activités du processus de construction du logiciel. Il est évident que si les spécifications sont incomplètes, ambiguës, changeantes, si elles sont mal comprises par les concepteurs et développeurs, la phase de recette verra son coût augmenter considérablement. L'importance de disposer de spécifications précises est donc une fois encore à souligner.

A contrario, le coût des tests réalisés en *cours de développement*, ce que l'on appelle communément les *tests unitaires* est relativement faible : ces tests sont écrits au fil de l'eau par les équipes de développement et s'intègrent naturellement dans la réalisation de l'application proprement dite. Il est par conséquent évident que la meilleure manière de réduire le coût parfois prohibitif de la recette est de maximiser le nombre d'erreurs détectées le plus tôt possible, et donc d'introduire le plus tôt possible les tests fonctionnels complets de l'application de sorte que le passage en recette se fasse sur un logiciel déjà en grande partie testé, unitairement et globalement.

Ces hypothèses et constats nous ont conduits à formuler la proposition qui est défendue dans cette thèse et qui se compose : d'un modèle abstrait de composants permettant l'expression de contrats comportementaux à partir des interfaces et dépendances de composants ; d'une sémantique formelle de la statique et de la dynamique de ce modèle sous la forme de langages et d'opérations simples ; d'outils théoriques pour la validation et la vérification de composants concrets, et plus particulièrement par des méthodes de test de conformité ; d'une intégration de cette démarche dans le cycle de production du logiciel ; enfin d'un prototype d'outil démontrant les capacités d'automatisation des tests fonctionnels à partir d'un modèle de composants abstrait et selon différentes implantations cibles.

En résumé, je défends l'idée que des composants formalisés dans un langage idoine peuvent être testés automatiquement et utilisés à différents degrés d'abstraction de manière à obtenir et maintenir un niveau de qualité mesuré.

Résumé

Le premier chapitre est une analyse de l'existant : comment sont actuellement produites les applications, dans quels langages, avec quelles méthodes ; et quels sont les problèmes que posent ces techniques. Le chapitre 2 est une synthèse des travaux et systèmes existants que l'on peut rapprocher de près ou de loin de la notion d'architecture de composants. Nous étudierons des plate-formes dites « à composants », des langages de modélisation, des langages de description d'architectures, des modèles et systèmes formels

permettant de construire de telles architectures et de raisonner dessus. Le chapitre 3 se veut un panorama du test de logiciel, de son vocabulaire, de ses problèmes, plus particulièrement focalisé sur le *test fonctionnel* ou test *boîte noire*, et encore plus précisément sur le test de conformité à partir de modèles à systèmes de transitions.

Après cet état de l'art et de la pratique, le chapitre 4 décrit de manière informelle les différents éléments que nous avons choisi de prendre en compte et qui collectivement définissent le modèle FIDL d'architecture à composants. Ce chapitre contient une section introductive présentant les grandes caractéristiques du modèle, une description de la syntaxe du langage et deux exemples de modélisation. Le chapitre 5 est une définition formelle des automates FIDL, une variété d'automates qui décrivent la sémantique des composants et systèmes de composants. Sont aussi définis précisément les expressions qui servent à dénoter ces automates et le processus de transformation de l'un en l'autre. Enfin le chapitre 6 relie les deux chapitres précédents en décrivant comment sont construits les langages associés à chacun des éléments d'un modèle FIDL, depuis les données primitives jusqu'aux assemblages de composants. Nous montrons en particulier que la composition de composants préserve certaines propriétés « naturelles » des systèmes ouverts de composants. La question de l'héritage comportemental est aussi abordée succinctement.

Le chapitre 7 est consacré aux problèmes de vérifications des propriétés de composition des modèles FIDL et surtout du test de conformité de composants concrets par rapport à des spécifications. Nous montrons que ce problème peut être traité de manière *décompositionnelle* en s'intéressant à certaines parties significatives du comportement spécifié des composants. Le chapitre 8, enfin, introduit brièvement la question de la mise en œuvre concrète de l'ensemble de cette démarche basée sur des architectures de composants. La structure et les possibilités d'un prototype sont décrites, ainsi que les interactions des activités de V&V basées sur des modèles FIDL dans le processus de développement.

Je conclurai enfin par un résumé des contributions que j'estime être les plus importantes de ce travail et par une revue des perspectives ouvertes et qu'il serait souhaitable d'approfondir. Le lecteur pointilleux pourra se reporter aux annexes pour y découvrir le détail de la grammaire du langage FIDL ainsi qu'un glossaire des nombreux termes techniques et acronymes qui parsèment ce document.

Chapitre 1

Processus de développement

Les « nouvelles technologies » avaient pour ambition de pallier les insuffisances des modes traditionnels de production de logiciel — systèmes centralisés, client-serveurs — en particulier pour ce qui concernait la maintenance et l'évolutivité. Ces technologies — langages orienté-objets, clients légers universels, intergiciels, réseaux ouverts ... — ont induit une modification du processus de développement logiciel qui s'est faite progressivement au fil des nouveaux projets, avec l'intégration de nouvelles méthodes de conception dirigées par les modèles UML et de nouvelles pratiques inspirées des techniques des logiciels libres et de l'*eXtreme Programming*. Le constat que l'on peut faire aujourd'hui et qui sera développé dans ce chapitre est que, si la phase de développement et dans l'ensemble les aspects technologiques sont bien maîtrisés, la conception des systèmes, leur articulation avec l'analyse des besoins du client et surtout la qualité et la fiabilité du produit fini posent encore de nombreux problèmes. Ces problèmes nous semblent provenir d'un manque de maturité du processus, de compréhension globale de l'architecture des systèmes et de formalisation du processus de validation et de vérification des développements.

Nous commencerons ce chapitre par une présentation du processus de développement qui nous permettra d'exposer les pratiques actuelles, les outils et les méthodes. Cette présentation nous permettra par ailleurs de présenter nos contributions pratiques dans l'amélioration du suivi et de la qualité du processus de développement de ce type d'application. Cette première section sera suivie d'une analyse critique de ce processus et d'une première exposition des solutions envisageables qui nous permettra de mettre l'accent sur les besoins de formalisation et de vérification d'architectures.

1.1 État de la pratique

Norsys est une société de services et comme telle est amenée à réaliser des logiciels à façon pour un grand nombre de clients possédant chacun des métiers différents. Toutefois, tous les développements réalisés depuis quelques années dans le domaine des « nouvelles technologies » possèdent nombres de caractéristiques communes :

- le langage support est Java ;
- l'infrastructure technique est basée sur J2EE[149], la spécification orientée « composants » de Sun pour les systèmes d'information d'entreprises ;
- les phases de recueil des exigences, d'analyse et de conception utilisent une modélisation à base de diagrammes UML complétée de nombreux documents textuels ;
- le processus de développement est proche d'un processus de type RUP — *Rational Unified Process* — avec des adaptations spécifiques et des allégements par rapport aux préconisations du modèle ;
- l'architecture logicielle est structurée en *couches*, depuis la présentation jusqu'à la persistance des données. Les services transversaux — authentification, habilitation, contexte transactionnel, répartition de charges ... — sont assurés par l'infrastructure J2EE ou développés de manière *ad hoc* ;
- les systèmes développés ont généralement à s'intégrer dans un existant complexe, mélange d'applications client-serveurs, de bases de données, de moniteurs transactionnels et de systèmes centralisés

en COBOL.

La complexité des développements provient de la nécessité de fonctionner dans un environnement hétérogène avec des contraintes techniques et organisationnelles fortes. Il est rare de rencontrer dans ce type de logiciel de la complexité dans les fonctionnalités ou les algorithmes.

1.1.1 Infrastructure technique

La cible technique est constituée de systèmes répartis généralement redondants hébergés sur des fermes de serveurs. La figure 1.1 représente une architecture technique type sur laquelle sont déployées les applications produites. Il s'agit là bien évidemment d'une configuration idéale qui peut être adaptée en fonction des contraintes de coûts, de performances ou de sécurité du projet.

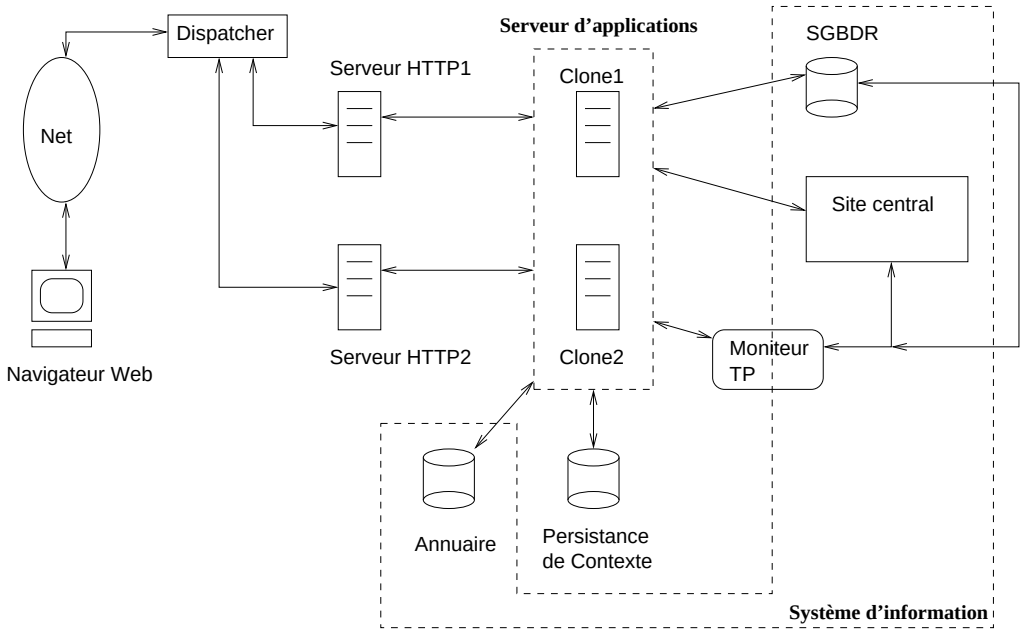


FIG. 1.1 – Architecture technique.

La communication avec le client est gérée par un ou plusieurs serveurs HTTP avec un répartiteur de charge permettant de minimiser temps de réponse et trafic réseau. Le serveur HTTP communique avec le *serveur d'applications* généralement au travers d'un protocole *ad hoc*, serveur qui peut être cloné de sorte que là aussi, la charge soit répartie sur plusieurs machines et la tolérance aux pannes soit améliorée. Le contexte, c'est-à-dire l'ensemble des informations relatives à une session, est persistant selon un mécanisme propre au serveur d'applications choisi : en cas de transfert de traitement d'un serveur à un autre ou de reprise sur erreur, le contexte peut être restauré assurant de manière transparente un service continu du point de vue du client. Les besoins d'authentification des clients, qu'ils soient réalisés par mots de passe ou par une infrastructure à clés publiques (PKI), sont le plus souvent gérés par un annuaire d'entreprise de type LDAP. Il en va de même pour l'habilitation des utilisateurs, c'est-à-dire pour la définition de leurs droits d'accès sur les fonctions du système. Notons que l'authentification se fait généralement au niveau des serveurs HTTP tandis que l'habilitation se fait dans le serveur d'applications.

Le serveur d'applications contient l'ensemble de la logique applicative (voir ci-dessous) et a la charge de communiquer au travers de connecteurs idoines avec le reste du système d'information : bases de données relationnelles, sites centraux, moniteurs transactionnels. L'architecture J2EE prévoit, au travers des connecteurs JCA, la possibilité de relier un tel système à n'importe quel autre en maintenant des propriétés transactionnelles. Pour une présentation générale de l'architecture de la plate-forme J2EE, voir la section 2.1.1 du chapitre 2.

1.1.2 Architecture logicielle

L'architecture logicielle utilisée est une architecture en couches qui reprend un certain nombre de « bonnes pratiques » désormais courantes dans le domaine. Elle s'inspire de *patrons de conceptions*[11, 62] usuels, tels que le patron *Modèle-Vue-Contrôleur* popularisé par le framework Struts pour la gestion des interactions avec le client, ou le patron *Recherche de Services — Service Locator —* pour la localisation d'interfaces métiers offertes par les EJB. La figure 1.2 reprend les différents éléments de manière synthétique.

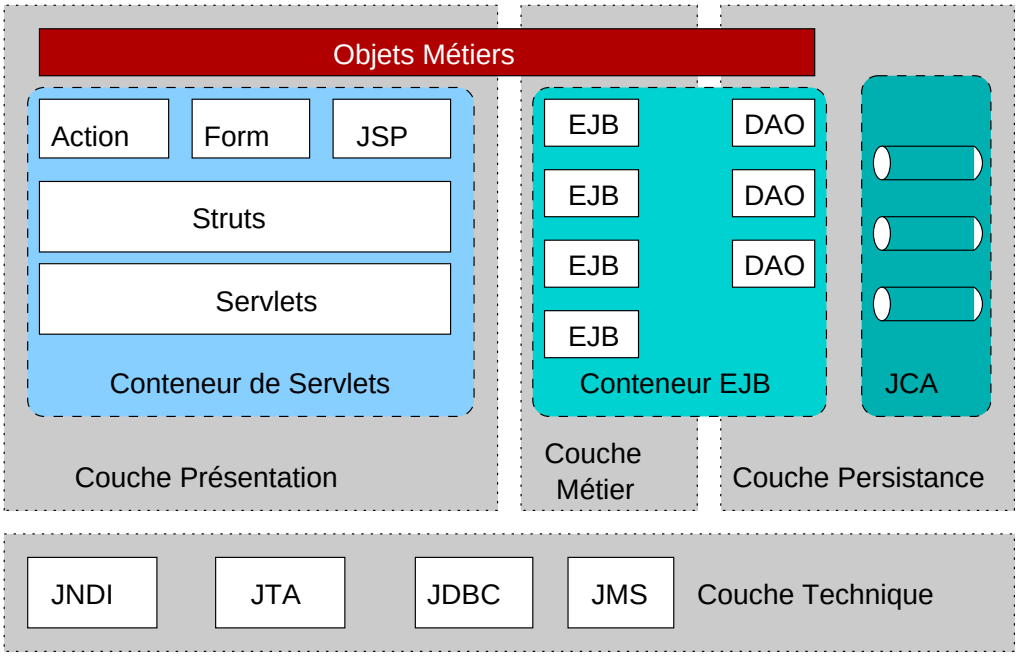


FIG. 1.2 – Architecture logicielle.

La *couche présentation* s'appuie sur des *servlets* encapsulées dans un conteneur, qui permettent de développer des contenus dynamiques en Java en s'abstrayant d'un certain nombre de détails techniques tels que la gestion de la transformation des paramètres — *marshalling* — depuis ou vers le protocole HTTP, la gestion du contexte de session et du contexte applicatif et surtout la communication par appels de méthodes distants avec la couche métier, c'est-à-dire les EJB. Struts fournit un canevas pour gérer les interactions avec le client et sur cette base sont développés des *actions*, des *formulaires — forms —* et des JSP — *Java Server Pages —* représentant respectivement la partie contrôle, modèle et vue du patron MVC.

La *couche métier* contient les *Enterprise Java Beans*, un ensemble de services métiers utilisés par la couche présentation en fonction des besoins de l'application. Ces EJB sont le plus souvent de type *Session sans état* et le traitement de l'accès aux données est délégué à des objets spécialisés appelés *Data Access Objects* ou DAO, plutôt que laissé à la discrétion du conteneur et d'EJB *Entités*.

La *couche persistance* est souvent la plus technique et la plus délicate. Dans l'idéal, elle réalise une simple projection du modèle de classes des objets métiers depuis ou vers un système de base de données quelconque, ce qui peut être fait de manière automatisée en s'appuyant sur un framework tel que les EJB *Entités* ou Hibernate. Dans la pratique, compte tenu du fait que les schémas de table sont généralement préexistants aux applications développées, que les canevas de projection objet-relationnel automatisés ne parviennent pas à optimiser correctement les transactions et les requêtes et que l'on n'a pas uniquement affaire à des systèmes relationnels mais aussi à des applications légataires, à des transactions stockées COBOL ou encore à des couches de moniteurs transactionnels propriétaires, cette gestion est le plus souvent développée de manière *ad hoc*.

La couche transversale *objets métiers* contient l'ensemble des objets du domaine applicatif, autrement

dit l'instantiation du diagramme de classes des données manipulées par l'application. Ces objets sont utilisés par toutes les couches car l'utilisation d'objets de granularité importante, voire de grappes d'objets, dans les interactions entre couches permet de réduire le nombre d'appels de méthodes réalisés et donc d'accroître les performances du système.

La *couche technique*, enfin, comprend un certain nombre de services sur lesquels s'appuient les différents composants du système :

- service de nommage — *Java Naming and Directory Interface* ;
- service de transaction — *Java Transaction API* ;
- service d'accès aux données — *Java DataBase Connectivity* ;
- service de messagerie asynchrone — *Java Message Service*.

1.1.3 Processus de développement

Le processus de développement utilisé est largement inspiré du RUP, mâtiné d'une bonne dose de pragmatisme et d'un zeste d'*eXtreme Programming*. Nous renvoyons aux travaux réalisés au sein de l'entreprise par E.RENAUX[136] pour une analyse plus poussée de ce processus et de ses défauts.

Il s'agit là d'un principe théorique qui dans la pratique recouvre un découpage plus « traditionnel » en activités et en lots :

- les phases d'*expression des besoins* et d'*analyse* se confondent. Elles produisent un cahier des charges, des diagrammes de cas d'utilisation, des scénarios et des règles métiers ;
- la *conception* produit les diagrammes de classes et éventuellement d'états-transitions associés, ainsi que les schémas de données ;
- le *développement* réalise effectivement l'application à partir des documents de conception ;
- la phase de test ou phase de *recette* valide l'application produite par rapport aux besoins initiaux.

Les itérations et incréments ont une granularité beaucoup plus grande que dans les préconisations des méthodes citées, problème important dont on verra par la suite qu'il découle de carences dans la modélisation de l'architecture globale de l'application.

Le processus de développement s'appuie par ailleurs sur une analyse d'*urbanisation* du système d'informations qui permet de découper globalement les processus métiers en différents *quartiers* et *blocs* reproduisant au niveau informatique l'organisation du client. Le processus tel que nous le présentons ici doit être compris comme une synthèse d'un ensemble de pratiques sur différents projets.

1.1.4 Méthodologie d'analyse & conception

La méthodologie articule la conception en *couches* et le résultat du processus d'analyse en *vues*. Les différentes couches recensées sont :

- la couche *Présentation* qui contient essentiellement les éléments régissant les interactions avec l'utilisateur : description de l'interface graphique, règles de navigation entre écrans ;
- la couche *Dynamique applicative* qui contient les processus de l'application proprement dite, c'est-à-dire essentiellement les actions Struts ;
- la couche *Métier* qui définit et contient les services métiers et les objets associés ;
- la couche *Persistance* qui décrit les règles d'interaction entre objets-métiers et système de stockage.

Le processus d'analyse et de conception produit un ensemble de *vues* qui sont chacune représentées par différents modèles et documents. Ces différentes vues sont :

- la **vue utilisateur** qui décrit les besoins de l'utilisateur en termes généraux ;
- la **vue processus** qui modélise chaque processus d'interaction entre un utilisateur et le système ;
- la **vue composant** qui identifie les composants du système, leur fonctionnement et leur interaction avec d'autres composants ;
- la **vue persistance** qui décrit les règles de projection entre objets-métiers et système de stockage persistant.

Nous décrivons succinctement chacune des vues dans les sections ci-dessous en proposant systématiquement une synthèse sous la forme d'un *méta-modèle* UML de chaque vue.

Vue utilisateur

Les besoins utilisateurs sont décrits sous la forme de diagrammes de cas d'utilisation et de diagrammes de classes. Un méta-modèle de cette vue utilisateur est donné ci-dessous (partie supérieure de la figure 1.3).

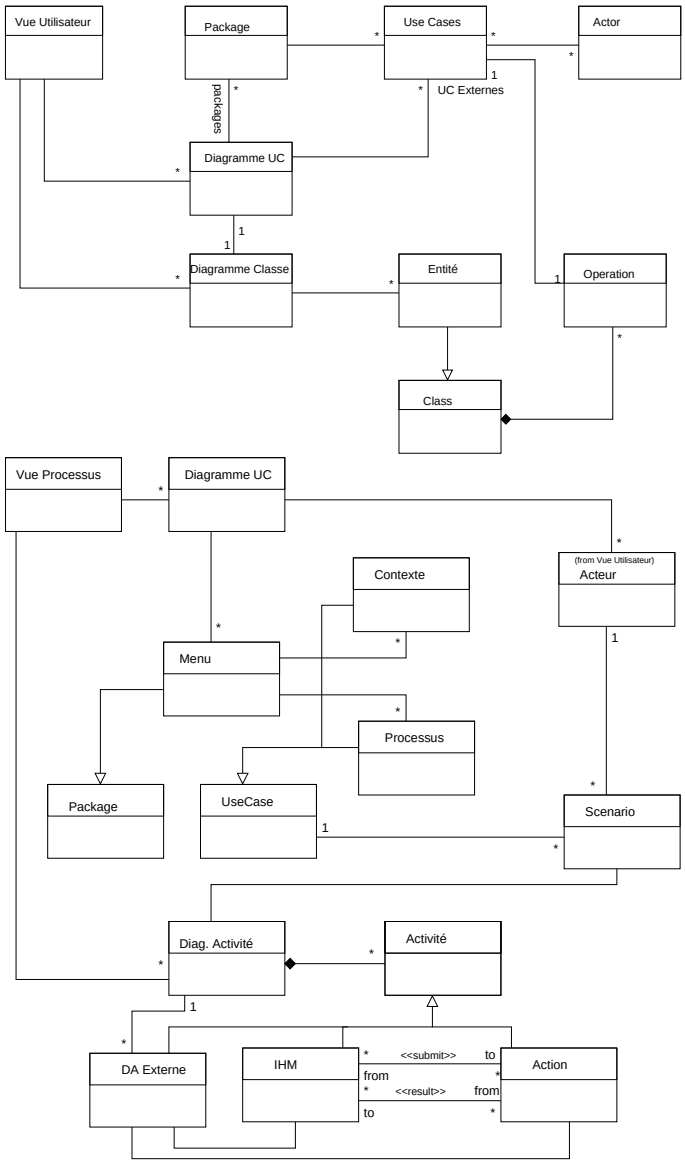


FIG. 1.3 – Vues utilisateurs & métiers.

Dans cette vue, on exprime les besoins de l'utilisateur, on définit le périmètre du système et les interactions avec les différents rôles de l'utilisateur. Cette vue permet de dégager des **entités fonctionnelles** (domaines) représentées dans le méta-modèle par une classe dont les attributs et les opérations sont définis par les différents cas d'utilisation.

Vue Processus

Cette vue, à partir des besoins utilisateurs exprimés précédemment, va définir précisément les interactions existant entre les différents rôles (acteurs) intervenant dans l'application et le système : c'est un *scénario*. Ces interactions sont représentées dans un diagramme de cas d'utilisation, chaque cas d'utilisation se trouvant par la suite associé à un ou plusieurs diagrammes d'activité. Les regroupements de cas

d'utilisation représentent en fait des menus/actions accessibles par différents rôles du système. Les interactions ou activités sont réalisées par des échanges entre des *éléments d'IHM* et des *actions* : une IHM permet d'invoquer une ou plusieurs actions qui en retour produit une nouvelle IHM.

Cette vue permet en particulier d'identifier et de caractériser les objets IHM utilisés (types d'éléments d'interfaces, ordres de navigation, accessibilité) et de définir les processus métiers, c'est-à-dire de documenter les algorithmes relatifs à chaque `Action` : quelles sont les éléments de données de l'IHM qui sont transmis et quels en sont les résultats ?

Vue Composants

La vue « composants » détaille :

- les éléments d'IHM, plus particulièrement du point de vue de la couche dynamique applicative, c'est-à-dire du *serveur d'application* dans l'architecture choisie ;
- les services métiers, c'est-à-dire les objets et leurs interfaces permettant de réaliser les fonctions demandées par l'utilisateur.

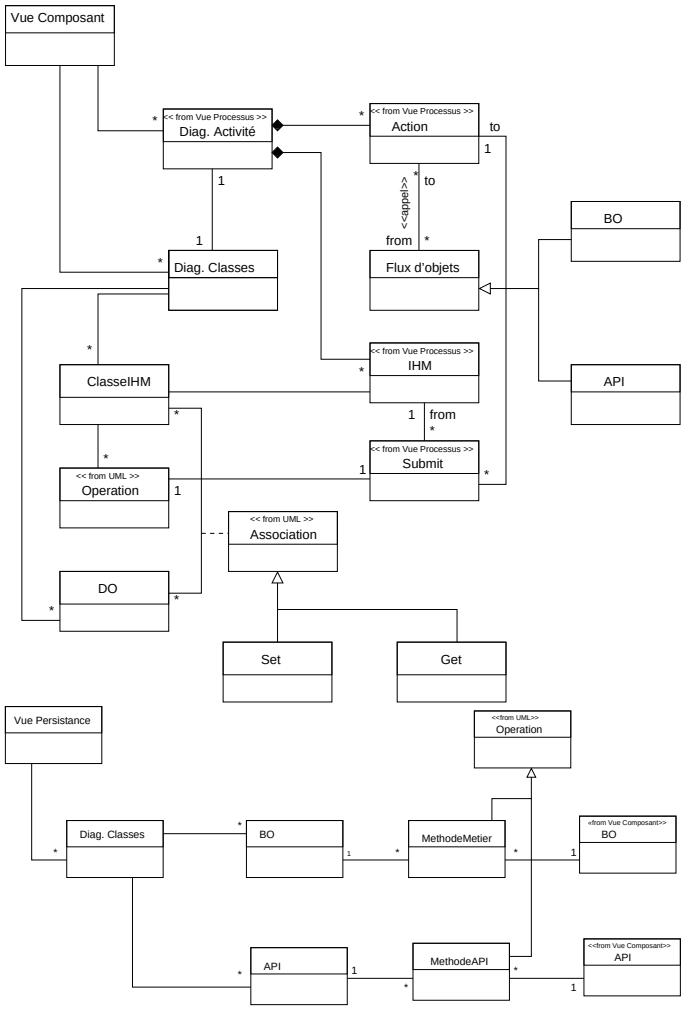


FIG. 1.4 – Vue composants métiers & persistance.

Le diagramme d'activité issu de la phase précédente est enrichi par des informations supplémentaires sur les éléments concrets de réalisation des classes IHM (maquette HTML, page JSP, attributs) et des éléments `Flux d'objet`. Ces flux d'objets correspondent à des invocations de services métiers (BO) ou d'éléments de l'infrastructure logicielle sous-jacente (API). À chaque transition déclenchée par une activité

d'IHM correspond une méthode dans la classe d'IHM correspondante, méthode qui sera documentée en décrivant les contrôles effectués sur les attributs de l'objet.

Vue métier/persistance

Cette dernière vue définit comment les objets métiers communiquent avec des données présentes dans des applications patrimoniales (sites centraux), des SGBD ou toute autre forme de stockage de longue durée. Comme indiqué dans le paragraphe précédent, les objets BO et API ont pour fonction de faire le lien entre les services du point de vue de l'application et les données et couches techniques. Des règles d'accès aux données persistantes sont par ailleurs définies à ce stade qui ne sont pas traitées dans le cadre de cette étude.

1.1.5 Infrastructure de développement

Les développements s'appuient sur un certain nombre d'outils et de pratiques ayant pour objectif d'accroître la qualité et la fiabilité du résultat final. D'une part est mise en œuvre une démarche globale d'*ingénierie dirigée par les modèles* permettant de tirer partie de l'existence de modèles de conception et de libérer le développeur des contraintes techniques induites par la plate-forme. Concrètement, cela signifie qu'à partir des diagrammes de classes des objets métiers, on utilise des outils de génération de code pour produire automatiquement les éléments nécessaires à l'infrastructure technique : interfaces distantes et fabriques dans le cas des EJB, fichiers de configurations XML pour la persistance des données et la couche présentation, formulaires Struts.

D'autre part, un processus d'*intégration continue* piloté par Maven et basé sur un *système de gestion de versions* permet d'automatiser la construction des livrables de l'application — paquetages, archives, descripteurs de déploiement —- et surtout de produire à intervalles réguliers une photographie de l'ensemble de l'application, sa documentation et une batterie de vérifications statiques et dynamiques. Ce principe est résumé dans la figure 1.5 et reprend la forme d'un patron MVC dans lequel les vues sont les différentes phases ou acteurs du processus de développement, le contrôleur est le moteur d'intégration continue et le modèle est le système de gestion de configurations.

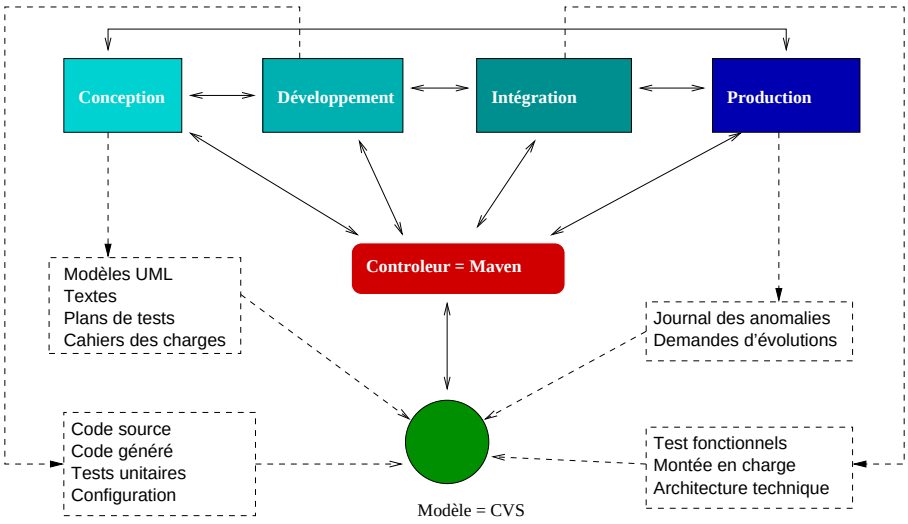


FIG. 1.5 – Modèle de processus de développement.

Les vérifications réalisées par le contrôleur à partir du code source sont les suivantes :

- vérifications des règles syntaxiques de codage : nommage des entités dans le code, indentation et formatage ... ;
- vérifications de règles — simples — de sémantique : code mort, variables inutilisées et déclarations incorrectes ... ;

– calculs de métriques par paquetage telles que le degré d'abstraction/concrétion, le taux de couplage, le degré de complexité ... ;
– vérification des règles d'accès architecturales entre classes et paquetages permettant de vérifier le respect par les développeurs de l'architecture globale du système ;
– exécution des tests unitaires et calcul de la couverture de code — instructions et branchements — réalisée.

Le développement et la mise en œuvre de ces outils a constitué une part importante de notre contribution au sein de l'entreprise et s'est étalée sur différents projets. En l'état actuel des choses, seule la vue *Développement* est réellement opérationnelle. Une partie de l'objectif de cette thèse est de faire en sorte que les autres vues, et plus particulièrement les vues *Conception* et *Intégration* soient mises en place par la gestion automatisée des tests fonctionnels à différents niveaux de l'architecture et la validation croisée entre modèles et sources.

1.2 Analyse critique

Après un rapide tour d'horizon du processus type de développement d'applications réparties sur architecture J2EE, cette section est consacrée à une analyse critique dudit processus en partant des problèmes constatés concrètement lors de la vie des projets et des éléments méthodologiques présentés.

Un des problèmes essentiels auquel nous sommes confrontés est le passage de l'*analyse* à la *conception*, c'est-à-dire la transformation d'un schéma de compréhension général du processus métier de l'application en un modèle de conception à base de composants. La proposition *Component Unified Process*[136, 137] a pour objectif de résoudre ce problème en introduisant au plus tôt une structuration en composants des différentes fonctionnalités de l'application. Notons que c'est partiellement ce qui est préconisé dans le processus de modélisation que nous avons détaillé ci-dessus. Nous nous intéressons toutefois à un problème différent : il s'agit, rappelons-le, de vérifier que ce qui est développé correspond bien à ce qui est attendu, pas plus, pas moins.

1.2.1 Complexité

Les systèmes d'information en particulier, et les gros systèmes informatiques en général, ont toujours été complexes mais cette complexité conceptuelle se trouve aujourd'hui démultipliée par la complexité des architectures techniques et applicatives mises en place. L'hétérogénéité des systèmes, la répartition des processus, l'omniprésence des réseaux, rendent la tâche du concepteur et du développeur encore plus difficile. Le « paradigme » de programmation orientée-objet n'a rien résolu puisqu'il exige de distribuer non seulement les processus de traitements informatiques mais aussi les processus métiers : le résultat d'une invocation d'une méthode d'un objet métier peut dépendre de l'invocation d'une dizaine, centaine ou millier d'objets éventuellement répartis sur un réseau.

Les plate-formes de composants n'ont pas mieux réussi à simplifier le problème et ont même plutôt accru la fragilité des applications en les rendant dépendantes de services techniques. Ces derniers sont parfois difficiles à appréhender et toujours délicats à intégrer dans une conception de par la multiplication des fichiers de configurations divers et des contraintes de programmation et de conception. Par exemple, la réalisation d'une application *web* classique sur plate-forme J2EE suppose non seulement d'être capable de programmer en Java mais en plus de maîtriser la syntaxe des fichiers de configuration de Struts, de l'application *Web*, du serveur d'application, des fichiers de description de propriétés, des descripteurs de déploiement ... Les ateliers de développement, s'ils permettent d'alléger certaines tâches, ne résolvent pas tout.

1.2.2 Cohérence des éléments du développement

Le processus de développement, considéré à partir de l'expression des besoins, n'est absolument pas incrémental ni itératif mais est bien plus proche d'un modèle en cascade traditionnel : les phases d'analyse, de conception et de développement s'enchaînent dans un ordre descendant mais sans que l'information ne

puisse « remonter la cascade ». Au final, il est fréquent que les documents d'analyse et de conception soient obsolètes lors de la livraison. Au mieux, ils sont écrits *a posteriori* en fonction du code effectivement livré.

Ce problème provient de l'absence de lien direct et surtout automatisé avec la réalisation concrète du logiciel, autrement dit du découplage entre les modèles et le code. Même dans les domaines où le code est généré à partir de modèles, l'information est à sens unique et il est difficile d'intégrer à la fois les développements manuels et la production automatisée d'une partie du code.

Enfin, nombre de développements sont des *redéveloppements*. On suppose donc que le travail d'analyse a déjà été fait et on en fait l'économie. Malheureusement, les processus évoluent aussi et sont rarement indépendants des technologies qui permettent de les mettre en œuvre. Les documents sur lesquels sont basés des redéveloppements sont donc souvent obsolètes, incomplets voire faux, d'où l'impossibilité de valider les logiciels produits *a priori* et de manière mécanique.

1.2.3 Interpénétration des domaines

Les fonctionnalités de l'application se trouvent souvent mélangées à des éléments réglant la navigation et l'accès de l'utilisateur à ces différentes fonctions. C'est particulièrement le cas dans les éléments d'IHM, qui mélangent niveaux de validation et règles de navigation. Il nous semble qu'il s'agit de problèmes strictement orthogonaux :

- l'application offre la possibilité à l'utilisateur de réaliser certaines opérations, ce sont les services offerts. Leur enchaînement, leur structure, leurs besoins, l'algorithmique sont détaillés par les différents documents décrivant les processus et le métier. En particulier, les enchaînements de processus nécessaires ou possibles sont modélisés sous la forme de diagrammes d'activités ou automates d'états finis ;
- l'IHM est une *traduction* à l'usage des opérateurs humains interagissant avec le système du comportement attendu du système. En particulier, les enchaînements imposés et les actions activées et inactivées du système en fonction de l'état de l'activité se trouvent réalisés graphiquement par des modifications d'écran et d'attributs des éléments composant l'IHM.

On retrouve le même problème dans la couche composants/persistance. Le problème de la persistance des objets métiers se trouve traité au même niveau que celui de leur définition et de leur manipulation. Des objets purement mécaniques tels que les *objets données* se trouvent présents dans la modélisation alors même qu'ils sont automatiquement générés par des règles de projection. Enfin, la structure des tables relationnelles se trouve reproduite dans le modèle qui *de facto* perd ainsi sa structure objet.

1.2.4 Formalisation

Au sein des phases d'analyse et de conception elles-mêmes, la cohérence entre les modèles et le respect d'un certain nombre de règles générales ne sont pas systématiquement vérifiés. Si les normes de conception et de modélisation (UML, OCL, RUP ...) prévoient effectivement des règles de validation croisées et de respect de cohérences, les outils disponibles ne sont pas toujours capables de vérifier le respect de ces règles. Par ailleurs, l'écriture de règles nécessite une expertise spécifique qui est rarement présente.

Les modèles d'analyse et de conception ne sont donc pas tels quels vérifiables automatiquement. Leur cohérence est assurée par des règles de nommage qui ne sont pas toujours respectées à la lettre, la dynamique et les algorithmes sont documentés en commentaires sous la forme de pseudo-code non formalisé et donc non interprétable, les modèles d'analyse (diagrammes d'activités) ne sont pas suffisamment précis pour permettre de dériver automatiquement des scénarios de tests (quand bien même ceci est prévu dans le processus, voir ci-dessus).

La structure des modèles n'est pas suffisamment congruente avec l'architecture finale, ni liée au code développé, pour permettre de faire des liens et déductions automatiques et générer éventuellement des squelettes de tests unitaires. Ces modèles ne sont pas maintenus d'une étape à une autre, ce qui produit donc des dérives entre phases qui rendent rapidement la tâche de génération automatique impossible. Ce problème est crucial lors de la *réutilisation* de modèles et d'applications existants : si modèles et codes ne sont pas congruents, la conception de la nouvelle application ne peut réutiliser tels quels les anciens modèles. La production de tests unitaires pour les différents composants du système devient plus difficile

car elle nécessite de la part des équipes de développement de se plonger dans les détails de la documentation d'analyse et de conception pour évaluer les fonctionnalités attendues de tel ou tel objet.

Cette absence de formalisation est aussi un problème aux extrémités de la chaîne puisque les imprécisions dans l'expression des besoins et l'analyse des fonctionnalités de l'application rendent très difficile la mise en œuvre de stratégies de tests système et de tests d'intégration efficaces pour améliorer la qualité de l'application avant le passage en recette. Un grand nombre d'erreurs triviales sont ainsi repérées, parfois avec difficulté, lors de cette phase de recette alors qu'elles auraient pu être détectées plus tôt.

1.2.5 Architecture

Le gros point noir du processus de développement, duquel découle une grande partie des autres problèmes, est l'absence d'un modèle global d'architecture de l'application et d'un découpage clair de celle-ci à différents niveaux de détails. Bien sûr, cette architecture existe et le découpage réalisé lors de la réalisation du cahier des charges et/ou de l'urbanisation du système d'information est repris lors des phases du développement.

Mais aucun modèle n'offre une vue d'ensemble synthétique du système, ni la possibilité de visiter les éléments du système de manière hiérarchique ou thématique : par couche ou par domaine fonctionnel. Les différents modèles sont découpés en fonction des phases du processus, et nous l'avons vu, il n'est pas prévu la possibilité de revenir sur des diagrammes plus abstraits en fonction de choix et de découvertes faits à des niveaux plus concrets.

Le résultat net est une extraordinaire complexité des modèles dans les détails desquels l'utilisateur se trouve très rapidement noyé. Cette complexité est mieux maîtrisée dans le code écrit qui se trouve structuré de manière plus évidente et pour lequel les outils de construction offrent une vue synthétique rapide.

Alors même que l'on parle beaucoup de composants, tant pour ce qui concerne les plate-formes que pour la structuration des applications et du processus de développement, on se rend compte que ces composants ne sont pas des concepts utilisés lors de l'analyse et de la conception. Dans les développements, seuls les composants imposés par l'infrastructure technique apparaissent : EJB, Servlets, éventuellement connecteurs JCA pour la couche persistance. Rien dans les modèles d'analyse ou de conception ne permet de déduire une structure compositionnelle de l'application : il n'y a pas de définition explicite de composants et pas de schéma d'architecture représentant les différents composants et leurs interactions.

1.3 Préconisation

Il n'y a bien sûr pas de solution unique pour résoudre cet ensemble de problèmes, et il est même certain qu'il est impossible de les résoudre complètement, mais l'expérience même de Norsys sur d'autres technologies a montré qu'il était possible d'*industrialiser* les développements pour accroître leur qualité et leur productivité. Nous avons déjà signalé les travaux d'E.RENAUX sur le *Processus Unifié de Composants* — CUP — [137] réalisés dans le cadre d'une collaboration avec Norsys.

Le point clé de cette démarche consiste à identifier les composants le plus tôt possible, dès la modélisation des exigences de l'utilisateur au travers des *cas d'utilisations*. Cette identification précoce permet d'assurer lors des phases ultérieures de conception et de développement un suivi continu. Cette approche est articulée autour de la notion de composant logique correspondant à un méta-modèle et concrétisée en quatre vues complémentaires basées sur ce méta-modèle :

1. la vue des cas d'utilisation ;
2. la vue d'interactions ;
3. la vue de conception ;
4. et la vue d'assemblage.

D'autre travaux de recherche sont en cours dans l'entreprise sur les problèmes de la réingénierie d'applications patrimoniales — J.HATTAT — et de la séparation des préoccupations par des *aspects* de conception — D.DIAZ.

De notre côté, nous avons focalisé notre travail sur les deux points suivants :

– la mise en place d'une *architecture à base de composants contractualisés* depuis l'analyse jusqu'à la réalisation ;
– l'amélioration de la qualité des dits composants par la génération automatisée de tests fonctionnels.

1.3.1 Pour une architecture des logiciels

Le meilleur processus de fabrication non informatique auquel on peut comparer la conception et la réalisation d'un système d'information est celui de la conception et de la réalisation d'un bâtiment ou d'un ouvrage d'art. Dans chacun des cas :
– le produit final est unique ;
– il fait appel à la compétence d'une multitude de métiers ;
– il nécessite une vision à la fois générale et détaillée ;
– il est soumis aux mêmes contraintes dans ses rapports avec l'utilisateur, contraintes exprimées au travers d'un cahier des charges ;
– son résultat ne peut-être évalué *in fine* que par l'usage qui en est fait ;
– il peut s'inscrire dans un ensemble pré-existant ou être conçu *ex nihilo* ;
– il peut produire un résultat dont la qualité va de désastreuse — le terminal T3 de l'aéroport Roissy-Charles-de-Gaulle ou la première version d'Amadeus, le système de réservation de la SNCF — à admirable — Taliesin par Frank Lloyd Wright ou le système d'exploitation Unix ;
– il peut être prévu pour durer — la cathédrale de Chartres ou Internet — ou être jetable ;
– les deux marchés représentent à l'échelle mondiale des tailles comparables et considérables — $3500 milliards pour le génie civil, $1322 milliards pour les services des technologies de l'information ;
– enfin, il rentre dans le jugement que l'on peut en faire une part essentielle de critères esthétiques.

De cette analogie, on peut inférer que le concept central au cœur de l'activité de conception et de réalisation de logiciels est celui d'*architecture*. L'architecture est ici comprise non pas uniquement dans sa dimension purement conceptuelle de production de formes, d'agencements de structures et de réalisation de plans, mais aussi dans sa dynamique concrète en tant que point d'articulation entre les différents acteurs d'un processus et leurs contraintes.

Un composant est donc dans cette vision architecturale la matérialisation dans un plan plus large d'un concept et d'un ensemble de contraintes, une partie d'un tout. Il est nécessairement défini par les relations qu'il entretient avec les autres composants du plan. De plus, un composant peut être lui même composé de parties concourant à la réalisation de ses fonctions. Une architecture est donc de manière duale définie comme un agencement de composants dans l'objectif de fournir un ensemble de fonctionnalités. Sans composants, pas d'architecture ; sans architecture, pas de composants.

Nous n'avons pas la prétention de penser que ce point de vue soit original mais il nous paraît essentiel, et c'est tout l'enjeu de ce travail, de réaffirmer son importance et surtout de se donner les moyens de le rendre opérationnel.

Compositionnalité

De toute évidence, les composants doivent pouvoir être composés pour former à leur tour de nouveaux composants, plus gros ou plus généraux. Inversement, un composant doit pouvoir être arbitrairement décomposé en divers constituants détaillant ses fonctionnalités.

Toutes les exigences qu'un composant impose à son environnement doivent être explicitement spécifiées. Les fonctionnalités ainsi que les dépendances d'un composant doivent être exprimées *contractuellement*, en termes des relations qu'il entretient avec son environnement et non pas en termes de la structure interne du composant.

Les relations que les composants entretiennent entre eux sont ainsi toujours exprimées par des *contrats bilatéraux* : un composant ne peut subordonner l'exécution d'un service à la réalisation par un tiers d'un autre service ou d'une obligation d'un autre contrat. De la sorte, tout composant devient substituable à un autre pour autant qu'il remplisse chaque obligation contractuelle déléguée par la substitution.

Ces relations peuvent être décrites soit comme des *connecteurs*, auquel cas la sémantique des relations entre composants est exogène, soit comme un ensemble de propriétés générales du système dans lequel sont plongés les composants.

Formalisation

Le comportement d'un composant et la structure d'une architecture doivent pouvoir être définis formellement, c'est-à-dire associés à une sémantique non ambiguë et susceptible d'être vérifiable, même partiellement, par des moyens mécaniques. De même, une architecture donnée — un agencement de composants — doit pouvoir être validée mécaniquement en fonction de règles de cohérence générales.

Il doit donc être possible d'exprimer des propriétés et de vérifier que ces propriétés sont bien présentes dans l'architecture. Les propriétés exprimables doivent au minimum être les propriétés de *sûreté*, c'est-à-dire une propriété exprimant l'impossibilité de survenue d'un événement.

Enfin, une architecture doit se prêter à toute opération de *raffinement* consistant à appliquer une transformation produisant une nouvelle architecture correcte à un niveau d'abstraction différent de l'architecture de départ.

Abstraction & concrétion

Un composant doit être une *abstraction* indépendante de toute implantation mais susceptible de s'incarner dans la plus grande palette possible de plate-formes techniques. Autrement dit, un composant est un *modèle*. De plus, si un composant est formellement spécifié et indépendant de toute plate-forme, il doit pouvoir être *concrétisé* mécaniquement de sorte que le résultat soit par construction une implantation conforme et exécutable.

De manière symétrique, étant donnés un composant et une réalisation concrète de ce composant, il doit être possible de vérifier mécaniquement que la réalisation concrète est conforme aux propriétés attendues du composant.

Les données, leur structure, leurs propriétés et leurs transformations constituant la majeure partie des fonctionnalités d'un système d'information, leur définition et leur manipulation doivent être prises en compte dans la description des propriétés des composants, ce à un niveau d'abstraction adéquat avec la représentation du *métier* que ces éléments modélisent.

Synthèse

En résumé, nous attendons d'un modèle d'architecture de composants qu'il soit :
- simple à manipuler avec un nombre de concepts restreints ;
- formel ;
- exécutable ;
- testable ;
- compositionnel et hiérarchique.

1.3.2 Test & fiabilité

Pour s'assurer de la validité d'un logiciel par rapport au cahier des charges, on peut appliquer le principe des *méthodes formelles* : partir d'une expression abstraite et non ambiguë des exigences, et réaliser, par des étapes de raffinement et de preuve du maintien des propriétés du niveau précédent, différents modèles de plus en plus détaillés, jusqu'à obtenir un modèle suffisamment précis pour qu'il soit possible de le traduire directement dans un langage d'implantation concret. C'est la stratégie des méthodes telles que B, Z ou VDM, illustrée dans la figure 1.6.

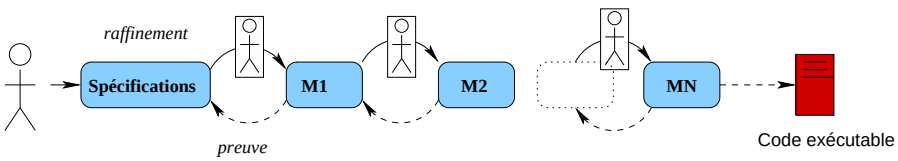


FIG. 1.6 – Développement formel par raffinement.

Cette méthode présente l'avantage évident de produire une application dont la validité eu égard aux spécifications est *prouvée*, sous réserve de la preuve de correction de la projection terminale d'un modèle en code.

Une autre solution est, toujours à partir d'une expression formelle des exigences, de produire automatiquement des *cas de test* qui permettront à l'issue du processus de développement de valider le logiciel produit : c'est ce qu'illustre la figure 1.7.

La première solution est aujourd'hui inaccessible dans le contexte qui est le nôtre, pour un certain nombre de raisons : le niveau de fiabilité exigé dans les applications de systèmes d'informations est nettement moins élevé que dans des systèmes critiques temps réels ; le coût de la mise en œuvre de méthodes formelles est très élevé, surtout compte tenu du fait que la phase de preuve ne peut être totalement automatisée ; l'intérêt de mettre en place un tel processus dans des systèmes hétérogènes en constante évolution paraît limité.

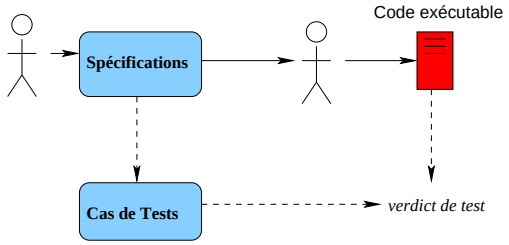


FIG. 1.7 – Développement formel par test.

La seconde solution est plus aisée à mettre en place. Elle exige toutefois un certain nombre de prérequis : l'existence d'une spécification suffisamment formalisée pour permettre la dérivation automatique de cas de tests pertinents ; le maintien d'une cohérence dans le processus de développement permettant de faire en sorte que les cas de tests produits initialement restent applicables à l'autre bout de la chaîne, c'est-à-dire lors de la construction du logiciel fini ; l'existence de procédures automatiques fiables et de mesures de la fiabilité atteinte par l'exécution d'un ensemble donné de tests.

Nous partons donc de l'hypothèse que le processus de développement par raffinement formalisé restera encore longtemps impraticable pour la majeure partie des développements. Il y aura donc encore longtemps production de deux artefacts disjoints : un *modèle* ou spécification des fonctionnalités attendues de l'application, et l'*application* elle-même. Par conséquent, il restera nécessaire de disposer des processus permettant de valider et vérifier avec le maximum de précision possible la conformité entre les spécifications et le résultat final. De la même manière que pour un bâtiment, tant que l'on ne saura pas produire automatiquement un bâtiment à partir des plans de l'architecte, il sera indispensable de disposer d'outils permettant le contrôle du processus de construction : suivi de chantier, bureau de vérification, certificat de conformité, normes, qualifications ...

Chapitre 2

Architecture & composants

Nous avons, dans le chapitre 1, insisté sur le besoin d'une conception formalisée de l'architecture des logiciels. Nous avons aussi conclu que la notion d'architecture était liée à la notion de composants. Ce chapitre sera donc consacré à ce qu'il est convenu d'appeler l'état de l'art dans le domaine de la modélisation, de la spécification et de la réalisation d'architecture de systèmes à base de composants.

Nous partirons des outils et plate-formes orientées-composant concrètes, dans lesquelles un composant est une entité logicielle bien définie, s'insérant dans un modèle d'exécution et de communication lui aussi précis. Le développement sur ces plate-formes — J2EE, Corba, .Net — constituent aujourd'hui une part importante de l'activité des sociétés de services. Nous essaierons de comprendre pourquoi ces plate-formes ont échoué à améliorer la qualité des logiciels.

Ceci nous mènera naturellement à l'étude d'un certain nombre de propositions pour la conception d'architectures. À partir des outils de modélisation proposés autour du langage UML, nous nous intéresserons plus particulièrement aux nombreux travaux autour des *Langages de Description d'Architecture* — ADL en anglais. Sachant qu'il existe déjà des excellentes synthèses sur les ADL les plus anciens, nous nous concentrerons sur quelques propositions plus récentes du domaine.

Enfin, nous aborderons certains modèles plus théoriques qui s'intéressent à la composition de composants possédant une description comportementale formalisée.

2.1 Plate-formes de composants

Une définition très fréquemment reprise du terme de composant est celle proposée dans Szyperski [150] (p.36, traduction par nos soins) :

> « Un composant logiciel est une unité de composition avec des interfaces contractuellement spécifiées et des dépendances uniquement contextuelles. Un composant logiciel peut être déployé de manière indépendante et composé par des éléments tiers. »

Cette définition est axée comme l'ensemble de l'ouvrage cité sur une vision essentiellement *technologique* et économique de la notion de composants : le problème considéré est celui de la construction de logiciels à partir de composants réutilisables et du développement subséquent d'un *marché* de composants.

Cette première partie présente les outils répondant à cette définition : les *plate-formes à composants* ou *intergiciels à composants*. Trois acteurs de taille inégale se partagent aujourd'hui ce marché : Java et J2EE qui en possèdent la plus grand part, .Net qui est la réponse de MICROSOFT et qui prend de l'ampleur, et Corba et le *Corba Component Model* — ou CCM — qui est conceptuellement la meilleure proposition des trois mais qui reste marginale. Nous inclurons dans cette catégorie Fractal qui se présente comme une plate-forme de composants, bien qu'elle ne soit pas encore présente dans le domaine des services logiciels.

La seconde partie de cette section s'intéressera à des technologies plus récentes et moins ambitieuses qui visent à pallier à la grande complexité de mise en œuvre des intergiciels. Ces technologies sont généralement inspirées des *langages de configuration* et offrent essentiellement un *cadre méthodologique* basé sur la notion de composants.

2.1.1 Intergiciels à composants

Les intergiciels sont nés de la volonté de résoudre le problème de la *distribution* des applications sur un ensemble de systèmes inter connectés par un réseau. L'objectif initial et à ce jour non atteint était de rendre *transparente* pour le développeur et l'utilisateur la structure répartie des applications. Pour le développeur, il s'agissait par ailleurs de faciliter le développement, le déploiement et la maintenance de telles applications. Les intergiciels que nous étudions ici ne constituent qu'une réponse parmi d'autres possibles à ce problème et l'on trouvera dans Tannenbaum et van Steen [151] une introduction complète et accessible sur les autres solutions techniques.

Tous les intergiciels à composants s'appuient sur un certain nombre de concepts communs :

- la notion de *conteneur* qui assure la séparation des préoccupations entre les aspects techniques des logiciels répartis et les aspects fonctionnels ;
- l'adressage symbolique médiatisé par un système d'annuaire dynamique ou *serveur de nommage* qui minimise le couplage entre les entités d'un système ;
- le *serveur d'application* qui orchestre l'exécution effective des applications.

J2EE

J2EE, pour *Java 2 Enterprise Edition* est une extension de l'API Java spécifiquement destinée à la réalisation de *systèmes d'informations d'entreprises*. Les développements visés sont des systèmes autonomes ou plus souvent des sous-systèmes collaborant avec d'autres sous-systèmes tels que des bases de données, des applications patrimoniales client-serveur ou en sites centraux, des ERP. J2EE — actuellement dans sa version 1.4 — est à la fois une norme pour les concepteurs de plate-formes et d'outils et un ensemble d'API pour les développeurs d'applications. Le site *Web* de SUN est bien entendu la référence sur cette plate-forme[148, 149].

Caractéristiques Nous ne rentrerons pas dans le détail de l'architecture de la plate-forme J2EE et nous nous contenterons d'en souligner les principales caractéristiques.

Le support d'exécution d'une application J2EE est le *serveur d'application* destiné à lier les différents composants de l'application entre eux. Différents types de composants sont supportés par J2EE : les composants *Web* tels que *servlets* et *Java Server Pages*, les composants EJB — *Enterprise JavaBeans* — ou composants métiers, les *connecteurs* vers des systèmes externes, les applications autonomes.

L'unité de déploiement du composant est l'*archive* accompagnée de son *descripteur de déploiement*. Elle peut en théorie être déployée sur n'importe quel serveur d'application conforme à la spécification J2EE. Une application constituée d'un ensemble de composants peut être empaquetée et déployée globalement.

Le concept de *conteneur* permet aux composants de s'abstraire des détails techniques de la gestion du contexte d'exécution et offre un point d'accès à divers services normalisés : l'invocation de méthodes distantes ou RPC, le *service de nommage*, le *service de gestion des transactions*, la *gestion de la persistance*, la *messagerie asynchrone*, la gestion du *cycle de vie* des composants en fonction de leur nature. L'accès à ces service se fait soit au travers d'une API offerte par le conteneur, soit par déclaration dans un descripteur *ad hoc*.

Les EJB constituent les composants métiers d'une application J2EE : ce sont eux qui contiennent les traitements à réaliser et qui assurent l'interface entre les données persistantes et la logique de présentation de l'application (voir chapitre 1, section 1.1.2). Les composants EJB sont répartis en trois catégories :

- les *EJB Session* n'ont pas d'identité propre au-delà d'une invocation ou d'une séquence d'invocations de méthodes ;
- les *EJB Entité* ont une identité persistante : ils représentent des informations métiers stockées dans un SGBD ;
- les *EJB Message* réalisent des traitements asynchrones déclenchés à partir de messages placés dans des files.

Critiques Les critiques de la section 1.2 ne sont pas toutes dues au processus de développement. Une partie non négligeable du problème a sa source dans la complexité de la plate-forme utilisée, en l'occurrence

J2EE. Cette plate-forme n'est pas en effet réellement une plate-forme à base de composants mais plutôt une solution technique orientée-objet pour les systèmes répartis. Cette solution n'est pas nécessairement mauvaise, mais appuyer une conception sur une telle structure ne peut que noyer le modèle architectural dans trop de détails. Et les outils existants ne parviennent pas à masquer suffisamment cette complexité pour permettre de s'en abstraire.

.Net

La plate-forme .Net est la réponse de MICROSOFT au développement des applications Java dans les systèmes d'information d'entreprise. Elle constitue une refonte de l'architecture COM/DCOM/ActiveX existant depuis de nombreuses années sur le système Windows et qui avait déjà pour objectif de faciliter la communication entre applications, locales puis distantes, et le développement d'un marché de composants réutilisables essentiellement dans le domaine des interfaces graphiques.

Nous nous basons pour cette étude sur Szyperski [150], chapitre 15, consacré à la vision de MICROSOFT de la notion de composants et sur Lantin [83] qui est une synthèse des techniques de programmation sur plate-forme .Net. Rappelons par ailleurs que les travaux sur AsmL[21, 68] ont pour cible la formalisation de composants et d'architectures .Net.

Les caractéristiques principales de .Net sont les suivantes :

- une infrastructure de langage commune — *Common Language Infrastructure*, CLI — dont la spécification est publique et qui comprend :
  - la définition d'un langage intermédiaire indépendant des langages de programmation de haut-niveau (CIL),
  - un système de types suffisamment riche pour supporter la plupart des concepts objets,
  - une spécification des assemblages, applications ou composants de services,
  - et un ensemble de méta-informations accessibles à l'exécution et qui permet en particulier de résoudre le problème des *versions* d'interfaces, problème récurrent sous Windows plus connu sous le nom d'« enfer des DLL ».

  Cette infrastructure est similaire au JDK dans le monde Java et cette similarité va jusqu'à l'implantation dans le *Common Language Runtime*, la plate-forme concrète de Microsoft implantant le CIL, d'un compilateur *Just-In-Time* afin que les applications puissent s'exécuter à la vitesse du code natif ;
- les *assemblages* qui sont l'unité de base de déploiement et qui possèdent la propriété de disposer d'un nom symbolique globalement unique. Les assemblages peuvent définir des dépendances explicites et des interfaces offertes, autrement dit ce sont d'authentiques composants. Le CLR et le système d'exploitation sont responsables de la réalisation effective des assemblages et donc de la création des composants et de la résolution de leurs dépendances ;
- l'intégration dans un *cadriciel* de composants de services standards permettant d'accéder à l'ensemble de l'API Windows ;
- l'invocation distante de méthodes soit au travers des mécanismes COM/DCOM/COM+, soit basée sur les *Web Services*. Dans le premier cas, les applications bénéficient de l'ensemble de l'infrastructure développée au fil des ans pour assurer l'interopérabilité des applications et des machines. Il s'agit notamment du service de nommage ou base de registres de Windows qui est désormais répartie, du service de persistance, de la gestion du contexte transactionnel et de la synchronisation des processus.

La diffusion de cette plate-forme s'est accompagnée de la promotion par Microsoft d'un nouveau langage orienté-objet nommé C# particulièrement adapté à la compilation vers le CIL et qui reprend en les améliorant nombre d'éléments de ses précurseurs Java et C++. Il est intéressant de constater que le langage Java s'est lui-même récemment enrichi de certains traits apparus dans .Net : annotations d'éléments du langage, types génériques, traitement uniforme des types primitifs et objets.

Bien entendu, les développeurs ont à leur disposition, .Netisée, toute l'infrastructure colossale de Windows et des tâches fastidieuses sur d'autres plate-formes comme la construction d'interfaces graphiques — *Windows Forms* — pour clients légers ou l'accès à une source de données — ODBC et ADO — sont tout à fait triviales dans le monde .Net.

Arrivé à maturité plus tard, puisque c'est seulement au moment où nous écrivons ces lignes que commencent à éclore les développements de projets d'applications d'entreprises sur cette plate-forme, .Net a bénéficié d'une part de l'expérience de ses précurseurs et bien évidemment de Java, et d'autre part de la

capacité de MICROSOFT à mettre en œuvre les moyens nécessaires pour offrir dès la sortie de la plate-forme l'ensemble de l'infrastructure et l'atelier de développement adéquat, ce qui en fait aujourd'hui un outil de choix pour les futurs développements.

Corba Component Model

Le *Corba Component Model*[122] s'appuie et fait partie de CORBA 3.0[123]. Il est composé d'un ensemble de modèles permettant de décrire différents aspects d'un système de composants. Ces modèles sont ensuite traduits par différents outils, soit au moment de la compilation et de la construction de l'application, soit au moment de son assemblage et de son déploiement, pour produire un ensemble d'objets et d'interfaces CORBA qui sont accessibles au travers de l'ORB par n'importe quelle application ou composant utilisant un ORB compatible.

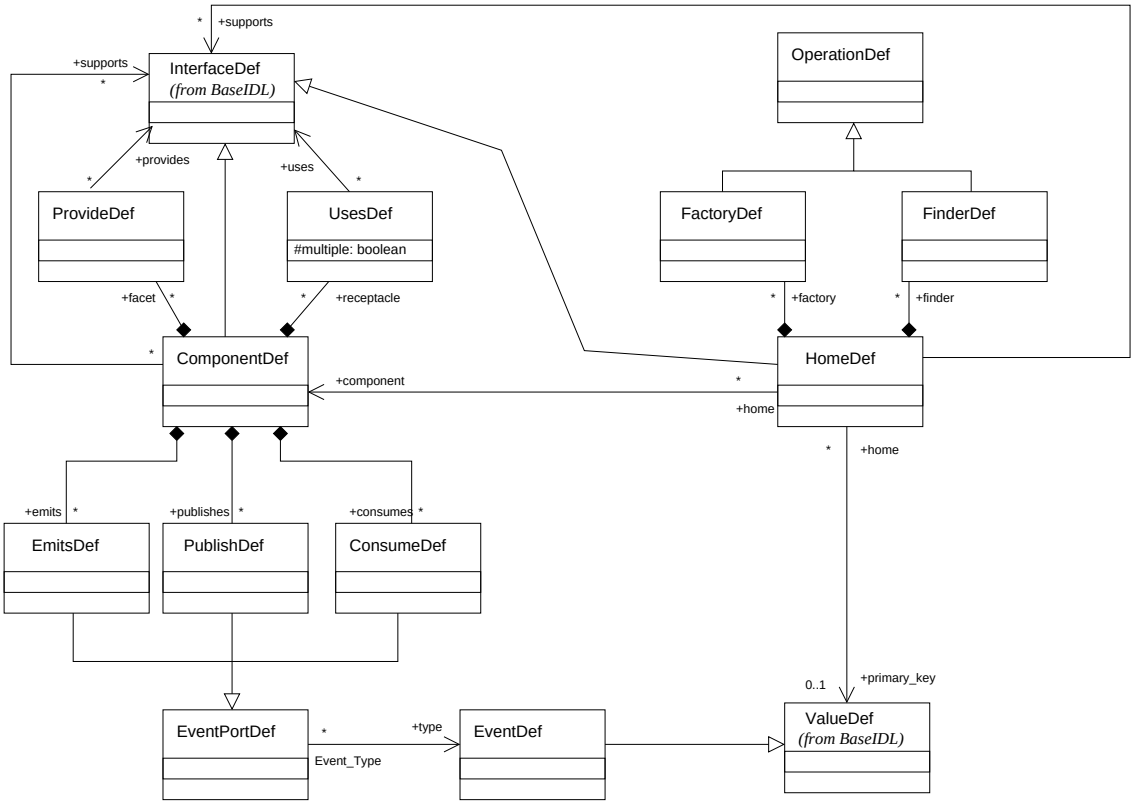


FIG. 2.1 – Corba Component Model (fragment).

Le modèle abstrait de composants Ce modèle est pour l'essentiel la définition du langage IDL3, un sur-ensemble du langage de description d'interfaces de CORBA qui ajoute un certain nombre de mots clés permettant de définir la structure logique des composants. Un composant est ainsi défini comme un ensemble d'interfaces fournies et requises, d'événements consommés et produits, et de propriétés.

À chaque composant est associée une *fabrique* de composants ou *home* qui est une interface décrivant les modalités de création et, pour le cas des composants de type entités, de recherche d'instances de composants. Ces déclarations sont projetées en IDL2 sous la forme d'interfaces CORBA pour pouvoir être accessibles au travers de l'ORB. La figure 2.1 est un diagramme UML représentant les éléments principaux du CCM.

Le modèle de programmation Il comprend la définition du langage CIDL — *Component Implementation Definition Language* —, permettant de définir l'implantation d'un composant, et de la plate-forme

d'exécution des composants (les *containers*). Quatre types de composants sont définis :

- les composants **service** ne maintiennent aucun état entre deux appels de méthodes et ne possèdent pas d'identité propre. Ils sont l'équivalent des *EJB Session* sans état ;
- les composants **session** maintiennent un état pour la durée d'une *conversation* avec un client, une succession d'invocations de méthodes. Lorsque la session est terminée — le plus souvent sur décision du client, le composant perd son identité. Ils sont l'équivalent des *EJB Session* avec état ;
- les composants **processus** persistent entre deux invocations mais n'ont pas d'identité. Ils n'ont pas réellement d'équivalent sur J2EE ;
- les composants **entité**, enfin, qui ont un état persistant et qui de plus possèdent une identité propre au travers d'une clé primaire.

À chaque type de composant est associé un type de *container* qui se charge de la gestion des aspects non fonctionnels — persistance, contexte transactionnel, sécurité, cycle de vie — du composant et du routage des invocations effectuées par les clients. Les interfaces entre composants et containers sont générées automatiquement lors de la phase de compilation de la description CIDL. Un composant peut être réalisé par plusieurs objets au sens CORBA, les *exécuteurs*, qui auront chacun la charge d'une partie des interfaces du composant, le conteneur orchestrant la création des différentes instances d'objet et leur assemblage.

Le modèle de déploiement Le déploiement d'un ou plusieurs composants est décrit par un fichier XML qui définit la manière d'utiliser un composant ou un ensemble de composants en terme d'architecture logicielle et de contraintes systèmes. Ce *descripteur* est associé au code du composant dans un fichier de déploiement — une archive `.jar`, par exemple — qui est alors utilisable pour le développement d'une application complète. Le logiciel de déploiement utilise ces informations pour dialoguer avec des serveurs d'assemblage et d'installation afin de mettre en œuvre les composants.

Critiques Le CCM reprend dans son modèle abstrait les concepts principaux des ADL : composants à multiples interfaces, connecteurs, fabriques de composants. Comme pour les autres plate-formes, le conteneur est l'élément clé qui isole le composant des détails des services techniques. Cette plate-forme est à notre avis la plus proche d'une architecture à base de composants et nous nous en sommes d'ailleurs fortement inspirés pour définir notre modèle de composant (voir chapitre 4). La réalisation concrète de logiciels basés sur le CCM reste encore une tâche ardue, du fait du manque d'outils permettant de s'abstraire des contingences de l'ORB.

2.1.2 Containers de composants

Les principaux intergiciels orientés composants que nous avons décrits succinctement ci-dessus sont des plate-formes complètes, complexes et globalement difficiles à maîtriser de par l'ampleur des domaines qu'elles essayent de recouvrir. La principale difficulté que l'on rencontre dans la mise en œuvre concrète de ces plate-formes est la quasi-impossibilité dans laquelle se trouvent le concepteur et le développeur de s'abstraire des contraintes techniques induites par les supports d'exécution et les outils.

Dans le cas de CORBA/CCM, le développeur se trouve confronté à un modèle satisfaisant, suffisamment riche et conceptuellement clair, mais dont les bénéfices en termes de conception se trouvent quasiment anéantis par la difficulté de mise en œuvre technique, quelles que soient les plate-formes. Dans le cas de J2EE, le problème de la complexité technique reste entier encore qu'un peu allégé par rapport à CORBA, mais par contre le modèle de composants n'est pas satisfaisant et trop pauvre pour supporter réellement une conception orientée composants. Enfin, à l'exception de .Net, aucune de ces plate-formes ne supporte la notion pourtant essentielle de *composite*.

Récemment sont apparus un certain nombre de *cadriciels* dont l'objectif, plus ou moins inspiré par les travaux sur l'*Ingénierie Dirigée par les Modèles — Model Driven Engineering* — et les ADL, est de permettre d'une part de construire réellement l'application comme un assemblage potentiellement hiérarchique de composants aux dépendances explicites, d'autre part de ne faire payer aux développeurs que ce qu'ils utilisent effectivement de l'infrastructure technique.

Spring, Kilim & PicoContainer

Ces trois *conteneurs de composants*, réalisés initialement en Java mais depuis partiellement portés sur .Net, partagent une même approche du problème qui peut se résumer par la mise en œuvre du patron de conception « *injection de dépendances* », bien qu'ils soient très différents dans leurs ambitions. La notion d'*injection de dépendances* est un terme inventé par Martin Fowler pour décrire un patron de conception orienté-objet dans lequel les dépendances entre objets sont remplacées par des dépendances vers des abstractions, ce qui dans le cas du langage Java se traduit par le typage des références par des interfaces.

Le plus léger des trois, Picocontainer se veut un cadriciel extrêmement simple permettant de construire des applications conçues comme des assemblages de composants à partir d'objets standards en Java (POJO). Il offre différentes implantations de conteneurs et une API minimaliste permettant d'assembler dynamiquement des instances d'objets, les *composants*, par découverte de leurs dépendances et de leurs interfaces, découverte rendue possible par l'utilisation des capacités réflexives des langages d'implantation.

Kilim est basé sur le même principe mais ajoute la possibilité de décrire les assemblages et la configuration des composants dans un descripteur externe qui sera chargé par le moteur au lancement de l'application.

Spring enfin, est le plus complet de ces cadriciels et se veut une infrastructure transversale destinée à faciliter le développement d'applications J2EE en implantant un modèle de composants. Comme pour les précédents projets cités, Spring utilise l'*injection de dépendances* pour réaliser au moment de son déploiement des assemblages de composants, composant étant ici comme précédemment synonyme d'objet.

Fractal

Fractal est un modèle de composant découplé de toute implantation concrète et qui présente un certain nombre de caractéristiques originales. Le modèle de base détaillé dans E.Bruneton *et al.* [55] se présente comme un ensemble d'interfaces définissant les exigences que doit remplir toute implantation du modèle. Un composant est ici composé d'un *contrôleur* ou *membrane* et de sous-composants, éventuellement *partagés* entre différents composites. Les composants interagissent au moyen d'*interfaces* qui peuvent être *externes* ou *internes* — accessibles uniquement aux composants encapsulés. Des interfaces de contrôle génériques sont définies permettant d'offrir des mécanismes d'introspection, de liaison dynamique et de gestion du cycle de vie.

Les caractéristiques les plus originales du modèle sont :

- la séparation d'un composant entre son corps et sa *membrane* qui permet de définir de véritables comportements pour l'interface d'un composant ou d'une architecture de composants, sans préjuger de son implantation. On peut ainsi définir des *membranes* gérant la sécurité, des membranes filtrant ou retraitant les messages, des membranes possédant tel ou tel service ;
- la prise en compte explicite des *composites* comme des composants à part entière. Un composant peut éventuellement offrir des interfaces d'administration sur sa structure, ou la laisser totalement opaque ;
- le *partage* des composants entre différents assemblages.

Ce modèle possède une implantation de référence en Java dénommée Julia et a servi aussi à la réalisation de composants permettant de construire une sorte d'*OS en kit*. Il est au centre de plusieurs travaux de recherche et possède une communauté de développement relativement active au travers du consortium ObjectWeb.

Le modèle Fractal est intéressant ne serait-ce que par sa relative économie de moyens lorsqu'on le compare aux plate-formes classiques. À notre avis, seule toutefois la définition explicite de composites est réellement un atout : la notion de membrane, intellectuellement séduisante n'est qu'une reformulation des connecteurs et des *conteneurs ouverts* et n'est pas un concept primitif d'architecture car il est toujours possible de la remplacer par des familles de composants spécifiques. Le partage de composants, quant à lui, pose des problèmes de formalisation des dépendances, de contractualisation des interactions et d'encapsulation des assemblages.

2.2 Langages de description & conception d'architectures

La précédente section était dédiée aux plate-formes et outils pour l'exécution d'architecture de composants. Nous remontons d'un cran dans l'abstraction en examinant quelques propositions permettant de concevoir des architectures de composants. La première partie est consacrée aux langages de conceptions généralistes. Leur caractéristique commune est de couvrir un spectre très large de situations et par conséquent d'avoir une sémantique relativement faible pour la modélisation du comportement des composants. La deuxième partie s'intéresse plus particulièrement aux ADL spécialement conçus pour représenter des architectures et des composants.

2.2.1 UML

UML est aujourd'hui l'outil de base pour la conception et l'analyse des applications, grâce à la simplicité de ses concepts de base et à la profusion d'outils existants pour manipuler des modèles. Le point fort d'UML, sa versatilité, est aussi son point faible : l'actuelle norme 2.0 est encore en phase d'adoption au sein du consortium OMG et l'intégration de la multitude de diagrammes disponibles ainsi que le flou entourant leur sémantique rendent l'implantation d'un processus réellement dirigé par les modèles encore très difficile.

La notion de *composant* est présente dans les versions précédentes du langage uniquement comme représentation d'une entité concrète du système déployée sur une architecture physique. Cette notion a été étendue à celle d'unité de composition architecturale avec l'introduction des *diagrammes de composants*. Ces diagrammes, dont nous donnons un exemple tiré de la spécification[127] dans la figure 2.2, permettent désormais de représenter explicitement des architectures de composants.

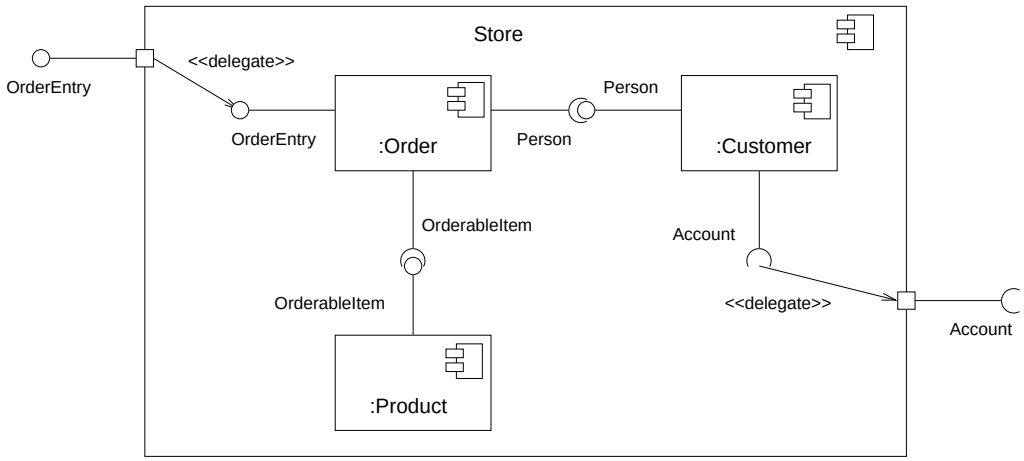


FIG. 2.2 – Exemple de diagramme de composants UML 2.0.

De plus, il est désormais possible d'attacher aux ports des composants des spécifications de *protocole* sous la forme de *State Machines* ce qui permet d'utiliser UML comme un ADL à part entière.

Ces diagrammes ont le mérite d'être simples dans leurs principes et de ne pas surcharger le langage. Les composants peuvent être détaillés au moyen de la syntaxe UML existante : diagrammes de classes, diagrammes d'états-transitions. La notion de connexion reste toutefois confondue avec celle de dépendance et seuls les ports de type synchrone sont pris en compte explicitement.

EDOC

EDOC — *Enterprise Distribued Object Computing* — est un profil UML — une spécialisation du langage et de son méta-modèle — pour représenter des concepts spécifiques à un domaine d'application, une technologie ou un processus. C'est une norme adoptée par l'OMG en 2001. EDOC est une alternative

intéressante car plus complète aux *diagrammes de composants* pour la modélisation d'architectures de composants.

Le cœur de EDOC est constitué de l'*Architecture de Collaboration de Composants* qui définit les concepts d'architecture, d'assemblage, de composants et d'interactions en termes de modèles UML. L'objectif affiché de ce modèle est de répondre à différents problèmes :

- la composition récursive de composants pour modéliser des systèmes à différents niveaux d'abstraction ;
- la *traçabilité* de l'évolution des modèles et des implantations, l'automatisation du processus du développement au travers d'une démarche MDA ;
- le couplage faible entre éléments d'un système pour promouvoir la réutilisation et l'évolution concurrente de différentes parties d'un système ;
- l'indépendance envers les technologies ;
- et enfin l'émergence d'un marché des composants métiers.

La figure 2.3 représente un composite dans la notation EDOC ou *processus communautaire* modélisant un système d'achat-vente. Ce schéma détaille différents types de ports et en particulier des *multiports* ou protocoles.

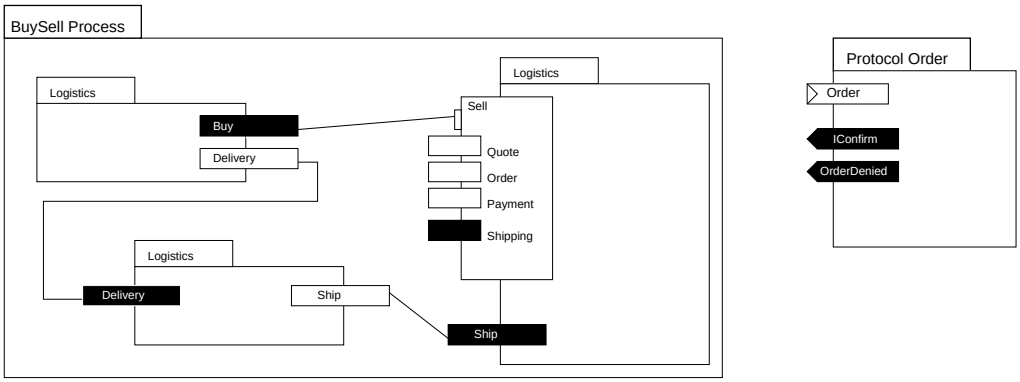


FIG. 2.3 – Exemple de schéma de composite EDOC.

Les ports d'un composant sont :

- soit des émetteurs ou récepteurs d'événements atomiques ;
- soit des ports définissant un *protocole d'interaction* dont le composant est l'initiateur ou le répondeur, décrit au moyen d'un diagramme d'activité — une *chorégraphie* dans la terminologie officielle — et subdivisés eux-mêmes en ports de granularité plus fine.

Si le modèle de communication est au niveau le plus fin un modèle *asynchrone* basé sur des flots d'événements, les protocoles permettent d'imbriquer des séquences d'événements et de protocoles pour constituer une *transaction* plus globale. Une *opération* au sens UML du terme est vue comme un ensemble de flots liés par une sémantique d'appel-retour.

Ce modèle de base est étendu en un *modèle de processus métiers — business process model —* qui permet de définir des processus concurrents communiquant par flots de données en tant que composants.

Comme à l'accoutumée, le document de normalisation est très précis et détaillé, décrivant chacun des éléments du profil, la notation associée, les contraintes structurelles et la sémantique de chacune des constructions. Cette proposition nous a paru très intéressante, ne serait-ce que par ses objectifs qui recouvrent une partie de nos préoccupations. Il manque toutefois une formalisation explicite et compacte de la sémantique des systèmes et architectures considérés.

AADL

Le *langage de conception et d'analyse d'architecture*[57] — *Architecture Analysis & Design Language* en anglais — est une proposition de standardisation de l'ingénierie dirigée par les modèles et l'architecture

pour les systèmes embarqués critiques développée par la *Society for Automotive Engineers*. Cette proposition s'appuie sur les travaux de normalisation de l'OMG dans le cadre de UML 2.0 et les travaux antérieurs sur des outils de méta-modélisation tels que MetaH.

AADL se présente essentiellement comme un socle commun de concepts sur lequel pourront s'appuyer des outils de développement et d'échange de modèles. Ce langage est plus particulièrement destiné à la modélisation de systèmes embarqués mêlant matériel et logiciel. Il prend en compte la modélisation des plate-formes d'exécution au travers de différents composants d'abstraction — processeur, bus, périphérique, mémoire — et la conception de composants logiciels parmi lesquels on distingue des composants actifs — *threads* et processus — et des composants passifs — paquetages et données. À chaque composant peuvent être attachées des *propriétés* prédéfinies par le langage AADL ou spécifiques au domaine, et des contraintes sur ces propriétés qui permettent de modéliser et vérifier des exigences *temps-réels*.

Les composants interagissent au travers de ports qui classiquement isolent un composant des autres éléments du système, ports qui supportent trois modes de communication : le flot de données typé avec une sémantique d'interruption — au sens d'interruption dans les systèmes d'exploitation — ou de files, l'appel de procédures et le partage de variables.

Des outils existent, en particulier pour la plate-forme Eclipse, mais pour l'instant ils se limitent à la manipulation de modèles au travers de diverses interfaces et des vérifications de consistance interne des modèles. AADL est essentiellement une norme d'échange de modèles, dont les forces affichées sont la capacité à intégrer des contraintes spécifiques aux domaines et les points d'extension. Ce langage est en quelque sorte le pendant pour le temps-réel de EDOC.

2.2.2 Les ADL

Medvidovic et Taylor [105] propose une taxonomie des principaux ADL selon plusieurs axes. Cette étude assez large identifie les concepts du domaine et la manière dont chaque langage les définit et permet de les manipuler. Nous avons choisi de nous intéresser dans les paragraphes qui suivent à deux ADL non présentés dans cette étude et qui nous ont paru posséder des caractéristiques proches de nos préoccupations : SOFA et ArchJava. Toutefois, nous introduisons la discussion avec une analyse de deux ADL « classiques » Wright et Rapide, parmi les premiers ADL à supporter une sémantique formelle.

Wright & Rapide

Wright[10] est à l'origine avec UniCon[145] de l'introduction de *connecteurs* formellement spécifiés entre composants et donc de la définition de protocoles de communication entre composants d'un système. Les composants et connecteurs sont spécifiés par des expressions CSP[40] avec la sémantique en termes de traces et de refus de ce langage. Un connecteur est vu comme l'exécution en parallèle de plusieurs processus décrivant les *rôles* dans lesquels le connecteur peut-être utilisé et un processus *colle* décrivant les interactions entre rôles. Un composant est défini aussi par des expressions CSP décrivant ses différents *ports* et son comportement.

La compatibilité entre rôles — d'un connecteur — et port — d'un composant — et donc la correction d'une architecture donnée, est assurée par la vérification d'une relation de *raffinement* étendue entre processus. Plus précisément, un port est compatible avec un rôle si l'intersection des traces du port et des traces déterministes du rôle est un raffinement de l'ensemble de traces du rôle. Une propriété importante que l'on peut vérifier est l'absence d'interblocage dans une configuration architecturale donnée.

La motivation principale de Wright pour la modélisation explicite des connecteurs est d'exprimer un nombre varié de politique de communication. Elle permet d'appliquer le formalisme sur une plus large gamme de systèmes en découplant la description des interactions de la description des fonctionnalités des composants. Les premières sont considérées généralement comme peu variables pour une famille de systèmes donnée tandis que les seconds sont *a contrario* très variables et dépendantes des fonctions précises du système. On notera, et c'est un point aussi souligné par les auteurs que l'on peut parvenir au même résultat en utilisant uniquement des composants et une seule modalité de communication.

Rapide[91] est un environnement complet de spécification et de développement de systèmes distribués orientés-objets guidé par l'architecture. Les composants sont des objets concurrents, hiérarchiques, typés

par interfaces et implantés par des modules. Ils communiquent par des connexions explicites en émettant des *actions* et invoquant des *fonctions*. L'ensemble des événements d'un système forme un ordre partiel d'événements qui constitue la base de la sémantique d'exécution.

Les *interfaces* définissent des signatures d'actions et de fonctions requises et fournies et éventuellement un *contrat* sous la forme de contraintes comportementales, sur le contenu des messages échangés au travers de l'interface, et sur la causalité entre événements reçus et émis par l'interface. Une *architecture* définit l'implantation d'une interface en termes d'interfaces imbriquées et de leurs connexions. Les *connexions* définissent des relations de causalité entre ordres partiels d'événements soit entre événements externes, soit entre événements externes et internes. La notion de service permet de regrouper un ensemble d'événements dans un type et de connecter directement des services complémentaires. Enfin, la notion de *mapping* permet de définir des transformations d'un ensemble d'événements dans un autre, offrant ainsi la possibilité de décrire des relations entre différents niveaux d'abstractions. Les propriétés que l'on peut vérifier sont nombreuses : la conformité d'une architecture par rapport à son interface, d'une implantation par rapport à son architecture, des propriétés sur les séquences d'événements ...

ArchJava

ArchJava[7, 8] est une extension au langage Java similaire à un ADL permettant d'intégrer des contraintes architecturales dans du code source. Les extensions au langage permettent de vérifier des propriétés d'*intégrité des communications* : les objets communiquent entre eux uniquement par les ports spécifiés. Ces propriétés sont vérifiées à la compilation par une extension du système de type de Java intégrant les notions de ports, de composants et de composites. On vérifie ainsi que des composants ne peuvent communiquer qu'avec des membres d'un même composite, des sous-composants ou le composant englobant.

Un *composant* est une instance d'une classe `component` contenant des définitions de *ports*. Un port est un ensemble de méthodes qui peuvent être *fournies*, *requises* ou *diffusées* (*broadcast*). Les méthodes fournies doivent être implantées par le composant, les méthodes requises étant fournies par l'autre extrémité de la connexion.

Un port peut être défini au moyen d'un type *interface de port* contenant uniquement des signatures de méthodes fournies, offrant ainsi la possibilité de définir dynamiquement le composant implantant réellement le port.

Un *composite* est un composant contenant d'autres composants et qui décrit des connexions ou des schémas de connexion autorisés entre ses sous-composants, et éventuellement avec ses propres ports. Un composite dispose de la capacité d'invoquer directement les méthodes de ses sous-composants, l'inverse n'étant pas vrai.

Dans Aldrich *et al.* [8], un modèle formel du langage ArchJava est défini essentiellement au travers d'un système de types et d'une sémantique opérationnelle. Ce modèle est basé sur FeatherweightJava [74], une formalisation du langage Java usuel. On vérifie à l'aide de ce modèle que l'intégrité des communications est bien maintenue par le typage et certaines instructions à l'exécution (vérification du transtypage) :

- un composant C ne peut appeler directement des méthodes d'un composant C' différent que si C' est un sous-composant encapsulé dans C ;
- dans tous les autres cas, la communication entre composants se fait au travers de leurs ports.

Le principal intérêt du modèle ArchJava est sa proximité avec le langage Java ce qui permet d'intégrer facilement des concepts architecturaux dans les développements, la correction étant assurée par le système de types. C'est aussi son principal inconvénient qui rend ce modèle trop lié à une implantation précise — en l'occurrence un langage et une plate-forme d'exécution — et surtout qui induit un glissement des problématiques architecturales de la conception vers la réalisation ce qui n'est pas l'objectif souhaité. De plus, les concepts de composants se télescopent avec les concepts traditionnels de l'objet ce qui peut rendre l'utilisation de ce système à grande échelle très délicate. Enfin, parce qu'il étend le langage, son utilisation nécessite une modification de la chaîne de production d'applications et de composants pour inclure une transformation du code source.

SOFA

Le modèle SOFA — pour *SOFtware Appliances* — est détaillé pour l'essentiel dans Plasil et Visnovsky [133] et fait partie d'un projet plus large d'une plate-forme de conception de composants, par ailleurs intégré dans le consortium *Objectweb*. L'objectif du modèle, qui emprunte la majeure partie de ses concepts aux ADL, est de permettre la *validation statique* de systèmes de composants hiérarchiques, la validité du raffinement d'architecture dans l'étape de conception, et la vérification dynamique au travers de l'embarquement d'assertions dérivées du langage de spécification à l'exécution.

Le concept de base de SOFA est la notion de *protocole* définissant un langage sur un alphabet composé de messages de type requête-réponse. Ce langage est rationnel ou approché par un rationnel. Un protocole peut être attaché à une *connexion*, qui est un lien bidirectionnel entre deux entités. Ou il peut être attaché à un *agent*, c'est-à-dire une entité utilisant une ou plusieurs connexions pour communiquer avec son environnement. L'alphabet d'un protocole est partitionné entre un ensemble de messages fournis et un ensemble de messages requis.

Un *cadre* définit un ensemble d'interfaces fournies et requises, une interface étant une collection de méthodes, et un protocole sur les alphabets induits par ces interfaces. Une *architecture* est la réalisation d'un cadre par un ensemble d'autres cadres et la définition de connexions entre ceux-ci, autrement dit une structure sensée implanter le cadre. Ces deux éléments peuvent se combiner de manière hiérarchique jusqu'à atteindre les composants primitifs par définition indivisible.

La principale propriété que l'on peut définir et vérifier sur les différentes entités composant un système est la notion de *substituabilité de protocole* qui permet de s'assurer de la compatibilité d'une architecture avec une *frame* et des deux parties prenantes dans une connexion. Cette propriété s'énonce informellement comme A est substituable à B si :

1. A fournit au moins tous les services fournis par B :
2. et A ne requiert pas plus de son environnement que n'en aurait exigé B *dans les mêmes conditions*.

On retrouve dans SOFA les deux niveaux de description proposés dans Rapide : l'interface ou ici le *cadre*, et l'architecture qui réalise cette interface.

2.3 Spécification formelles & composants

Cette dernière section est consacrée à des modèles très théoriques prenant en compte des problématiques de composition et de connexions de composants.

2.3.1 π-calcul et algèbres de processus

Le π-calcul [113, 114] est une algèbre de processus destinée à modéliser et étudier les propriétés des systèmes distribués. Ce formalisme se distingue de ses précurseurs et de ses successeurs par une remarquable économie de moyens syntaxiques et une expressivité non moins remarquable. L'élégance et l'expressivité du langage proviennent de l'uniformité de traitement offerte par la notion de *noms*, un terme très abstrait qui est à la fois une variable, un identifiant de canal de communication et une donnée que l'on peut échanger.

La notion d'équivalence comportementale est capturée par le concept de *bisimulation*, relation fondamentale entre les termes du π-calcul et que nous retrouverons dans le chapitre 3 consacré au test de composants. Le principal inconvénient du π-calcul est aussi son principal avantage : sa *puissance*. Cette puissance induit *de facto* l'indécidabilité de la plupart des propriétés intéressantes dans le cas de la version générale du langage et en particulier de l'équivalence comportementale des termes, donc de la relation de bisimulation. La sobriété de la syntaxe et de la sémantique rendent inutilisable le π-calcul autrement que comme objet de réflexion théorique, et c'est pourquoi de nombreux travaux ont cherché à accroître son « *utilisabilité* », par exemple au travers de calculs polyadiques, de l'introduction de types primitifs et d'un système de types sur les termes.

Piccola

Piccola est un langage de composition développé par le *Software Composition Group* de Berne[4, 92, 120]. Bien qu'il ne s'agisse pas à proprement parler d'une plate-forme de composants, cette approche nous semble intéressante car elle met l'accent sur la notion de connecteurs, réifiés sous la forme de scripts Piccola. Ces scripts permettent à des composants écrits simplement en Java de s'échanger des données structurées — appelées *forms* — en s'abstrayant de l'architecture matérielle et logicielle de l'application. La sémantique du langage est basée sur une version du π-calcul , le π-calcul asynchrone polyadique dans lequel les opérations de communication peuvent mettre en œuvre des n-uplets de noms et sont réalisées de manière asynchrone. Il ne semble pas toutefois que ces développements s'intéressent à la vérification et à la validation de programmes autrement qu'au moyen d'un système de types[119].

Darwin

Darwin[98] est un ADL qui permet de définir des architectures de composants inter-connectés et hiérarchiquement structurés, et dans lequel la sémantique des opérations de connexion est définie au moyen de termes du π-calcul polyadique synchrone. L'idée centrale consiste à attacher à chaque port de service fourni et requis un terme du π-calcul et, à partir d'une configuration donnée, de vérifier par application des règles de réduction du calcul que la configuration est correcte, autrement dit que chaque port requis se trouve connecté au bon port fourni. Ce principe de base est étendu au problème de la création de nouvelles instances de composants, création qui est exprimée aussi comme un terme du π-calcul et dont la sémantique permet de vérifier à la conception et à l'exécution sa viabilité en fonction d'un contexte. Notons que ce langage ne s'intéresse pas au fonctionnement des composants ni à la spécification de leurs services mais uniquement à la validation d'une structure donnée.

CORBA & π-calcul

Les travaux de Canal *et al.* [42][45] utilisent le π-calcul pour spécifier le comportement d'interfaces CORBA et [43] en détaillent les aspects formels. Le comportement d'interfaces — rôles — et de composants d'un système est spécifié sous la forme d'agents du π-calcul et une relation de *compatibilité* entre agents est proposée permettant de vérifier la conformité de deux interfaces entre elles, dans le but par exemple d'adapter le comportement de l'une à l'autre et de pouvoir dériver automatiquement des *adaptateurs* et connecteurs possédant certaines propriétés et compatibles avec une architecture donnée. Cette *relation de compatibilité* permet de prouver que la composition de deux composants au travers de leurs interfaces est correcte si les *connexions* entre interfaces sont compatibles, une propriété importante que nous étudierons dans le contexte qui est le nôtre au chapitre 6. Enfin, une relation d'héritage entre agents et d'extension de comportement est définie qui préserve la compatibilité des interfaces.

Dans Bracciali *et al.* [35], le problème de l'adaptation automatique du comportement d'interfaces est étudié dans le cadre présenté ci-dessus. La technique utilisée consiste à construire par étapes un *agent d'adaptation* sous la forme d'un terme du π-calcul , à partir d'une application — *mapping* — entre les signatures et les contraintes de chacune des interfaces à adapter, de sorte que le nouvel agent composé avec les deux agents initiaux produise un comportement correct.

Le choix du π-calcul est ici, comme pour les travaux précédemment cités, guidé par la capacité de cette théorie à modéliser facilement la mobilité et la dynamique structurelle des systèmes. Il n'est bien sûr pas le seul formalisme de la famille des algèbres de processus à avoir été utilisé pour spécifier le comportement d'architectures logicielles : l'ADL *Wright*, par exemple, (voir section 2.2.2) utilise CSP comme outil de spécification de comportements et le *kell-calcul* est une variante complexe du calcul des ambients pour définir une sémantique formelle à la plate-forme Fractal.

M-Calcul & Kell-Calcul

Le M-calcul[142] est un *calcul de processus* inspiré de prédécesseurs tels que le *calcul des ambients*, le *join-calculus*, le *blue calculus*, la *Chemical Abstract Machine*. C'est-à-dire qu'il s'intéresse non seulement aux processus, à leurs compositions au travers d'opérateurs algébriques, mais aussi à la notion, centrale pour les modèles à composants, d'encapsulation et de structure. La principale entité de ce calcul est la

cellule qui, par analogie avec la cellule biologique, est constituée d'une *membrane* et d'un *plasme*, chacun étant décrit comme un processus.

Le point qui distingue le *M-calcul* de ses concurrents est la manière dont ont lieu les communications entre cellules. Les cellules sont organisées de manière hiérarchique à partir d'une racine, le système. La membrane agit comme un contrôleur et un filtre sur les messages qui entrent et sortent du plasme, ce qui autorise toutes sortes de modélisations de systèmes distribués complexes : pare-feux, systèmes d'authentification, systèmes tolérants aux pannes, résolution dynamique de noms ...

Le *kell-calcul*[143] est un raffinement du M-calcul dans lequel ont disparu les concepts distincts de membrane et de plasme au profit d'une vision plus uniforme de processus imbriqués *à la* calcul des ambients. La communication est rendue plus abstraite et plus générale par l'utilisation d'un mécanisme de filtrage de motifs paramétrant le langage, d'où la qualification du kell-calcul comme une famille de langages. Ce filtrage opère sur la structure des messages échangés entre les différentes cellules ou *kells* et peut être arbitrairement complexe. L'objectif de ce projet est de fournir une sémantique formelle, complétée de l'arsenal usuel : système de types, relations d'équivalences, résultats de décidabilité, pour la plate-forme de composants Fractal (voir section 2.1.2).

2.3.2 Composants & coalgèbres

Les approches présentées ci-dessous s'inscrivent dans la théorie des coalgèbres définie en termes catégoriques et dont une synthèse est donnée dans Jacobs et Rutten [76]. La notion de coalgèbre *dualise* la notion d'algèbre et permet de définir un cadre pour la spécification abstraite de systèmes dynamiques au comportement infini, la comparaison par bisimulation et la preuve par coinduction.

Abstract Behavior Types & REO

Le modèle de composants proposé dans Arbab [12] est basé sur la notion de *type abstrait de comportement* défini comme une relation entre des *flots de données temporisés* d'entrée et de sortie. Informellement, les composants modélisés sont supposés échanger des messages avec leur *environnement* au travers d'interfaces soit en entrée, soit en sortie. Une interface est un flot de données temporisé : une séquence infinie de paires d'éléments d'un ensemble quelconque et de réels strictement croissants. Les événements sont supposés ordonnés, atomiques et possèdent une durée non nulle.

Un composant est ainsi défini par ses interfaces d'entrée et de sortie et par une relation entre les flots de données sur ces interfaces. Ce modèle permet de fournir une sémantique au langage de composition REO introduit par ailleurs dans Mehta *et al.* [106] qui définit un ensemble de connecteurs et de composants primitifs composables pour produire des systèmes plus larges.

Un connecteur en REO est un ensemble de *canaux* organisés en graphe où les nœuds sont des regroupements de points d'attaches de canaux et les arcs entre deux nœuds contenant les points d'attache du canal. Un canal est un *medium* de communication entre deux points d'attache. Cet ensemble minimal de définitions est complété par une opération de regroupement de points d'attache, ou *join*, réunissant plusieurs points d'attache dans un seul nœud, et une sémantique de transmission des messages dans les nœuds qui permet en particulier de répliquer les messages sur tous les canaux liés à un point d'attache.

L'intérêt principal de cette approche est l'accent mis sur la topologie abstraite des connecteurs et des nœuds. Cette structure et ces opérateurs permettent de définir des propriétés de coordination et de communication de composants de manière *exogène*. Les composants eux-mêmes n'ont besoin d'aucune connaissance sur les autres acteurs de leur environnement et sont complètement encapsulés.

Arbab et Rutten [13] présente cette même approche en la reliant aux notions de bisimulation et de coinduction. La définition d'une relation de bisimulation permet d'obtenir un principe de preuve pour l'équivalence de connecteurs, principe qui peut être utilisé pour la vérification de protocoles ou pour optimiser un système fait de connecteurs plus simples. La sémantique opérationnelle de REO est définie par des *automates de contraintes*, introduits dans Mehta *et al.* [106]. Ces contraintes sont des gardes permettant de définir la relation existant entre les données sur les différents ports de l'automate et des propriétés du flot temporel ordonnançant les données.

On retrouve dans Reo, formalisés pour le temps-réel, tous les concepts des architectures de composants : connecteurs, interfaces et points d'attache, composition et encapsulation.

Composants génériques et monades

Une autre approche plus abstraite basée sur la théorie des coalgèbres est présentée dans Barbosa [19], Barbosa et Meng [20], Meng et Aichernig [107]. Un composant est défini par ses entrées, ses sorties et une certaine structure de coalgèbre définissant le comportement du composant par les relations entre ses entrées et ses sorties. Cette structure est paramétrique au sens où pour un même composant on peut décrire différentes *formes* du système de transition décrivant le comportement du composant, au moyen de constructions classiques dans la théorie des catégories[97].

On peut définir de manière abstraite une opération d'encapsulation, transformant les ensembles d'entrées-sorties et différentes opérations de composition : séquentielle, alternative — *ie.* non-déterminisme induit par l'environnement — parallèle, synchronisée. On notera que les propriétés de ces opérations sont dépendantes des caractéristiques de la structure paramétrant le comportement des composants observés. Dans Meng *et al.* [108], cette approche est utilisée pour formaliser la sémantique de certains diagrammes UML, diagrammes de classe, cas d'utilisation et diagrammes d'états, et décrire des propriétés de raffinement.

Pour intéressante qu'elle soit sur le plan théorique, cette approche est très éloignée des problèmes posés par le développement d'architectures de composants, le terme même de composant n'étant pas clairement rattaché à un concept opérationnel.

2.4 Conclusion

Nous avons présenté dans ce chapitre différents outils basés sur la notion *d'architecture de composants* : des plate-formes d'exécution dans lesquelles les composants sont des entités exécutables, des langages de conception semi-formels ou formels permettant de concevoir et modéliser des architectures de composants et des systèmes formels, intégrés à une plate-forme ou non, et permettant de définir des composants et des architectures dotés d'une sémantique précise et d'un système de preuve de propriétés.

Tous ces outils partagent un minimum de concepts clés :

1. les *composants* qui forment les briques de base d'un système ;
2. les *données* qui structurent l'information échangée et transformée par les composants ;
3. les *ports* qui permettent d'assembler des composants et de faire transiter de l'information ;
4. les *composites* qui permettent de produire à partir d'un ensemble de composants connectés par leurs ports un nouveau composant lui-même de nouveau composable.

Les *connecteurs* ne semblent pas être un concept primitif de l'architecture car ils peuvent être modélisés à volonté par des composants. De même, la notion de *conteneur* ou de *membrane* est réductible à celles de composite.

Les outils présentés posent toutefois quelques problèmes lorsque l'on souhaite les utiliser dans le cadre d'un processus de développement dans lequel les activités de vérification et de validation sont distinctes des activités de production de code. Les intergiciels à composants sont trop peu abstraits et trop complexes pour intervenir dans une conception formelle d'architecture. Par contre, le fait qu'il y ait une congruence forte entre les concepts du modèle et les entités concrètes manipulées à l'exécution devrait permettre de faciliter la validation du code produit. Les langages de description d'architectures sont à l'inverse adaptés aux activités de conception et de modélisation, permettant dans certains cas de mettre en œuvre des techniques de raffinement, de vérification de modèles ou de preuves de propriétés, mais sont souvent éloignés des plate-formes concrètes. Les approches par extension formelle d'un langage existant[8, 45] nous semblent offrir le bon compromis entre abstraction et réalisation concrète, mais elles sont trop liées à un langage ou une technologie particulière.

Chapitre 3

Test de conformité

Ce chapitre est consacré à une revue des théories, techniques et outils relatifs au test fonctionnel de logiciels en général et plus particulièrement aux méthodes de génération automatique de tests à partir de spécifications formelles de comportement sous la forme de systèmes d'états-transitions. Cette technique particulière de test est le plus souvent dénommée *test de conformité* ou *conformance testing* et est un cas particulier de test dit fonctionnel.

Nous commencerons cet exposé par un certain nombre de généralités et de définitions sur le test de logiciels, la profusion de termes pouvant prêter à confusion. La deuxième partie de ce chapitre sera consacrée à l'étude de la problématique du test fonctionnel automatisé basé sur un modèle formel qui constitue le cœur de notre démarche. Nous étudierons comment différents formalismes définissent un contexte de test ou un cadre formel de validation des résultats du test, et comment il est possible de construire des ensembles de *cas de test*, c'est-à-dire des *suites de tests*, de manière automatisée. La définition de la notion de conformité d'une implantation donnée par rapport à une spécification sera bien évidemment abordée. Nous terminerons enfin ce chapitre par une synthèse des méthodes de test structurel et une discussion des problématiques relatives au test de composants architecturaux.

3.1 Généralités

G.Myers dans son ouvrage abondamment cité[117] définit le test comme suit :

> « Testing is the process of executing a program with the intent of finding errors. »

Autrement dit, le test est, comme toute activité expérimentale, un processus qui présuppose l'existence d'erreurs et qui a pour objectif de les trouver. Par conséquent, et ce point a été souligné historiquement par E.Dijkstra dans une autre citation célèbre,

> « Testing can show the presence of bugs but never their abscence. »

le résultat d'un processus de test fini est donc, dans le meilleur des cas, une forte présomption d'absence d'erreurs mais jamais une certitude et de fait un test réussi est un test qui *trouve* une erreur. Cela ne nous empêchera pas par la suite de raisonner en posant l'hypothèse de l'existence d'un ensemble de tests exhaustif, susceptible de nous fournir une garantie certaine de correction, mais ce raisonnement servira uniquement de point de départ théorique pour produire des tests de la manière la plus adéquate possible.

L'unité de base du test est le **cas de test** ou **test élémentaire**. Un cas de test, d'après UIT-T [158],

> « [...] précise le comportement du testeur dans une expérimentation séparée qui teste un aspect de l'implantation sous test [IUT pour *Implementation Under Test*] et qui donne lieu à une observation et à un verdict. »

Un ensemble de cas de test constitue une **suite de tests**.

3.1.1 Typologie

Cette section définit un certain nombre de termes couramment utilisés dans le domaine du test. Certains d'entre eux recevront une définition plus précise dans les autres sections de ce chapitre. Le lecteur pourra

se référer aux ouvrages classiques du domaine d'où sont tirés, parfois paraphrasés ou légèrement modifiés, les termes introduits dans cette section : [25, 30, 117, 158, 164].

Stratégie de test

Une typologie du test peut tout d'abord être faite en fonction de la stratégie de test mise en œuvre et des objets manipulés par le processus de test. Dans le **test fonctionnel**, le processus de test dispose :

– d'une *spécification* du comportement du programme ;
– d'un *programme* à tester, supposé respecter la spécification.

La spécification est ici utilisée à la fois pour définir la *suite de tests*, c'est-à-dire un ensemble — fini — de tests atomiques, et pour valider le comportement observé du programme soumis aux tests, jouant ainsi le rôle d'**oracle** de test. Le test fonctionnel est aussi appelé test *boîte noire*, test *basé sur les modèles*, test *basé sur les spécifications*, test de *conformité*, test *comportemental* ...

Lorsque la spécification décrit des propriétés non applicatives du logiciel, par exemple des temps de réponse, des taux de transferts, la tolérance aux pannes, la capacité de reprise sur erreur, la montée en charge, on parlera de **test non-fonctionnel** ou de *test de qualité de service*. La distinction entre une propriété applicative et une propriété de qualité de service est purement arbitraire comme l'est la séparation entre exigences fonctionnelles et exigences techniques. Elle dépend généralement du processus métier matérialisé par le logiciel : le fonctionnel d'un développeur de SGBDR ou de logiciels de routage est le non-fonctionnel d'un système d'information d'entreprise.

On oppose généralement au test fonctionnel le **test structurel** dit aussi test *boîte blanche*, test *boîte de verre*, test *basé sur le code*. Dans le test structurel, le processus du test dispose du programme à tester et de son *code source*, ce dernier servant à la génération des cas de tests, dans l'optique de produire une suite de tests adéquate pour un certain objectif de couverture (voir section 3.3). Le test structurel est à notre avis une option supplémentaire au test fonctionnel, un moyen de compléter l'information donnée par la spécification avec des informations contenues dans le code du logiciel testé, dans le but d'améliorer la pertinence des résultats obtenus par le test.

Nous désignerons l'ensemble des éléments à la disposition du processus de test pour construire une suite de tests par le terme de **modèle de test** : documentation, spécifications formelles ou non, exigences, code source, assembleur peuvent tous faire partie du modèle de test et être utilisés pour construire des cas de test.

L'**objectif de test** définit la manière dont vont être produits les cas de test. Étant entendu qu'il est impossible, hormis les cas les plus triviaux, de tester exhaustivement une entité logicielle, la fixation d'un objectif de test permet de sélectionner les cas de test parmi l'ensemble de tous les cas de test possibles. Un autre terme désignant le même concept est celui de **critère de test**. Le terme anglais de *test purpose*, aussi traduit par objectif de test, est un cas particulier de l'objectif de test au sens défini ici : une sélection restreinte dans l'ensemble des comportements de l'entité à tester. La définition d'un objectif dépend évidemment des éléments à la disposition du testeur.

Le terme de **couverture** désigne une famille de critères à partir de laquelle on cherche à obtenir une suite de tests permettant de couvrir une certaine fraction du modèle de test. Dans son acception la plus courante, la couverture est celle du code de l'implantation sous test dans le cadre du test structurel.

Des critères de test peuvent aussi être *statistiques*, en liaison avec la notion de fiabilité statistique du logiciel et de *profil opérationnel* d'utilisation, ou des critères *heuristiques* — on parlera aussi d'*hypothèses de test* — liés à un certain *modèle de défaut* : par exemple, le fait de sélectionner parmi un ensemble de valeurs possibles pour les paramètres d'une fonction des valeurs limites est un critère heuristique courant.

Un objectif pour le processus de test peut être constitué d'une combinaison quelconque d'un ensemble de sous-objectifs et peut servir soit dans le cadre d'un processus de génération automatique, soit dans un processus de production manuelle des cas de tests. Le problème principal consiste à s'assurer que l'objectif a bien été atteint.

Enfin, une stratégie de test est toujours liée à un **modèle de défaut** implicite ou explicite, qui est un ensemble d'hypothèses sur les défauts présents dans l'implantation testée : on ne peut généralement détecter que les pannes que l'on cherche. Ce modèle de défaut dépend du modèle de test.

Granularité

La *granularité* des entités que l'on souhaite tester est une autre dimension du processus de test. Elle a un impact évident sur la stratégie de test puisque selon les cas, le processus aura à sa disposition différentes informations pour construire sa suite de tests et évaluer le résultat.

Le **test d'acceptation** appelé aussi plus communément *test de recette* ou simplement *recette* vise à valider un système complet du point de vue de l'*utilisateur* du système. C'est généralement la dernière phase du développement d'un logiciel avant sa *livraison* et le démarrage, si nécessaire, d'une nouvelle itération du cycle de développement. Étant réalisé du point de vue de l'utilisateur, le test d'acceptation va chercher à exercer l'ensemble des fonctionnalités du logiciel en situation de production. Les *jeux et plans de tests* sont développés à partir des exigences du cahier des charges, complétées de la connaissance des utilisateurs du système agissant comme testeurs.

Le **test système** est similaire dans sa portée au test d'acceptation : l'ensemble du système constitue l'IUT et le processus va chercher à valider son comportement par rapport aux exigences exprimées par l'utilisateur, transcrites dans l'analyse et la conception. Le test système est aussi appelé test *basé sur les exigences — requirements based testing* en anglais.

Le **test unitaire** consiste à tester une entité susceptible d'être composée ou intégrée par la suite dans une entité plus vaste. Le point important dans le test unitaire est la notion d'*isolement* : l'IUT est testé en dehors de toute interaction avec d'autres entités du système ce qui suppose d'être capable :

1. de l'*exécuter* de manière autonome ;
2. de *simuler* sous contrôle du testeur l'environnement nécessaire à son bon fonctionnement .

Le **test d'intégration** a pour objet de vérifier qu'un ensemble d'entités logiquement liées peuvent coopérer. Les entités concernées peuvent être aussi simples que des fonctions ou méthodes d'une classe, une grappe d'objets formant un tout cohérent ou un ensemble de sous-systèmes formant un système plus complet. Le test d'intégration peut être vu comme une phase préparatoire au test unitaire d'un niveau de granularité plus élevé.

Le **test de non-régression** permet d'une part de s'assurer du maintien des fonctionnalités d'une IUT entre deux versions, d'autre part de vérifier que des erreurs corrigées dans une version n ne réapparaissent pas dans une version $n + 1$. Pour chaque erreur détectée par la recette ou le test système, un cas de test témoin doit être produit pour enrichir la suite de tests de non-régression.

3.1.2 Exécution du test

Le test est une activité de vérification *dynamique* d'un objet logiciel, par opposition aux techniques de vérifications statiques. Il suppose l'*exécution* du logiciel et l'observation de son comportement. Cet dynamicité pose d'emblée la question de la fiabilité et de la traçabilité des résultats du test puisque cette exécution est dans tous les cas réalisée par le truchement d'un compilateur ou d'un interpréteur, voire d'un intergiciel. Le testeur doit être capable de relier des événements entre deux objets distincts : le code exécute et le code source. Cette liaison est généralement faite par l'introduction d'informations de *déverminage* à la compilation, informations qui ne seront pas conservées en production. Il est donc amené à faire l'hypothèse de la fiabilité du compilateur.

Cette exécution produit un résultat qui dépend des fonctionnalités de l'entité logicielle testée : une fonction produit une valeur à partir de paramètres d'entrée, un objet change d'état selon les méthodes qui sont invoquées, un système de gestion transforme des données stockées dans une base de données à partir des actions réalisées par un utilisateur, un système de pilotage d'une chaîne de production émet des commandes en fonction de la réception de signaux de son environnement.

La conclusion d'un cas de test est donc le résultat de la comparaison entre le comportement *observé* de l'objet testé et le comportement *attendu* qui est partie intégrante de la définition d'un cas de test. Cette comparaison produit donc un **verdict de test** qui peut être :

- un **succès** lorsque le comportement observé est égal, quel que soit le sens de *égal* dans le contexte choisi, au comportement attendu ;
- un **échec** lorsque le comportement observé et celui attendu divergent.

En anglais, on parlera dans ce dernier cas de *failure*, terme que l'on peut aussi traduire par *panne*. Il ne faut pas confondre une *panne* avec un *défaut — fault* en anglais. Un défaut est un élément de l'objet testé à l'origine de l'échec ou de la panne. Une *erreur* est une action ou une inaction humaine ayant provoqué l'introduction du défaut et donc indirectement la panne. Un même défaut peut être à l'origine de plusieurs pannes et inversement une panne peut être causée par la conjonction d'un ensemble de défauts. Le cas peut même exister de défauts qui se corrigent entre eux pour *masquer* une panne. Le *déverminage — debugging* en anglais — est l'activité consistant à identifier les défauts et corriger les erreurs ayant entraîné un échec lors du test.

Certains auteurs et la norme UIT-T [158] définissent aussi comme résultat possible du test le résultat *non-concluant* : aucun échec n'a été constaté mais il est nécessaire de poursuivre l'exécution du test pour valider le résultat. Cette notion renvoie à la définition d'un objectif de test. Un test atomique ou un ensemble de tests peuvent être considérés comme non-concluants tant que l'objectif de test n'est pas atteint. Ce type de verdict n'est pas à notre avis directement lié à l'exécution d'un seul (cas de) test. Un résultat non-concluant peut aussi survenir lorsque le comportement de l'objet testé est inobservable, c'est à dire que l'on est incapable de conclure sur le résultat du test du fait de l'absence de résultat observé.

Ceci nous amène naturellement aux notions d'**observabilité** et de **contrôlabilité** de l'IUT qui définissent sa **testabilité**[60, 84]. Si l'on modélise un IUT comme une boîte noire acceptant des événements en entrée et produisant en réaction des événements en sortie, l'IUT est *observable* si des sorties distinctes sont le fait d'entrées distinctes. Autrement dit, l'IUT est observable si la relation entre entrées et sorties est une *fonction*. Un IUT est *contrôlable* si, pour chaque événement susceptible d'être produit en sortie, il existe un événement d'entrée permettant d'observer cette sortie. Un IUT qui est contrôlable et observable est une *fonction surjective*. Il s'agit ici de propriétés de l'IUT — et indirectement de la spécification à partir de laquelle l'IUT a été développé — qui peuvent avoir de toute évidence un grand impact sur la capacité du test à révéler des pannes.

La procédure de décision effectuant la comparaison entre résultat attendu et résultat observé est appelée **oracle de test** et sa construction correcte est bien évidemment un point essentiel du processus de test. Cet oracle est déduit, manuellement ou automatiquement, du *modèle de test*. Sa correction suppose, comme nous l'avons vu, que ce modèle soit lui même *testable*, ou en d'autres termes observable et contrôlable. Le problème de l'oracle n'est pas un problème trivial mais dans le cas où le modèle utilisé pour le test est une spécification formelle, ce problème est normalement résolu par la sémantique propre au système formel utilisé.

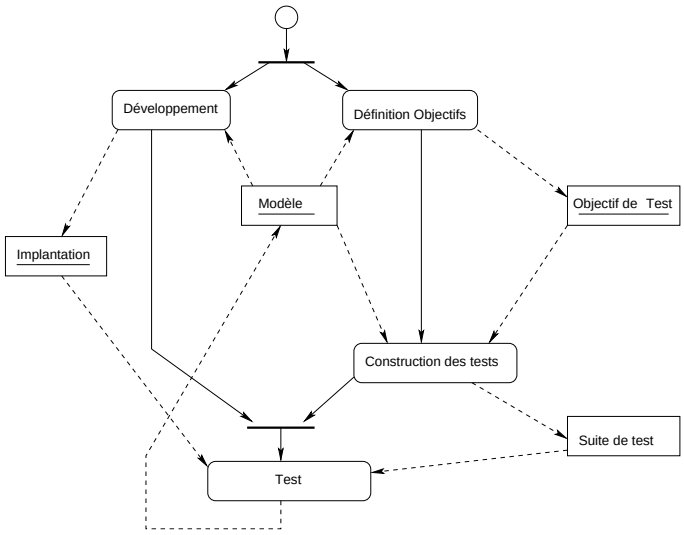
3.1.3 Synthèse

La figure 3.1 représente sous la forme d'un diagramme d'activité UML les interactions entre les différentes entités menant à la définition et l'exécution d'un processus de test. Ce schéma met en avant un point essentiel : l'impact que le processus de test peut avoir sur la conception. D'une part, comme nous l'avons vu dans la précédente section, la testabilité d'une entité logicielle dépend fortement de celle de sa spécification au sens le plus large. Par conséquent le fait de mettre en place un processus de test suppose que l'on ait vérifié cette dernière ou que l'on puisse la corriger en fonction des résultats du processus de test. L'évaluation de la testabilité d'un modèle ainsi que la définition de techniques de conception favorisant la testabilité sont des thèmes de recherche actifs.

D'autre part, et de manière assez évidente, l'échec d'un test compte tenu des différentes entités qui entrent en jeu dans sa construction et son exécution peut signifier que :

- soit l'implantation testée présente un défaut, ce qui est le cas le plus « favorable » ;
- soit le modèle présente un défaut : par hypothèse, le processus de développement est une activité humaine tandis que, dans le cas qui nous intéresse, le processus de construction des cas de tests est une activité automatisée. Il est donc possible que des erreurs dans le modèle ne soient pas conservées dans l'implantation alors qu'elles le seront dans la suite de tests ;
- soit la suite de tests elle-même présente un défaut. Cette possibilité est supprimée toujours par hypothèse d'une transformation automatisée — et fiable — du modèle en suite de tests.

La figure 3.2 est une vue plus précise de l'activité de test unitaire proprement dite. Le processus requiert tout d'abord la validation des tests d'intégration des différentes entités intégrées dans l'IUT. On suppose par ailleurs que leur testabilité est contrôlée en amont du processus. La conception des tests



htbp

FIG. 3.1 – Activité de test.

dépend, nous l'avons vu, du modèle et de l'objectif fixé, le premier étant utilisé par ailleurs pour construire l'*environnement de test* ou *contexte de test*, c'est à dire la simulation de l'ensemble des ressources dont dépend l'IUT. Le processus d'exécution produit un verdict : si c'est un échec, alors l'IUT contient un ou plusieurs défauts qu'il est nécessaire de corriger avant de retester ; si c'est un succès, il est nécessaire alors de vérifier que l'objectif de test est bien atteint. Notons que dans un processus automatisé, les suites de tests étant produites à partir de l'objectif de test, cette vérification est immédiate et toujours réussie.

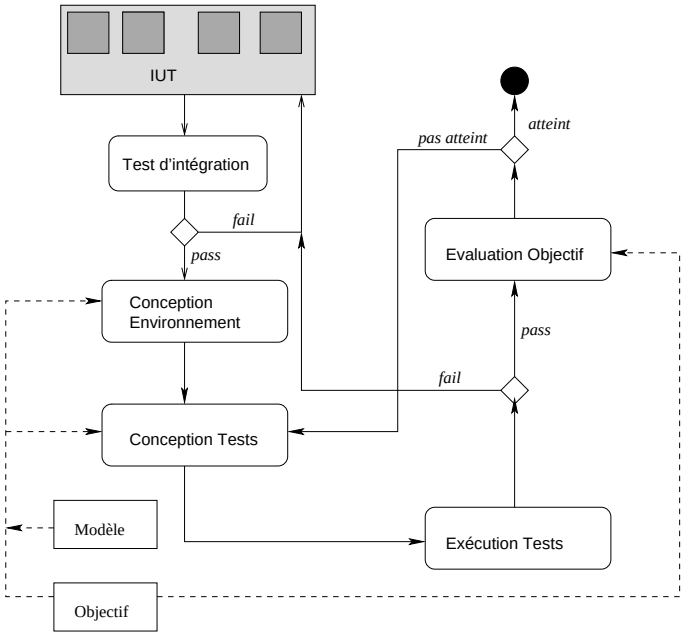


FIG. 3.2 – Activité de test (détail).

3.2 Test de conformité

À la section précédente nous avons fixé de manière informelle la terminologie parfois confuse que l'on retrouve dans tout ou partie des travaux sur le test. Nous avons par ailleurs identifié un processus général de test unitaire prenant en compte de manière abstraite l'ensemble des facteurs conduisant à la réalisation effective d'un processus de test. Il n'est clairement pas question de passer en revue l'ensemble du domaine extrêmement riche que constitue le test de logiciel et nous allons donc nous attacher ici à en détailler la fraction qui nous intéresse particulièrement, le test basé sur des *spécifications formelles comportementales* sous la forme d'automates d'états finis, de systèmes de transition, de machines abstraites.

Ces travaux sont issus pour beaucoup de la problématique du test de protocole de communication. De ce fait leur applicabilité au test de composants logiciels tels que nous les avons définis dans le chapitre 2 est presqu'immédiate, les architectures de test étant fortement similaires. D'un point de vue théorique, ces travaux sont reliés d'une part au problème de l'équivalence sémantique de processus modélisés comme des systèmes de transitions, et d'autre part à des recherches sur l'identification d'automates de MOORE et plus généralement de transducteurs et d'automates de mots.

Nous effectuerons toutefois une incursion dans une autre famille de tests, basés sur des spécifications algébriques, dans la mesure où certains travaux sur cette catégorie de formalisme ont produit des concepts pertinents pour le test de conformité de systèmes de transitions.

3.2.1 Test algébrique

Dans Bernot *et al.* [26] sont posées les bases d'une théorie du test appliquée aux spécifications algébriques. Cette théorie est développée dans Marre [99] et par la suite reprise notamment dans Barbey *et al.* [17, 18], Le Gall et Arnould [84], Lestiennes et Gaudel [89], Péraire *et al.* [129].

On cherche ici à formaliser des notions plutôt floues que sont la testabilité, la validité d'un test, la conformité, les hypothèses nécessaires au test. L'idée principale consiste à partir d'une *suite de tests* exhaustive pour laquelle, sous une *hypothèse de testabilité* de l'implantation considérée, on a l'assurance que si l'implantation passe la suite de tests, elle est conforme. Cette suite de tests étant généralement impraticable, elle va être raffinée par applications successives d'*hypothèses de test*, non-biaisées (saines) et valides, qui permettent d'en réduire la taille. Le processus s'arrête lorsque l'on atteint une suite de tests acceptable.

Les deux types d'hypothèses principales sont dénommées *hypothèse de régularité* et *hypothèse d'uniformité*[26], elles formalisent des pratiques courantes du test telles que le test de partitionnement, le test aux limites, la sélection de profondeurs de boucles ou de récursions, ... Phalippou [131] introduit d'autres hypothèses telle l'*équité* ou l'*indépendance* qui permettent de gérer les problèmes liés au non-déterminisme et au parallélisme.

Le couple formé des hypothèses de test et d'une suite de tests réduite issue de l'application des hypothèses à la suite de tests exhaustive doit maintenir deux propriétés :

1. il doit être *valide* : si l'implantation passe la suite de tests réduite, alors elle passera la suite de tests exhaustive ;
2. il doit être *non-biaisé* : si l'implantation passe la suite de tests exhaustive, alors elle passera la suite de tests réduite.

En d'autres termes, une suite de tests doit accepter toutes les implantations correctes et ne pas accepter les implantations incorrectes.

Dans Le Gall et Arnould [84], à la notion d'ensemble de tests *exhaustif* telle que définie précédemment s'ajoute la notion d'ensemble complet défini comme une suite de tests non-biaisée et *maximale* : toute autre suite de tests a un pouvoir de détection plus faible. Un point important de l'article est qu'un ensemble complet n'est pas nécessairement exhaustif et qu'il n'existe pas nécessairement d'ensemble exhaustif pour un programme donné dans un certain cadre d'observations. Si certaines propriétés du programme testé ne sont pas observables, elles ne pourront être vérifiées par le test. Un programme sera donc considéré comme *partiellement correct* s'il est valide par rapport à une suite de tests complète.

Cette formalisation générale du test est reprise par ailleurs dans la norme UIT Z.500 [158], « Cadre général des méthodes formelles appliquées au test de conformité ».

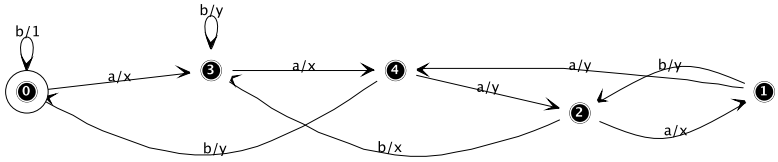


FIG. 3.3 – Spécification d'un FSM.

3.2.2 Test d'automates d'états fini (FSM)

Un *automate d'états fini* ou FSM — *Finite State Machine* — est un n-uplet $(Q, I, O, q_0, \lambda, \delta)$ avec
- Q un ensemble fini d'états ;
- $q_0 \in Q$ un état initial distingué ;
- I et O des alphabets finis d'*entrée* et de *sortie* ;
- $\lambda : Q \times I \rightarrow O^*$ la fonction partielle de sortie ;
- $\delta : Q \times I \rightarrow Q$ la fonction partielle de transition.

Un FSM est aussi appelé *transducteur séquentiel*[28, 140] et calcule une fonction rationnelle transformant un langage d'entrée inclus dans I^* en un langage de sortie inclus dans O^*.

Étant données un automate A dont la structure est connue et un automate M dont la structure est inconnue mais dont les alphabets sont supposés identiques à ceux de A, le *test de conformité* de FSM consiste à déterminer si les deux automates calculent la même fonction en examinant les sorties produites par M pour un ensemble de mots d'entrées, les séquences de test.

Dans le cas général où λ et δ sont des fonctions partielles, on distinguera[146] des séquences de test de *conformité faible*, qui vérifient l'équivalence de sortie pour les seules entrées spécifiées ; et des séquences de test de *conformité forte* générées à partir d'une complétion de la spécification par des transitions sans sortie pour les lettres non spécifiées dans chaque état.

Un certain nombre d'hypothèses sur M et de contraintes sur A sont généralement posées pour permettre la construction effective de séquences de test : minimalité et/ou déterminisme de A, bornage du nombre d'états de M, correspondance des alphabets, déterminisme de M, ...

Un tel ensemble de *séquences de vérification* produit un résultat certain et bien évidemment, hormis les cas les plus triviaux, sa construction est ardue. On trouvera dans Lee et Yannakakis [87] les complexités de cette construction en fonction des caractéristiques de A : existence ou non de *reset*, de *séquences distinguantes*, ... Nous donnons ci-dessous quelques méthodes classiques — ou moins classiques — de génération de séquences de tests pour les FSM.

Le problème de l'équivalence de FSM remonte au moins aux travaux de Moore dans Moore [115] : étant donné un automate de Moore — l'implantation — peut-on à l'aide d'expérimentations, c'est à dire de séquences de l'alphabet d'entrée, déduire des sorties produites par l'automate testé l'*état initial* de celui-ci ? Plus généralement, peut-on toujours *distinguer* deux automates à l'aide de séquences de test ? La réponse dans le cas général est *non* : la propriété de distinction est indécidable pour un automate arbitraire. Elle est décidable dans le cas d'automates minimaux de taille bornée et l'on obtient même directement une borne supérieure sur la taille des suites de tests nécessaires pour distinguer deux automates donnés — *ie.* une implantation inconnue mais supposée de taille bornée, déterministe et totalement connexe, et une spécification connue.

3.2.3 Sélection des cas de test dans les FSM déterministes

Nous présentons ici quelques méthodes classiques appliquées au test de FSM déterministes. Cette problématique est historiquement importante et théoriquement intéressante, même si en pratique son applicabilité à la génération de tests pour des problèmes réels est faible de par la complexité des algorithmes et surtout la taille exponentielle des suites de tests générées. D'après Lai [82], il semble que seule la méthode UIO ait été appliquée à une échelle industrielle dans le cadre du test de logiciels de

Ensemble W : $\{b, aa, ba\}$.

baaabbaa	xxyxyxy	baaabb	xxyxy
baaababaa	xxyxxyxx	baaababbba	xxyxxyx
baababa	xxyxyx	baabaaa	xxyxxy
baaababa	xxyxxyx	baaababb	xxyxxy
baaababaab	xxyxxyxxy	baa	xx
baaababaaaa	xxyxxyxxyx	baabba	xxyx
baaababab	xxyxxyxy	baabaa	xxyxx
baabbb	xxy	baaababaaba	xxyxxyxyx
baaabba	xxyxyx	baaababaaa	xxyxxyxxy
bbaa	xx	baaababababba	xxyxxyxyyx
baaababbaa	xxyxxyxx	baabab	xxyxy

FIG. 3.4 – Séquences de test W (partiel).

télécommunications. Pour illustrer ces différentes méthodes, nous utiliserons la spécification présentée dans la figure 3.3, exemple tiré de Sidhu et Leung [146].

Toutes ces méthodes construisent des *séquences de test* qui sont des mots finis de l'alphabet d'entrée de l'automate considéré. Un cas de test consiste donc à fournir à l'implantation la séquence de test en entrée — la sensibiliser — et à comparer le mot produit en sortie avec la spécification.

Le *tour de transition*

Dans le cas d'un FSM fortement connexe, le tour de transition consiste à construire une séquence de test unique par parcours de l'ensemble des transitions du graphe du FSM. Cette construction peut être faite soit analytiquement par un parcours du graphe, soit par un parcours aléatoire suivi d'une élimination des sous-séquences redondantes.

La méthode W

La méthode W a été introduite dans Chow [48] et depuis abondamment reprise comme base d'autres méthodes ou citée comme référence. Le principe de cette méthode consiste à construire un ensemble de séquences de tests, des mots appartenant à l'alphabet d'entrée du FSM spécifié, permettant de vérifier pour chaque état q, d'une part l'existence de cet état dans l'implantation sous test, d'autre part la conformité des sorties produites à partir de cet état.

Cette méthode est basée sur la construction d'un ensemble de séquences *caractéristiques* ou *ensemble de distinction*, appelé ensemble W, qui est tel que tous les états du FSM produisent à partir de W un ensemble de séquences de sorties différent et sont donc distinguables les uns des autres. Elle utilise aussi un ensemble de séquences de *couverture des transitions* du FSM modélisé, appelé ensemble T. Les deux ensembles sont concaténés de sorte que les séquences d'entrées définies permettent tout d'abord d'atteindre un état — c'est le rôle de T — et ensuite de vérifier la conformité des sorties produites dans cet état — ce que fait W.

La méthode W permet aussi de vérifier la conformité de FSM dans le cas où l'ensemble des états de l'IUT est potentiellement plus grand que celui de la spécification. Dans ce cas, on intercale entre les ensembles T et W du *bruit* sous la forme d'un ensemble des mots possibles de taille inférieure ou égale à $m - n$, où m est le nombre d'états maximum estimé et n est le nombre d'état de la spécification.

Pour l'automate présenté dans la figure 3.3, la figure 3.4 présente un exemple de suite de tests générée par la méthode W. Notons que cet ensemble n'est pas complet pour des raisons de place et qu'il n'est pas non plus optimisé : de nombreuses séquences se recouvrant, il est tout à fait possible de réduire cette suite de tests en incluant plusieurs tests dans une même séquence.

La méthode Wp

La méthode Wp est une amélioration et une généralisation de la méthode W introduite dans Fujiwara *et al.* [61] dont l'objectif est de réduire la taille des suites de tests générées lorsque c'est possible.

Le principe de l'algorithme est de construire un *ensemble d'ensembles* d'identification indexé par les états du FSM. Pour chaque état q, il est construit un ensemble W_q de séquences de I^* tel que les sorties produites à partir de l'application de toutes les séquences soient différentes pour tous les états de Q. Ainsi, au lieu de concaténer l'ensemble W complet pour identifier chaque état après une séquence de tests de transitions, on peut utiliser seulement les séquences d'identification propres à chaque état et ainsi limiter la taille des séquences.

La méthode UIO

La méthode UIO[5, 139] repose sur la construction, non plus de séquences de distinction comme les méthodes précédentes, mais sur la construction de séquences uniques d'entrée/sortie : une séquence unique d'entrée/sortie, une séquence *UIO* donc, pour un état $q_i \in Q$ est un mot de $(I \times O)^*$ tel que pour tout état $q_j \in Q$, $UIO(q_i) \neq UIO(q_j)$. Pour tout FSM réduit, c'est à dire pour lequel aucun état n'est équivalent pour les deux relations δ et λ, il existe un ensemble de séquences UIO.

L'utilisation des UIO à la place des séquences de distinction permet dans la majorité des cas d'obtenir des suites de tests plus courtes, comme l'illustre la figure 3.5 qui représente l'ensemble UIO et la suite de tests correspondante pour le FSM de la figure 3.3.

Ensemble UIO : $\{(b/1), (a/y, a/y), (b/x), (a/x), (a/y, b/x)\}$.

bb	
aa	xx
aaab	xxyx
aaaabbaab	xxyxyxxyx
aaaba	xxyxx
aaabaaaa	xxyxxyxy
aaabba	xxyxyx
aaab	xxyx
aaaabb	xxyxyx
aaabbb	xxyxyy

FIG. 3.5 – Séquences de test UIO.

3.2.4 Sélection de cas de test généralisée

La possibilité de définir des spécifications non déterministes, soit directement, soit indirectement par une opération de composition, est intéressante pour les capacités d'abstraction qu'elle offre. De plus, dans le cas des *automates d'états fini étendus* ou EFSM[33], du fait des valeurs de variables, ou dans celui de FSM[94] communiquants du fait de l'entrelacement possible des messages, il se pose un problème évident soit d'explosion du nombre d'états si l'on cherche à déterminiser ce type de spécification, soit d'incapacité à garantir un résultat.

Un certain nombre de méthodes a donc été proposé pour sélectionner dans ce contexte des suites de tests finies qui permettent d'obtenir une certaine garantie quant à la couverture de fautes obtenue.

Approches heuristiques du test de FSM

Lee *et al.* [88] présente un algorithme de parcours aléatoire — *random walk* — d'un ensemble de FSM communiquants pour couvrir les transitions de chacun des FSM au lieu de couvrir l'ensemble des transitions globales. Cette approche est étendue dans Zaïdi [167] sous le nom d'algorithme *Hit-or-Jump* pour résoudre le problème de l'*optimum local*, c'est-à-dire de l'incapacité d'un algorithme purement aléatoire à trouver certaines solutions — certains chemins — simples mais avec une faible probabilité d'occurrence qui lui permettrait de sortir d'une boucle locale.

L'algorithme fonctionne en deux temps : une recherche locale d'une certaine profondeur est effectuée en fonction de l'état courant de manière à atteindre des transitions non marquées, si la recherche est infructueuse, un saut aléatoire est effectué en dehors du domaine correspondant à la profondeur choisie. Dans les

deux cas, le chemin permettant d'atteindre la transition ou l'état sélectionné est construit et utilisé comme cas de test.

On notera que ce problème est bien connu dans le domaine de l'optimisation combinatoire et qu'il a donné lieu à un nombre de travaux de recherche considérable et notamment à un grand nombre d'algorithmes heuristiques — voir Michalewicz et Fogel [111] pour une étude détaillée et récente du problème et de ses solutions.

3.2.5 Test de LTS

La théorie du test de LTS — *Labelled Transition System* ou Système de transition étiqueté — est introduite dans Tretmans et Belinfante [155] et détaillée dans Phalippou [131], Tretmans [157]. Brinksma et Tretmans [38] est une bibliographie commentée sur le sujet. Cette théorie est issue de travaux sur la sémantique opérationnelle des langages de programmation pour lesquels on a cherché à caractériser des relations d'équivalence selon un modèle idéal du test : étant donné un contexte de test définissant ce qui est observable d'un processus, deux processus sont équivalents si les observations de toutes leurs exécutions sont identiques. Différentes *équivalences observationnelles* sont étudiées dans van Glabbeek [161] pour des LTS ne comportant pas de transition non-observables, généralement notées τ. Par ailleurs, la malléabilité du formalisme des LTS permet de définir différentes sémantiques d'exécution des programmes modélisés et donc différentes équivalences possibles.

Un LTS S est un n-uplet $(Q, q_0, \Sigma, \rightarrow)$ avec :
- Q un ensemble d'états ;
- $q_0 \in Q$ un *état initial* distingué ;
- Σ un ensemble d'étiquettes de transitions ou d'*actions* ;
- $\rightarrow \subseteq Q \times \Sigma \times Q$ une relation de transitions.

À la différence d'un automate, un LTS, dans toute sa généralité, peut avoir un ensemble d'états infini, un alphabet infini et ne possède pas d'états terminaux. Selon certains auteurs, la notion de processus se confond avec la notion d'état : on parlera de relations entre processus plutôt que de relations entre les états d'un LTS. Nous utiliserons dans la section suivante la notation $\xrightarrow{\sigma}$ pour désigner l'image par la relation $\rightarrow$ d'une séquence d'étiquettes σ. On trouve aussi dans la littérature la notation $q \stackrel{a}{\Rightarrow} q'$ qui est le quotient de $\rightarrow$ par la relation d'équivalence induite par les τ-transitions.

Nombre de relations, à commencer par la plus connue, la bisimulation, sont définies de manière *coinductive*[76] comme la plus grande relation possédant certains caractéristiques et contenant les états initiaux.

Relations d'équivalences dans les LTS

Étant donnés deux systèmes S et C modélisés sous la forme des systèmes de transitions étiquetés $(Q, q_0, \Sigma, \rightarrow)$ et $(P, p_0, \Lambda, \rightarrow_C)$, il est possible de définir un grand nombre de *relations d'équivalence* ou de *préordres* entre S et C, ou plus exactement entre les états — processus — de S et de C. Nous reprenons ici une fraction de la hiérarchie des équivalences et préordres de van Glabbeek [161]. Chaque sémantique est définie par une *relation d'équivalence* notée $\equiv \subseteq Q \times P$, et l'on a $S \equiv C$ si et seulement si $q_0 \equiv p_0$.

L'**équivalence de traces** est la relation la plus faible de la hiérarchie. Deux états ou processus sont équivalents s'ils peuvent produire les mêmes séquences d'étiquettes de transition : autrement dit, les langages clos par préfixes sur les alphabets Σ et Λ induits par les deux processus sont identiques.

Si $T(q)$ dénote l'ensemble des traces — clos par préfixes — pour un état q quelconque, alors

$$C \equiv_{Tr} S \Leftrightarrow T(q_0) = T(p_0).$$

L'équivalence de **traces complétées** est plus stricte : elle prend en compte non seulement les — préfixes de — traces mais aussi les états à partir desquelles plus aucune transition n'est possible. Un mot $\sigma \in \Sigma^*$ est une *trace complétée* pour un état q si et seulement si $\sigma \in T(q)$ et

$$\forall a \in \Sigma, \nexists q'' \in Q, q \xrightarrow{\sigma} q' \xrightarrow{a} q''.$$

Si $CT(q)$ dénote l'ensemble des traces complétées de q, alors

$$C \equiv_{CT} S \Leftrightarrow CT(q_0) = CT(p_0).$$

L'**équivalence de refus** est à la base de la sémantique de CSP. Pour tout état q, on définit l'ensemble $X(q) \subseteq \Sigma$ des *refus* de q comme

$$X(q) = \{a \in \Sigma \mid \nexists q' \in Q, q \xrightarrow{a} q'\},$$

l'ensemble des actions que q *refuse* d'exécuter. L'équivalence de refus — appelée aussi équivalence de trace-refus — entre S et C est définie comme :

$$C \equiv_{Ref} S \Leftrightarrow \forall \sigma \in \Sigma^*, q \in Q, p \in P, \begin{cases} q \equiv_{Ref} p \wedge q \xrightarrow{\sigma} q' \implies \\ \qquad p \xrightarrow{\sigma} p' \wedge X(q') = X(p') \wedge q' \equiv_{Ref} p', \\ q \equiv_{Ref} p \wedge p \xrightarrow{\sigma} p' \implies \\ \qquad q \xrightarrow{\sigma} q' \wedge X(q') = X(p') \wedge q' \equiv_{Ref} p', \\ q_0 \equiv_{Ref} p_0. \end{cases}$$

Les deux processus, pour être équivalents, doivent non seulement pouvoir exécuter les mêmes séquences d'actions, mais dans chaque état atteint, doivent refuser les mêmes ensembles d'actions.

La **bisimulation** enfin — ou simulation symétrique — est l'équivalence la plus fine entre processus caractérisables par les séquences de transitions et les états. Cette relation a été développée surtout à partir de Milner [112] et abondamment étudiée dans le cadre du π-calcul , sous diverses formes. Dans sa définition de base, elle est définie comme suit :

$$C \sim S \Leftrightarrow \begin{cases} q \sim p \wedge q \xrightarrow{a} q' & \implies & \exists p', p \xrightarrow{a} p' \wedge q' \sim p' \\ q \sim p \wedge p \xrightarrow{a} p' & \implies & \exists q', q \xrightarrow{a} q' \wedge q' \sim p' \\ q_0 \sim p_0. \end{cases}$$

Chaque couple d'états de la relation doit accepter les mêmes actions et conduire à des états eux-mêmes bisimilaires.

Il est bien sûr possible de définir des relations d'équivalence plus fines allant jusqu'à l'isomorphisme des deux systèmes considérés. Signalons par ailleurs que ces différentes équivalences n'ont de sens que si l'on suppose que les processus sont non déterministes, éventuellement qu'ils contiennent des actions internes inobservables et que le nombre d'états est potentiellement infini. Par non déterministe, il faut entendre comme dans le cas des automates que $\rightarrow$ est bien une relation, pas une fonction. Non déterministe peut aussi s'entendre comme *non déterminisable* : même si le nombre d'états est fini, il est trop grand pour espérer pouvoir construire un système déterministe équivalent. Dans le cas particulier des processus déterministes, on se retrouve dans la situation classique des automates finis — dont tous les états sont terminaux — et toutes les équivalences se confondent avec l'équivalence de traces.

Il est intéressant de souligner que les équivalences entre processus sont très souvent définies comme des *équivalences de test*. C'est le cas dans van Glabbeek [161] : les différentes relations y sont caractérisées par des *scénarios de test* décrivant les interactions entre un observateur et le processus considéré. Bien évidemment, plus l'observateur a de contrôle sur le processus, plus il peut observer d'actions et distinguer ses différents états, plus la relation d'équivalence décrite est fine.

Comme démontré initialement dans Abramsky [1], la puissance de l'observateur peut aussi être caractérisée par des formules de logique modale — logique de Hennessy-Milner — dont les modèles sont des processus ou systèmes. Plus on dispose d'opérateurs, plus la logique est puissante et plus les modèles deviennent précis.

Relations de conformité dans les IOLTS

Ce modèle de base LTS s'est enrichi pour travailler sur différents types de systèmes, en particulier des IOLTS. Ceux-ci proviennent de travaux sur les I/O automates Garland et Lynch [63], Lynch et Tuttle [95] introduits pour décrire la sémantique de processus réactifs ou communiquants dans les langages tels que LOTOS. Un IOLTS est simplement un LTS $(Q, q_0, \Sigma, \rightarrow)$ dans lequel l'alphabet Σ est partitionné en trois : l'alphabet d'entrée Σ_I, l'alphabet de sortie Σ_O et l'alphabet interne Σ_T ; et pour lequel toutes les entrées sont toujours possibles. L'alphabet peut éventuellement être complété par δ exprimant le blocage ou la terminaison du processus. La problématique principale du test d'IOLTS consiste à définir des relations de conformité et des algorithmes permettant de vérifier l'adéquation entre un modèle, généralement spécifié librement comme un LTS et une implantation considérée comme un IOLTS.

La relation ioco La *relation de conformité* utilisée dans J. C. Fernandez *et al.* [75] comme dans Tretmans et Belinfante [155] est appelée **ioco**. Soit $S = (Q, q_0, \Sigma, \rightarrow)$ un (IO)LTS, on notera $\mathcal{L}(S)$ l'ensemble des séquences acceptées par S :

$$\mathcal{L}(S) = \{\sigma \in \Sigma^*, \exists q \in Q, q_0 \stackrel{\sigma}{\Rightarrow} q\}.$$

Cette relation est définie comme :

$$I \,\mathbf{ioco}\, S \iff \forall u \in \mathcal{L}(S), \{a \in \Sigma_O \mid u.a \in \mathcal{L}(I)\} \subseteq \{a \in \Sigma_O | u.a \in \mathcal{L}(S)\}.$$

Autrement dit, l'implantation ne peut produire plus de sorties que prévues par la spécification. On notera que dans J. C. Fernandez *et al.* [75], les entrées et sorties sont vues du point de vue de l'*environnement* de l'IUT et donc inversées par rapport à cette définition. Phalippou [131] définit un certain nombre d'autres relations de conformité possibles, nommées R_1 à R_5 mais qui peuvent s'exprimer dans le même cadre.

De même que l'on établit une hiérarchie de relations d'équivalence observationnelle entre processus modélisés par des LTS, on peut établir une hiérarchie entre processus modélisés par des IOLTS. Cette hiérarchie est développée dans Tretmans [156] où plusieurs relations d'implantation sur des IOLTS sont étudiées : les relations de *test d'entrée-sortie* $\leq_{iot}$, de test d'entrée-sortie conforme **ioconf**, de refus d'entrée-sortie $\leq_{ior}$ et de refus d'entrée-sortie conforme **ioco**.

Ces relations sont toutes caractérisées par l'inclusion des *sorties* possibles après une trace mais se distinguent par l'ensemble de traces sur lequel la relation d'inclusion doit être vérifiée. La notion de *quiescence* est ici essentielle : un état est quiescent si aucune *sortie* n'est possible dans cet état. Pour faciliter la construction des relations de conformité, on va donc modéliser la quiescence par une transition spéciale étiquetée δ. Cette transition peut correspondre soit à un refus de l'ensemble des sorties possibles, soit comme une lettre terminale et peut-être prévue ou non dans la spécification. La distinction entre $\leq_{iot}$ et **ioconf** permet d'éviter la modélisation explicite de toutes les entrées dans la spécification. Celle-ci est alors un simple LTS étiqueté par les lettres d'entrées et de sorties et non pas un IOLTS pour lequel *toutes* les entrées sont toujours possibles. C'est l'approche la plus courante.

Autres relations Dans Petrenko *et al.* [130], les auteurs introduisent la notion d'équivalence de traces avec quiescence et file — *queued-quiescent trace equivalence*. Le testeur est ici divisé en deux parties, un *testeur des entrées* qui va fournir une séquence d'entrées à l'IUT et un testeur des sorties qui va valider les sorties produites par l'IUT en fonction de celles attendues dans la spécification, produisant un verdict `pass` ou `fail`. Le principe consiste à définir pour une séquence de test α donnée, les sorties possibles en fonction de l'atteinte d'états quiescents, états dans lesquels seules des entrées sont possibles.

Sous réserve que la spécification ne possède pas de cycle de sorties, on peut construire un ensemble de cas de test fini par énumération des séquences quiescentes-enfilées de taille inférieure à k où k est la taille de la plus longue séquence de sortie dans l'ensemble des traces quiescentes de l'état initial. La relation d'équivalence de traces quiescentes-enfilées est moins fine que la relation **ioco** définie dans Tretmans et Belinfante [155].

Cette idée est étendue pour permettre de distinguer les cas où des états quiescents intermédiaires peuvent apparaître pour une séquence α en introduisant des couples testeurs d'entrées-testeurs de sortie pour chaque sous-séquence de α menant à un état quiescent. On peut remarquer que cette construction revient à travailler sur une spécification qui est la clôture par une relation d'indépendance entre les entrées et les sorties situées entre deux états quiescents. D'autres relations sont définies dans le cadre du test réparti : elles sont décrites à la section 3.2.8.

3.2.6 Algorithmes de tests dans les IOLTS

Si les IOLTS fournissent une théorie compacte et globale du test, il n'en reste pas moins vrai que l'application de cette théorie de la conformité suppose de réaliser une sélection de tests comme pour les autres formalismes et modèles présentés précédemment.

TGV

Le principe de construction des cas de test dans TGV[75] est de réaliser un produit de synchronisation à la volée entre la spécification S et l'*objectif de test* TP. Ce dernier est un LTS acyclique complet sur

l'alphabet de la spécification dont les états sont marqués `Accept` ou `Reject`. Un objectif de test TP doit de plus posséder la propriété d'être contrôlable : si un état autorise un événement de sortie, alors aucun autre événement n'est permis. En d'autres termes, l'objectif de test TP doit être *déterministe au sens du test*. L'objectif de test représente une partie du comportement spécifié de l'IUT que l'on souhaite expressément vérifier. Les objectifs de test sont déterminés par le testeur en fonction de la stratégie globale de vérification du projet.

Les différentes étapes de l'algorithme de construction présenté dans J. C. Fernandez *et al.* [75] sont les suivantes :

1. construction du produit de mixage $TP \times S$. Ce produit synchrone est réalisé en construisant un *automate de suspension* : les différentes formes de quiescence que sont le blocage et l'attente d'une entrée sont transformées en boucles étiquetées par la lettre δ ;
2. construction d'un graphe dirigé acyclique représentant le cas de test avec un *préambule*, un *corps* et un *postambule*. Le préambule initialise le cas de test pour atteindre un état contenant des transitions de TP. Le postambule est un chemin de *reset*, permettant de revenir à l'état initial de S après avoir atteint l'état `Accept` dans TP. Au cours de cette étape, les conflits de contrôlabilité, c'est-à-dire l'existence d'au moins deux transitions de sortie à partir d'un même état, sont supprimés en élaguant les branches nécessaires ;
3. définition des verdicts associés à chaque transition :
   - `Pass` pour une transition vers un état du postambule tel que S est dans son état initial,
   - `(Pass)` pour une transition menant vers le postambule,
   - `Inconclusive` pour une sortie autorisée par la spécification mais non présente dans TP,
   - `Fail` pour toute transition de sortie non prévue par la spécification ;
4. décoration des états par des temporisateurs et des transitions δ pour les boucles de transitions internes. Un *temporisateur* est associé à chaque sortie possible dans un état et annulé quand la sortie n'est plus possible, soit parce qu'elle a eu lieu, soit parce qu'elle n'est plus autorisée par la spécification. Ces temporisateurs permettent de gérer les cas d'entrelacement des sorties.

La particularité de TGV est d'effectuer ces différentes étapes à la volée et donc de ne pas avoir à déplier l'ensemble du graphe de la spécification pour construire l'ensemble de test.

Étant donnée la spécification S présentée en partie gauche de la figure 3.6 et l'objectif de test TP en partie droite de la même figure, dont la fonction de transition est implicitement complétée par l'alphabet de $\mathcal{S}$, 3.7 représente leur produit de synchronisation. Ce produit de synchronisation est légèrement différent de celui réalisé par TGV car il contient encore des transitions internes marquées `^t`. Les états terminaux marqués d'un cercle blanc représentent les tests *réussis*, les états non coaccessibles — par exemple l'état 28 — représentent des tests *échoués* et les états intermédiaires des test *non-concluants*. L'exemple présenté sur ces différents schémas est tiré de J. C. Fernandez *et al.* [75].

Sous réserve d'une hypothèse d'*équité* — bornée – de l'implantation, cette construction permet de s'assurer de la validité d'une implantation même dans les cas de non-déterminisme de cette dernière. Tant que le résultat est non-concluant et que la borne maximale fixée par l'hypothèse d'équité n'est pas atteinte, le test est rejoué jusqu'à obtention d'un verdict définitif. La sélection repose donc sur le choix d'un objectif de test, solution que l'on retrouve par ailleurs dans d'autres approches basées sur ce même outil[50, 118]. Cette sélection est formalisée en une méthodologie par Martin [100] dans le cas du test d'*applets* JavaCard.

TORX

La génération des cas de tests dans l'outil TORX est basée sur les travaux théoriques de Tretmans [157]. Contrairement à TGV, la génération et l'exécution des cas de test sont effectuées concurremment selon un processus non-déterministe[156] dit aussi test *online*. La notion d'objectif de test est absente mais le processus peut être guidé par le testeur.

Le cas de test est un arbre étiqueté par l'alphabet de la spécification auquel on ajoute une transition de *quiescence* pour gérer les comportements de blocage. Il est construit récursivement par application non-déterministe des trois règles suivantes à partir d'un nœud étiqueté par l'état initial du LTS q_0 :

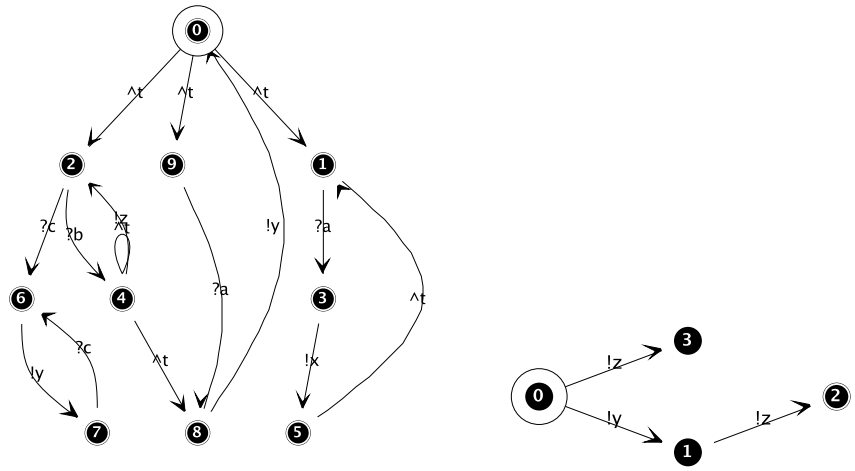


FIG. 3.6 – Spécification S & Objectif de test TP.

1. si le nœud courant est étiqueté par un état terminal, il est marqué `pass` ;
2. **une** branche étiquetée par une lettre de l'alphabet d'entrée apparaissant sur une transition depuis l'état courant est créée à partir du nœud courant et l'état atteint par la transition étiquette le nouveau nœud qui devient le nœud courant ;
3. pour chaque lettre de l'alphabet de sortie, on crée une branche étiquetée par cette lettre. Une branche étiquetée par la lettre δ est aussi créée. Le nœud créé par chaque branche est ensuite marqué :
   – si la lettre marquant la branche ne fait pas partie des transitions autorisées à partir de l'état étiquetant le nœud courant, le nœud est marqué `fail`,
   – sinon, l'état atteint étiquette le nouveau nœud qui devient le nœud courant.

3.2.7 Sélection de cas de tests pour les IOLTS

Une première approche est esquissée dans Tretmans [157], chapitre 6, où l'auteur propose un cadre théorique au problème de la sélection : une *fonction de valuation* permet d'attribuer à chaque suite de tests K une valeur dépendant de son pouvoir de détection de défauts. Cette fonction est pondérée par une partition de l'espace des états et des erreurs induites par les différentes suites de tests. Une *fonction de coût* est aussi calculée en raison de la taille des suites de tests générées, un arbitrage étant ensuite effectué par le testeur entre ces deux valeurs. Toutefois, aucune solution concrète n'est proposée pour le calcul, essentiel, de la fonction de valuation, sauf dans le cas d'une sélection basée sur un objectif de test.

Dans Alilovic-Curgus [9], une méthode de sélection de cas de tests basée sur la définition d'une *métrique* de l'espace des séquences de test est proposée. Il est alors possible de sélectionner des cas de tests parmi un ensemble donné assurant une certaine couverture qui est ici la distance maximale entre les séquences sélectionnées et l'ensemble des séquences possibles. Le principal inconvénient de l'algorithme proposé est qu'il nécessite de partir d'une suite de tests, par exemple un objectif de test.

Dans Pyhälä [134], Pyhälä et Heljanko [135], la sélection des cas de test se fait en utilisant un critère de couverture de la spécification, par exemple la couverture de transitions. L'algorithme présenté est similaire à l'heuristique de recherche locale présentée ci-dessus pour les FSM : l'algorithme cherche à maximiser les transitions couvertes par une recherche des transitions exécutables dans son voisinage et effectue un choix aléatoire dans le cas contraire. Kervinen et Virolainen [79] est un raffinement de ces principes : une fonction d'évaluation est associée à chaque transition courante et à chaque état en fonction d'un objectif de couverture, la valeur d'un état prenant par ailleurs en compte la valeur des états accessibles à partir

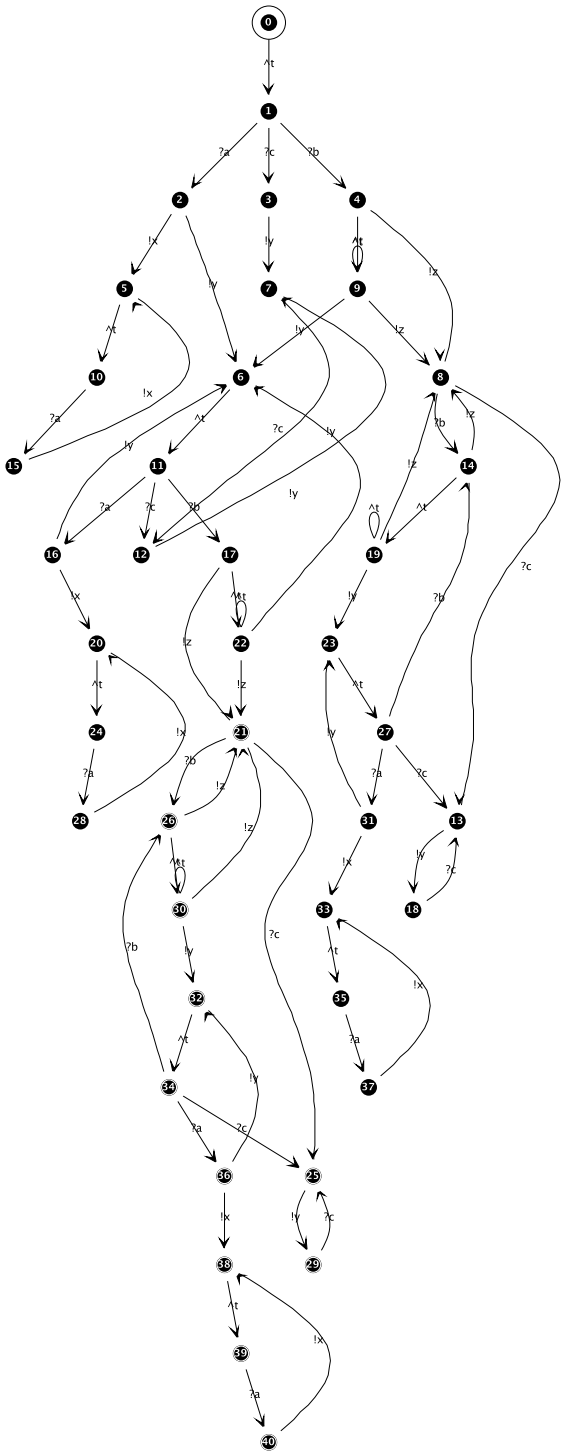


FIG. 3.7 – Synchronisation $TP||S$.

de celui-ci jusqu'à une certaine profondeur. Différentes stratégies de sélection sont ensuite possibles, une stratégie dite pessimiste qui considère que l'IUT sélectionne en priorité les transitions avec la plus faible valuation, une stratégie adaptative qui modifie la valeur des transitions en fonction des transitions déjà parcourues. Ces idées sont améliorées dans Feijs *et al.* [56] en prenant en compte la notion de distance et donc un calcul de couverture de l'ensemble infini des tests possibles par une propriété topologique de l'espace considéré.

Yannakakis [166] est une synthèse récente des approches heuristiques « modernes » pour la sélection des cas de test dans les IOLTS : il examine successivement la modélisation du problème comme un jeu à deux joueurs, comme un problème d'optimisation combinatoire et comme un problème d'apprentissage supervisé. L'auteur rappelle un certain nombre de résultats de décidabilité et de complexité sur ces différentes modélisations. La modélisation du problème du test comme un jeu est aussi à la base du test d'*Abstract State Machines* présenté dans Blass *et al.* [31].

Ces différents travaux s'inspirent de critères de couverture utilisés dans le test structurel (voir section 3.3) pour sélectionner de manière objective et contrôlée des suites de tests. Nous pensons que cette approche est, d'un point de vue pratique, très intéressante, surtout lorsqu'on cherche à intégrer les données dans le processus de sélection de cas de tests.

Gaudel et James [64], Lestiennes et Gaudel [89] unifient deux formalismes, les IOLTS et les spécifications algébriques dans une même démarche. Les spécifications sont ici des IOLTS dont les messages comportent des données vues comme des types algébriques abstraits. L'ensemble de test exhaustif est construit à partir d'un certain nombre d'hypothèses de testabilité de l'implantation :

- l'implantation est un IOLTS acceptant toujours toutes les entrées ;
- le non-déterminisme est limité par une hypothèse d'équité bornée : il existe un entier n tel que n exécutions d'un test assure que tous les choix non-déterministes auront été faits ;
- toutes les actions parallèles sont indépendantes ;
- l'implantation dispose d'un *reset* correctement implanté ;
- l'IOLTS représentant l'implantation est fortement convergent : il n'existe pas de chemin infini de transitions étiquetées par un événement interne.

L'ensemble de test exhaustif est alors défini comme l'ensemble des séquences de message de la spécification complétées par les sorties non autorisées — compte tenu de l'événement particulier δ dénotant l'absence de sorties. À partir de cette définition, on utilise une variante de l'algorithme non-déterministe présenté ci-dessus dans la section consacrée à TORX pour construire des cas de tests appartenant à cet ensemble.

La construction de la suite concrète de tests dépend en fait d'hypothèses faites sur les paramètres des messages. À partir d'une description symbolique, on va tout d'abord définir une longueur maximale de test (hypothèse de régularité) permettant de couvrir l'ensemble des *opérations* possibles sur le type des paramètres, c'est-à-dire tous les constructeurs du type de données en question. Pour améliorer la couverture des différents cas, on va déplier les opérations selon les différentes règles disponibles. Enfin, les valeurs sont instanciées par résolution des contraintes résultant des prédicats de gardes pour les différentes transitions non-autorisées, généralement par résolution d'une conjonction des négations des différentes conditions d'activation de transitions de sorties.

3.2.8 Le test réparti

S'il n'existe pas à notre connaissance de *théorie* du test de composants ou d'architectures logiciels, il existe par contre de nombreux travaux sur le problème du *test réparti*. Dans le test réparti, le testeur n'est plus monolithique mais peut interagir avec le programme testé au travers de plusieurs *points de contrôle et d'observation* ou PCO éventuellement répartis sur un réseau. On remarque que cette situation correspond au problème du test d'un composant pouvant offrir et requérir plusieurs interfaces et posséder plusieurs flots de contrôle plus ou moins indépendants.

On distinguera le *test réparti coordonné*, dans lequel les testeurs ont la possibilité de se synchroniser entre eux du *test réparti pur* dans lequel les testeurs ne disposent pas de cette facilité. La figure 3.8 représente schématiquement ces architectures de test : le trait pointillé entre les testeurs n'existe pas dans une architecture de test réparti pur.

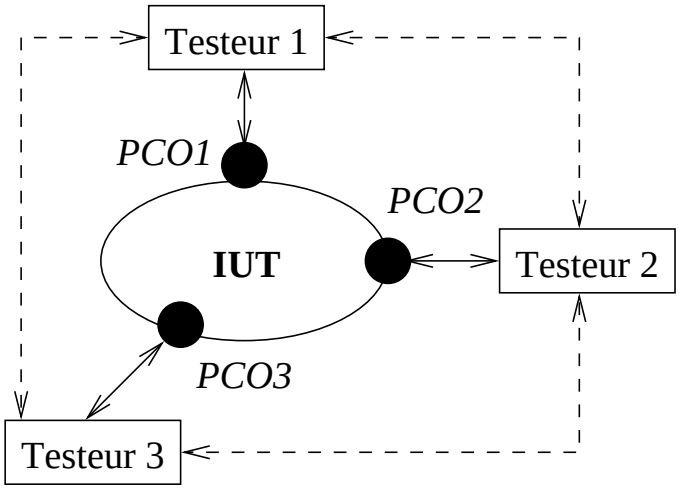


FIG. 3.8 – Architecture de test réparti.

Test réparti pur

Le test réparti pur a été étudié en particulier dans Hierons et Ural [71], Luo *et al.* [93], Sarikaya et von Bochmann [141] pour ce qui concerne le modèle des (E)FSM. Nous ne connaissons pas d'étude concernant ce problème appliqué au test d'IOLTS. Dans cette configuration, des défauts peuvent apparaître de par la nature partielle des observations et du contrôle que chaque testeur a de l'implantation sous test : il peut ne pas y avoir de lien direct entre une entrée sur un PCO donné et les sorties sur ce PCO, ce qui revient à dire qu'on ne peut pas toujours connaître l'état global du système testé à partir de ses états locaux, un problème classique de l'algorithmique distribuée.

Dans tous les cas, le système est représenté par un FSM dont l'alphabet d'entrée et de sortie est partitionné entre les différents PCO du système, les transitions peuvent donc mettre en œuvre des ports différents pour l'entrée et la sortie.

Luo *et al.* [93], Sarikaya et von Bochmann [141] cherchent à construire une séquence de test de type *tour de transition* qui sera dite *synchronisable* : une séquence est synchronisable si toutes les paires de transitions de la séquence sont synchronisables, c'est-à-dire si l'on peut toujours les ordonner localement sur un ou plusieurs ports. Il existe bien sûr des cas où une telle séquence n'existe pas, ce qui signifie que l'on ne peut pas toujours vérifier la conformité d'une IUT dans une architecture distribuée pure.

Sur le même modèle, Hierons et Ural [71] cherche à construire une séquence de vérification selon l'algorithme UIO. Un problème connexe à celui de la synchronisation est le décalage de sorties, dans le cas où une sortie correcte observable sur un port peut apparaître sur deux transitions consécutives dont l'une n'est pas dépendante d'une entrée sur le même port. La méthode présentée est de construire des séquences d'UIO synchronisables pour chacun des ports et chacun des états du système en transformant le graphe de la spécification initiale de manière à identifier les problèmes de synchronisation. Comme précédemment, la possibilité de construire un ensemble de test dépend fortement de la structure initiale de la spécification.

Test réparti synchronisé

Le cas du test réparti synchronisé a été comparativement plus étudié car il correspond à une situation plus favorable pour le testeur et réaliste dans un contexte de test en cours de développement. Cette architecture est similaire au test centralisé puisque les testeurs ont toute latitude pour reconstruire en communiquant un état global à partir de leurs états locaux.

Dans Cacciari et Rafiq [41], ce problème est étudié dans l'univers des FSM et le principal résultat est un algorithme permettant de synchroniser des testeurs locaux avec un nombre garanti minimal de messages échangés.

Dans le monde des IOLTS, Ulrich et König [159] introduit une sémantique concurrente non-entrelacée

pour l'ordonnancement des messages entre les différents testeurs. La construction de séquences de test est basée sur la construction de réseaux de Petri à partir de spécifications IOLTS puis sur le *dépliage* du réseau global obtenu à partir des spécifications locales de manière à obtenir un graphe dirigé acyclique des préfixes d'événements dépendants les uns des autres. Le dépliage s'arrête lorsque des configurations d'événements déjà atteintes sont rencontrées, les événements concurrents introduisant des points de branchement dans le dépliage. Cette construction est applicable uniquement en l'absence d'interblocage dans la spécification, une propriété qui peut éventuellement être vérifiée par *model-checking*. Le graphe de dépendance déplié peut ensuite être utilisé pour générer des séquences de test de conformité qui sont ensuite projetées sur les testeurs locaux pour exécution.

Ce même principe est appliqué dans Jard [77] pour étendre la génération de cas de tests de TGV au cas des testeurs répartis. Le formalisme utilisé est légèrement différent et plus complexe que dans Ulrich et König [159] mais surtout le principe du dépliage des événements concurrents est dirigé par l'*objectif de test*. De plus, les dépendances entre événements sur des testeurs différents sont contrôlées par l'insertion de messages de synchronisation entre testeurs comme dans Cacciari et Rafiq [41]. Notons que dans tous les cas cités, il est nécessaire de construire la spécification globale de l'IUT.

La formalisation de la notion de testeur réparti et des relations d'implantation correspondantes est détaillée dans Brinksma *et al.* [37], Heerink et Tretmans [70]. L'auteur définit un MIOTS, un IOLTS dont les alphabets d'entrée et de sortie sont partitionnés sur un ensemble de canaux — les PCO —, et une relation d'implantation **mioco** similaire à **ioco** — un préordre sur les ensembles de traces-refus —- mais paramétrée par la partition des alphabets.

Bien que nous ayons remarqué l'absence d'outils théoriques propres au test de composants architecturaux, van der Bijl *et al.* [160] s'est intéressé au problème du test de *composants* en général dans le cadre de la relation **ioco** et des IOLTS. Il montre que sous l'hypothèse de certaines restrictions, il n'est pas nécessaire de retester le résultat de la composition de deux composants qui sont **ioco** conformes à leurs spécifications respectives. La composition est ici entendue au sens d'un produit de synchronisation des alphabets internes aux deux composants suivi d'une projection masquant les lettres synchronisées.

3.3 Test structurel & Critères de couverture

Nous présentons dans cette section une synthèse des techniques du test structurel. Ces techniques ne constituent pas l'objet principal de notre étude mais il nous a paru intéressant de les évoquer pour au moins deux raisons. Premièrement, parce qu'il s'agit des techniques les plus couramment utilisées dans le domaine du test de logiciel. Et deuxièmement parce que l'on constate une convergence entre les deux approches, les notions de critères de test et de couverture étant de plus en plus utilisées pour sélectionner des cas de tests fonctionnels (*cf.* section 3.2.7).

Un critère de couverture est un objectif de test qui est basé sur une mesure de couverture d'un certain nombre de traits du modèle de l'objet testé, généralement le code source. C'est historiquement un concept essentiel et qui a donné lieu à un nombre considérable de travaux théoriques et pratiques sur la définition de critères pertinents — on dira aussi *adéquats* — et sur les comparaisons entre critères. C'est aussi bien souvent le seul critère objectif dont on dispose en pratique pour définir un objectif de test.

Goodenough et Gerhart [66] est certainement un des articles les plus cités dans le domaine du test : il fonde l'ensemble de la théorie de la sélection des cas de tests en introduisant les concepts fondamentaux de *fiabilité* d'une suite de tests et d'*adéquation* par rapport à un critère.

Une suite de tests pour un programme sur un certain domaine d'entrée est dite *fiable* si lorsque la suite de tests ne produit aucune panne, le programme ne contient aucune erreur. Une suite de tests pour un programme sur un certain domaine d'entrée est dite *adéquate* si la suite de tests est capable de détecter tous les programmes produisant des sorties incorrectes pour tous les éléments du domaine d'entrée. Ces notions sont fortement similaires aux propriétés attendues des suites de tests fonctionnelles.

Typologie des critères

Les critères de couverture peuvent être basés sur le graphe de contrôle du modèle de l'IUT, sur le graphe de flots de données, sur la structure de l'espace d'entrée et de l'espace de sortie, ou une combinaison

```
public boolean verifyUIO(Map m) {
    if(m.size() != states().size())
        return false;
    List[] words = new List[m.size()];
    words = (List[])
         m.values().toArray(words);
    final int len = words.length;
    for(int i=0;i<len;i++)
        for(int j=i+1;j<len;j++)
            if(words[i].equals(words[j]))
                return false;
    return true;
}
```

FIG. 3.9 – Code source Java de la figure 3.10.

quelconque de l'un de ces critères. Zhu *et al.* [168] propose une classification des critères de test et des analyses de leur efficacité, comprise comme leur capacité à détecter des erreurs. Ces analyses se basent en particulier sur les travaux de Frankl et Weyuker [59] qui définissent une relation de subsomption entre ensembles de critères de test.

Les critères basés sur la structure du flot de contrôle du programme vont de la couverture des instructions — *all-statements* — à la couverture des chemins — *all-paths*, en passant par la couverture du nombre cyclomatique, la couverture des branches — *all-edges*, la couverture des conditions et décisions — *Multiple Condition Decision Coverage* ou MCDC.

La figure 3.10 est une représentation du graphe de contrôle d'une méthode Java obtenu à partir du code intermédiaire. Ce graphe correspond au code source de la figure 3.9 qui est une méthode qui vérifie pour un FSM que chaque séquence associée à chaque état est bien une séquence UIO (voir ci-dessus 3.2.3).

À partir du graphe présenté, on peut donc définir en fonction de différents critères de couverture des suites de tests qui sont des ensembles de chemins dans le graphe de contrôle. Il restera pour chacune de ces séquences à définir les valeurs de variables permettant de réaliser effectivement la séquence choisie, ce qui est un problème souvent ardu.

Par exemple, l'ensemble $\{(2,3,5,7,9,11,5,7,9,10,12,9,10,13),(2,4),(2,3,5,8)\}$ est potentiellement adéquat pour les critères *tous-les-nœuds* et *tous-les-arcs* du graphe de contrôle de la figure 3.9. Il reste à montrer qu'un certain ensemble des valeurs des variables d'entrée permet de sensibiliser ces chemins.

Les critères basés sur le flot de données s'intéressent à la couverture des relations entre la définition d'une variable et son utilisation. Une hiérarchie de critères basée sur la notion de chemin *def-use* est aussi construite. La figure 3.11 est une représentation d'un graphe de flot de données simplifié pour le graphe de contrôle 3.10. Les nœuds sont des utilisations ou des définitions de variables — marquées Use i ou Def i — et les arcs sont une extension des arcs du graphe de contrôle permettant de relier chaque nœud.

Nous n'avons pas ici distingué les nœuds de type *p-use* — utilisation d'une variable pour un prédicat de branchement conditionnel — des nœuds *c-use* — utilisation pour une instruction, un appel ... Vincenzi *et al.* [162] proposent un outil et une méthode pour le test de couverture de flot de contrôle et de données pour programmes Java. Comme dans les exemples présentés figures 3.10 et 3.11, le principe est d'extraire ces graphes du *code-octet* représentant un programme Java.

Le problème principal lié à l'utilisation de critères de ce genre consiste bien évidemment à construire un jeu de tests permettant de respecter le critère ce qui n'est pas toujours possible et est même indécidable dans le cas général. La figure 3.12 présente une vue partielle de cette hiérarchie des critères de test pour quelques critères courants.

Ce problème de *sensibilisation* des chemins d'exécution possibles du programme a été déjà abordé dans Boyer [34] comme une forme dérivée de la preuve automatique de théorème. Le principe de SELECT est de réaliser une exécution symbolique du programme testé pour construire des ensembles de chemins d'exécution ayant la forme de prédicats sur les variables du programme. Les systèmes de contraintes ainsi construits sont ensuite résolus pour fournir des valeurs d'entrée permettant de sensibiliser chaque chemin. Ce principe a été repris dans Gotlieb *et al.* [67] et dans l'outil InKa dérivé de ces travaux, en utilisant un

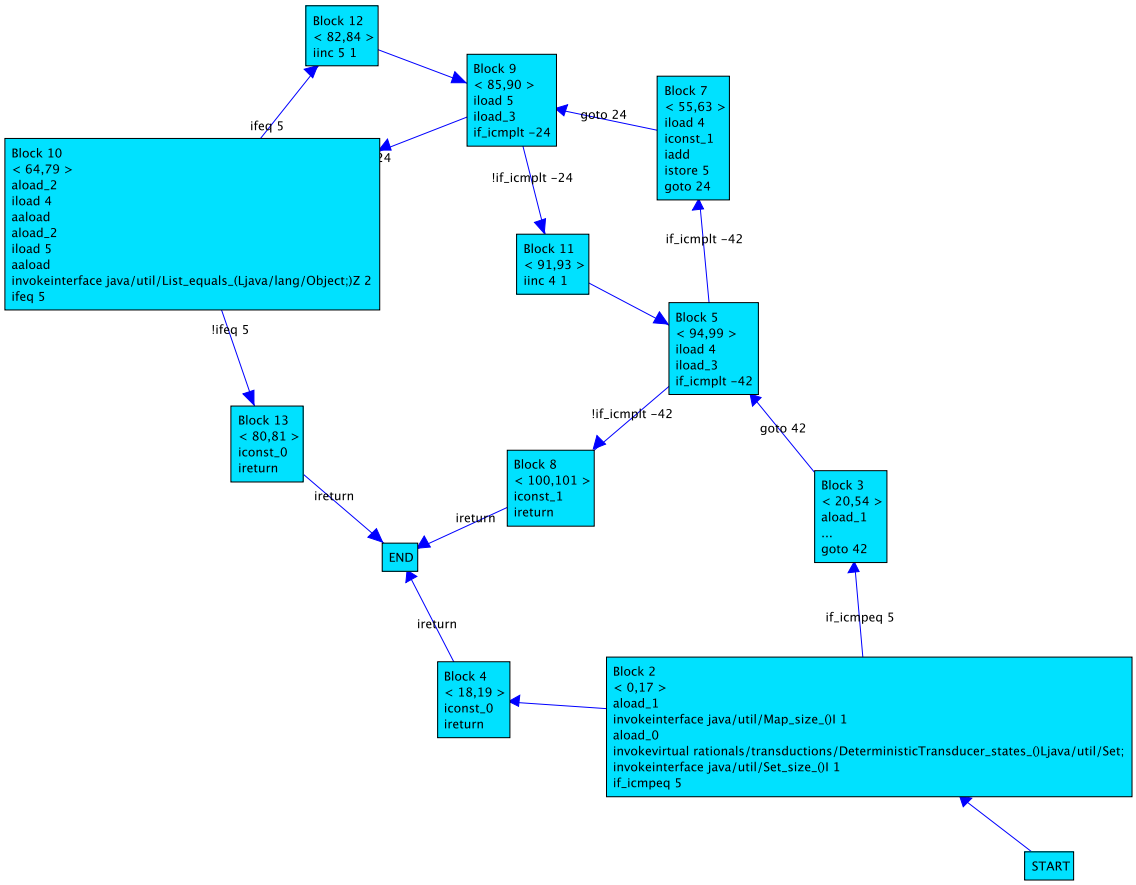


FIG. 3.10 – Exemple de graphe de contrôle.

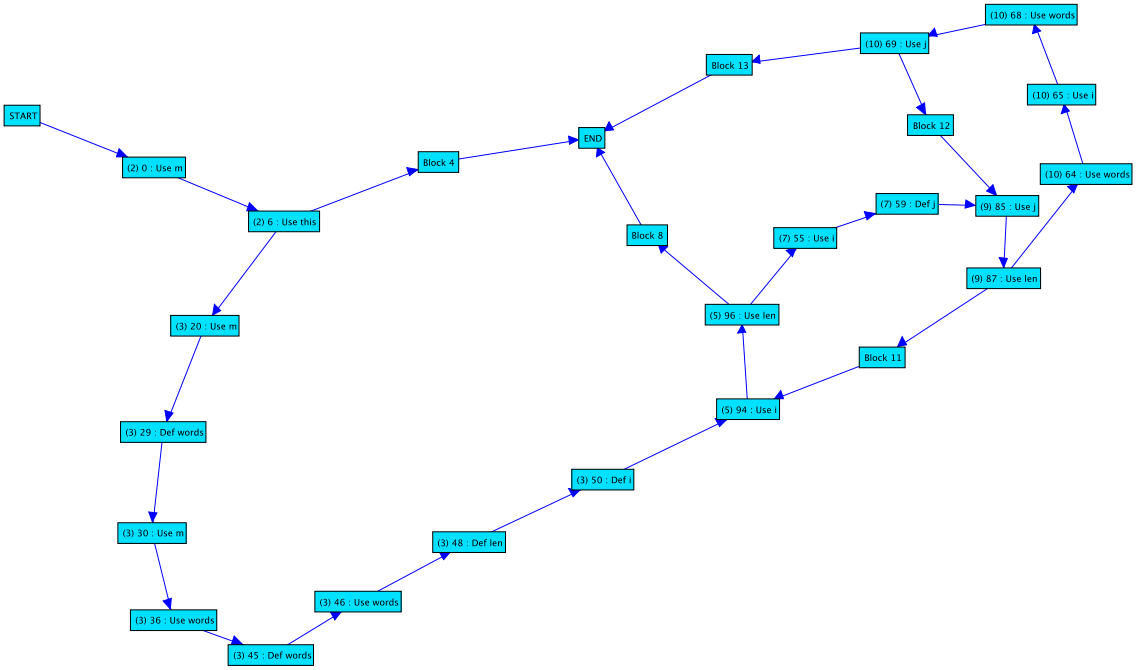


FIG. 3.11 – Exemple de graphe de flot de données.

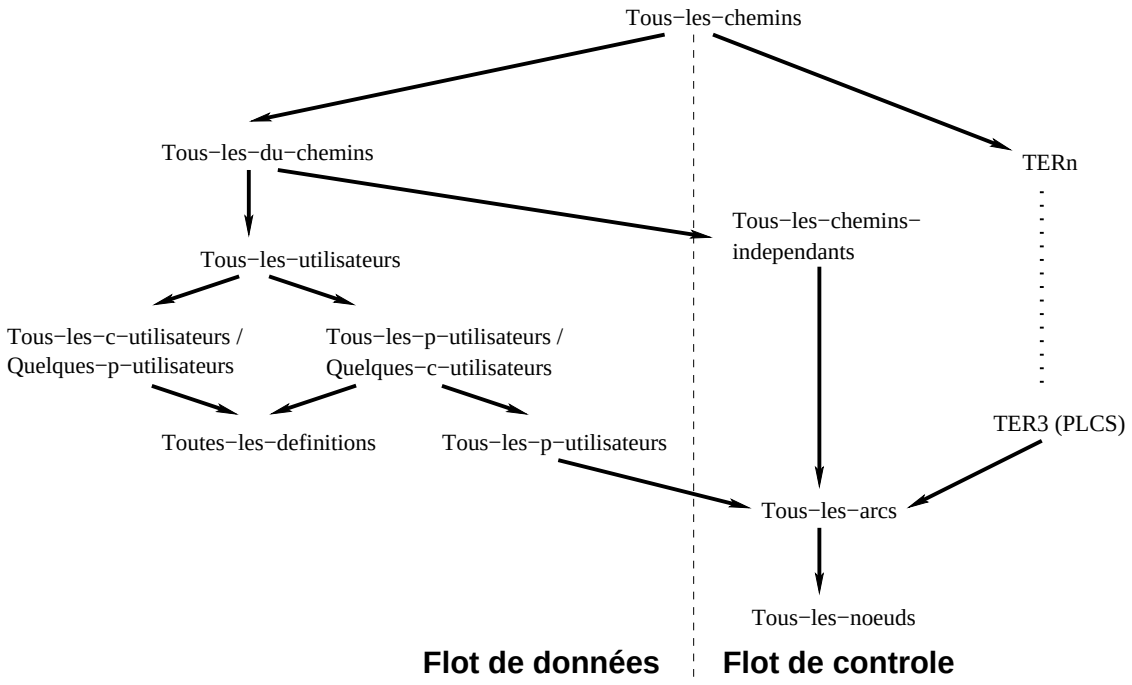


FIG. 3.12 – Hiérarchie usuelle des critères de couverture.

système de résolution de contraintes.

On peut aussi utiliser d'autres solutions algorithmiques pour rechercher des valeurs permettant de sensibiliser un chemin particulier comme par exemple l'algorithme génétique de Pargas *et al.* [128] : un vecteur de valeurs d'entrée est optimisé en fonction d'un objectif — chemin ou état — et du nombre de prédicats que le vecteur de valeurs permet de satisfaire — fonction de *fitness*. McMinn [104] est une étude des différentes techniques de recherche heuristiques utilisées pour la découverte de données permettant de sensibiliser des chemins dans un graphe de contrôle.

Analyse de domaine

Pour sélectionner les cas de tests, on peut aussi s'intéresser aux caractéristiques de l'espace d'entrée ou de sortie du programme et définir un objectif en fonction de celui-ci et d'une *partition* du domaine en sous-domaines permettant de sensibiliser tel ou tel chemin ou de valider tel critère. Une tactique usuelle consiste à s'intéresser aux valeurs limites de chaque sous-domaine, sous l'hypothèse que ce sont ces valeurs qui seront le plus pathogènes. Dans le cas des domaines numériques entiers, cette stratégie permet classiquement de détecter des erreurs d'utilisation de relations entre valeurs, un $\leq$ se transformant par exemple en un $<$. Kosmatov *et al.* [80] présente une formalisation de ce critère au cas de spécifications formelles de type Z ou B. Les partitions de domaines sont déduites des prédicats sur les variables de la spécification et résolues par approximation dans un domaine continu — par exemple les réels — puis retransformées dans le domaine — discret d'origine. La notion de partitionnement de domaine est aussi sous-jacente dans les hypothèses de régularité et d'uniformité introduites dans Bernot *et al.* [26].

3.3.1 Sélection statistique & aléatoire

Dans la première partie de ce chapitre, nous avons introduit la notion de *profil opérationnel* comme un outil d'aide à la sélection des tests dans le test dit système. Un profil opérationnel est simplement une distribution probabiliste de l'espace d'entrée de l'IUT. Une tactique de test évidente consiste donc à échantillonner cette espace d'entrée pour l'utiliser comme donnée de test. Un modèle statistique permet d'interpréter les résultats du test pour évaluer un niveau de fiabilité du logiciel testé. Bernot *et al.* [27]

propose de formaliser le test probabiliste fonctionnel, de manière similaire à l'ingénierie de la fiabilité[96], en calculant à partir d'un échantillonage des valeurs de l'espace d'entrée une probabilité de correction — pour des tests réussis — en fonction d'une marge d'erreur et d'un intervalle de confiance.

Thévenod-Fosse et Waeselynck [152] définit une notion de qualité de test en fonction d'un critère de test et de la taille de la suite de tests sélectionnée par échantillonage sur une distribution de l'espace d'entrée construite en fonction du critère de couverture à atteindre. Il ne s'agit plus ici d'un profil opérationnel correspondant à un usage attendu de l'IUT mais d'un *profil de test* spécifiquement construit en fonction d'un objectif prédéfini et permettant d'évaluer la qualité du résultat produit de manière statistique. Bien sûr, la fiabilité globale du test réalisé reste dépendante de la confiance que l'on place dans le critère choisi, avec cette différence par rapport au cas déterministe que les cas de tests sont sélectionnés aléatoirement et donc sans le biais d'une sélection arbitraire. Cette technique est développée dans Denise *et al.* [53], en utilisant des techniques de génération aléatoire de structures combinatoires, pour le test statistique basé sur des spécifications prenant la forme de graphes.

Lorsque la distribution est uniforme, on a alors un *test aléatoire*. Ntafos [121], Thevenod-Fosse *et al.* [153] montrent expérimentalement que ce n'est pas forcément l'approche la moins efficace pour la détection des erreurs.

3.3.2 Test mutationnel

Le test mutationnel — *mutation testing* en anglais — constitue une technique indirecte pour sélectionner une suite de tests pertinente. Cette technique a été introduite dans DeMillo *et al.* [51]. Son principe est d'évaluer la fiabilité de la suite de tests par rapport à un *modèle de faute* :

- on définit pour un langage ou un format binaire donné un ensemble d'*opérateurs de mutations* atomiques : inversion de relations binaires, changements de variables, transformations d'opérateurs arithmétiques, modifications de constantes en variables ... Ces opérateurs de mutation représentent des erreurs courantes que peut introduire un *programmeur compétent* dans un logiciel et constituent donc un modèle des fautes que la suite de tests va rechercher ;
- on applique ces opérateurs de manière « systématique » sur le logiciel testé, produisant ainsi des *mutants* qui sont des versions du logiciel proches de l'original mais généralement incorrectes. Chaque mutant diverge de son parent par une seule opération de mutation selon le principe du couplage des erreurs : une suite de tests révélant des erreurs simples sera capable de révéler des erreurs complexes. Autrement dit, les erreurs complexes sont issues de plusieurs erreurs simples (voir Offutt [124] pour une discussion expérimentale de la question). Le problème des mutants équivalents est loin d'être trivial mais est négligé en pratique ;
- on écrit ou enrichit une suite de tests *tuant* le maximum de mutants : lorsqu'elle est exécutée sur les mutants, elle révèle qu'ils sont incorrects en produisant l'échec du test ;
- le *score de mutation* de la suite de tests est égal au nombre de mutants tués sur le nombre total de mutants non-équivalents.

Ce score de mutation est une mesure de la qualité de la suite de tests et donc de la qualité du logiciel : si le logiciel passe la suite de tests dont le score de mutation est élevé, c'est donc qu'il ne contient pas les fautes injectées par le test de mutation. Implicitement, cette méthode se base sur deux hypothèses, l'*hypothèse du programmeur compétent* : le logiciel testé est *a priori* presque correct et les erreurs de programmation sont simples ; et l'*hypothèse du couplage* : les erreurs complexes sont des suites d'erreurs simples.

On voit bien que la fiabilité de cette approche dépend du choix des opérateurs de mutation et de leur adéquation avec les erreurs susceptibles de se produire réellement, ainsi que du nombre et de la répartition des mutants générés. Ce dernier point la rend d'ailleurs très coûteuse en temps d'exécution et de réalisation des tests. Dans Baudry [24], cette approche est utilisée couplée avec un algorithme génétique pour optimiser des suites de tests.

Le test mutationnel a une littérature relativement abondante et a donné lieu à la création d'une faune variée d'opérateurs de mutations, en particulier dans le cadre des langages-objets. Cette technique est très utilisée pour comparer des méthodes de sélection de cas de test entre elles.

3.4 Discussion

On a vu que le test logiciel, même restreint au seul cas du test de systèmes à états-transitions, offre un champ extrêmement vaste de travaux qui se sont toutefois essentiellement dirigés dans deux directions : d'une part la sélection efficiente de suites de tests dans le cas des FSM, d'autre part la définition de relations de conformité pertinentes dans le domaine des IOLTS.

Nous avons aussi vu que le problème de la sélection des cas de test se ramenait essentiellement au problème de la définition d'un *objectif de test*, au sens large du terme, dépendant des artefacts à la disposition du testeur, et qu'il existait un nombre important de critères heuristiques difficilement comparables entre eux.

3.4.1 Test basé sur les modèles

Nous nous sommes concentrés dans l'exposé de cet état de l'art sur des méthodes *primitives* de test. Ces méthodes constituent la base d'autres méthodes utilisant des modèles de plus haut niveau tels que les diagrammes UML ou, plus rarement, les ADL.

Il existe une littérature importante consacrée à l'application de ces méthodes au cas de diagrammes UML et plus particulièrement des Statecharts[69, 72, 73, 125, 126, 132, 144]. La question des tests d'intégration, et de leur ordonnancement à partir de modèles UML, est abordée dans Hartmann *et al.* [69] et Jeron *et al.* [78]. Nebut *et al.* [118] est une approche basée sur les cas d'utilisation et les contraintes associées. À partir d'un diagramme de cas d'utilisations enrichi de prédicats et de variables, et des dépendances entre cas d'utilisations, on construit un diagramme d'états finis qui est injecté dans TGV pour produire une suite de tests systèmes. Briand et Labiche [36] a une approche similaire basée sur l'utilisation de contraintes OCL.

Muccini *et al.* [116] propose une méthode d'extraction de cas de test unitaires à partir d'une description d'architecture dont le comportement est modélisé sous forme de LTS. Ce travail est dans l'idée proche de notre démarche : il s'agit de construire des tests de composants à partir d'une spécification d'architecture. Dans le détail, les méthodes diffèrent : le test n'est pas ici conduit de manière systématique mais à partir de la définition d'un *objectif de test*, le critère de sélection choisi est basé sur la complexité de McCabe[102], les tests abstraits dérivés doivent être manuellement « traduits » en tests exécutables en fonction du composant testé. A.Hoffmann *et al.* [6], Batteram *et al.* [23] présentent une architecture et un outil pour le test de composants CCM. La question du problème de la sélection des cas de tests n'est toutefois pas abordée. Cavalli *et al.* [44] est une application aux objets CORBA de l'algorithme *Hit-Or-Jump* présenté à la section 3.2.4.

3.4.2 Analyse

Lai [82] souligne la distance qui existe entre la foultitude de modèles et d'algorithmes produits par le monde universitaire et l'état concret des pratiques dans l'industrie :

> « [...] this state-of-the-art research is not necessarily state-of-the-practice. Academic methods are seldom employed in industry. [...] There is not much progress in the use of test sequence generation techniques for practical testing of communication networks. »

La portée de cette affirmation doit toutefois être modérée par le fait que l'auteur n'envisage qu'une classe de systèmes de transitions dans son étude, les FSM. Il n'en reste pas moins vrai que l'activité du test dans le monde industriel est encore largement « manuelle ». La situation est identique dans le monde des services logiciels comme on l'a vu au chapitre 1.

Les notions d'équivalence dans les FSM et les IOLTS sont fondamentalement différentes. Dans le premier cas, on a une équivalence entre des *fonctions* : le FSM est un transducteur calculant une fonction à partir de paramètres d'entrée. Le but du test de conformité de FSM est de vérifier ce calcul en parcourant l'espace des valeurs du paramètre d'entrée qui est ici un *langage*. Dans le second cas, on vérifie une équivalence entre des *langages*. Les suites de tests ont donc des statuts et des structures différentes : un ensemble de séquences d'entrée d'une part, un ensemble de mots du langage d'autre part. Et le processus de test est bien sûr différent : sensibilisation du FSM d'une part, synchronisation des deux langages d'autre part.

Le panorama que nous avons fait dans ce chapitre nous permet de mieux nous orienter dans l'approche choisie. On voit clairement que notre objectif est similaire à celui du test de conformité réparti pour des systèmes de transitions à entrée-sortie (section 3.2.8). Ces travaux s'appuient sur les relations de conformité définies à la section 3.2.5. Ces relations, et en particulier la relation **ioco**, ne sont toutefois pas satisfaisantes dans notre contexte : elles ne tiennent pas compte de la nature *contractuelle* des spécifications de composants que nous utilisons ni de l'asymétrie existant dans les composants entre les interfaces requises et les interfaces fournies.

Par ailleurs, notre approche de la modélisation des composants est fondée sur les automates et la théorie des langages. Il est donc naturel que nous choisissions d'exprimer la notion de conformité en termes d'une relation d'équivalence sur des langages et par conséquent que les tests permettant de vérifier cette relation soient réalisés sous la forme d'un produit de synchronisation entre les langages.

Chapitre 4

Modèle de composants FIDL

L'analyse des modèles et outils existants permettant de définir et réaliser des architectures de composants nous a permis de dégager les concepts fondamentaux relatifs à ce domaine : composants, interfaces, ports et connexions. Nous avons constaté cependant qu'il existait une certaine distance entre des modèles permettant de *concevoir* et vérifier des propriétés sur des architectures de composants, schématiquement les *langages de description d'architecture* formels, et d'autre part les plate-formes et implantations concrètes permettant de construire et exécuter des systèmes de composants. Notre objectif est d'être capable non seulement de raisonner sur des modèles mais aussi d'utiliser ces spécifications pour *vérifier* des implantations concrètes de modèles, en particulier par le test. Par ailleurs, on souhaite que ces modèles soient suffisamment abstraits de l'implantation pour être *réutilisables*, ce dans l'optique de construire une démarche formalisée d'*ingénierie dirigée par les modèles*. Le but final est bien entendu de découpler la modélisation des processus métiers de la conception des systèmes les réalisant.

Dans ce chapitre, nous présentons en détail mais de manière informelle un *modèle de composants abstraits* nommé FIDL pour *Formal Interface Definition Language*. Après quelques généralités sur les notions de composants, la seconde partie de ce chapitre est consacrée au langage FIDL. Nous illustrons cette présentation par quelques exemples significatifs de modélisation.

4.1 Généralités

Un composant est une *unité architecturale* communiquant avec son environnement au travers de *ports*. Un système est constitué par l'interconnexion d'un ensemble de ports de composants compatibles. Un composant est considéré comme une *boîte noire* dont l'état interne ne peut être observé qu'au travers des messages échangés avec les autres composants. Autrement dit, nous ne faisons aucune hypothèse dans le modèle sur les modalités d'exécution des éléments d'un système.

Les composants fournissent et utilisent des *services* au travers des ports synchrones typés par interfaces. Les interfaces fournies sont appelées *facettes*, celles utilisées sont appelées *réceptacles*. Les composants peuvent également communiquer en utilisant des *événements asynchrones*, les *sources d'événements* sont des ports utilisés pour émettre des événements, les *puits d'événements* sont les ports par lesquels les composants reçoivent les événements asynchrones et chaque port d'événement asynchrone définit le type — la structure — des événements susceptibles de transiter par ce port. Nous utiliserons le terme de *message* pour désigner collectivement toutes les occurrences d'événements asynchrones, d'appels, de retours ou d'exceptions de méthodes. Le schéma 4.1 est la représentation d'un composant introduite dans Marvie et Merle [101] et très proche de celle utilisée dans le CCM.

4.1.1 Interfaces & ports synchrones

Une interface FIDL définit un *protocole* de communication entre deux composants au travers d'une *connexion*, protocole défini sous la forme d'*opérations* — ou méthodes — et d'*attributs* susceptibles de générer des appels, des retours et des exceptions. Une opération est identifiée par son nom, le type et la

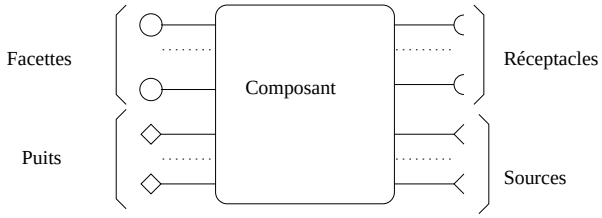


FIG. 4.1 – Composant.

modalité — `in`, `inout` et `out` — de ses paramètres. Le cas des attributs est particulier : ils ne sont accessibles que par l'intermédiaire d'*accesseurs* et de *mutateurs*, c'est à dire des opérations implicites nommées `getXXX` et `setXXX` respectivement pour chaque attribut `XXX`.

Une interface peut hériter d'une ou plusieurs autres interfaces ce qui ne pose pas de problèmes de résolution statique : l'absence de corps de méthode n'introduit pas les conflits d'héritages typiques des langages à héritage multiple tels que le C++. L'héritage pose toutefois un certain nombre de problèmes lorsque l'on prend en compte la spécification formelle des interfaces et cette question sera étudiée dans la section 6.4 du chapitre 6.

Une interface définit le *type* d'une facette ou d'un réceptacle susceptible d'être déclaré dans la définition d'un composant. Ce type est un élément permettant de vérifier la validité d'une connexion entre facette et réceptacle : la facette doit être un sous-type du type du réceptacle pour que la connexion soit, au moins syntaxiquement, considérée comme légale.

Enfin, il est possible de définir des modalités de connexion pour chaque facette :

- *unique* si le port ne peut être utilisé que par un et un seul composant à la fois (c'est le cas par défaut) ;
- *single* si le même port peut être partagé par plusieurs composants connectés ;
- *multiple* si chaque composant connecté à ce port est *isolé* des autres composants. Le port est en fait multiplexé et chaque connexion possède son propre état conversationnel indépendant de toutes les autres connexions.

Cette notion de *modalité* d'interface permet d'intégrer dans la spécification les typologies de modèle d'exécution du composant qui sont précisées, dans le cas du CCM et de J2EE, par des descriptions externes. On dispose ainsi des notions familières d'objet *session*, *processus* ou *méthode*, transposées des composants à leurs ports et disponibles dans le modèle, et un même composant pourra donc offrir simultanément des facettes de différentes modalités. Par la suite, nous considérerons uniquement le cas des facettes de type *unique*. Le cas des facettes *single* introduit le partage de facettes entre plusieurs composants et ruine la possibilité de composer les composants en ne tenant compte que de leur *topologie de connexion*. Il impose donc pour valider une architecture de vérifier le respect des contrats en fonction de l'ensemble des composants connectés et ne permet pas de raisonner uniquement de manière compositionnelle. C'est une version du problème bien connue de l'*aliasing*. En ce qui concerne les facettes *multiple*, leur traitement n'introduit pas de difficulté conceptuelle supplémentaire mais est techniquement assez complexe : le système que nous décrivons et la sémantique des langages de traces étant déjà touffus, nous avons choisi de reporter le traitement de ce cas à des travaux ultérieurs.

4.1.2 Événements & ports asynchrones

Les ports d'événements asynchrones, *sources* et *puits* sont typés par le type d'événement — un objet-valeur — qu'ils sont susceptibles de consommer ou de produire. Ces objets-valeur — ou `eventtype` — sont des structures de données arbitrairement complexes. En IDL3, un `eventtype` peut inclure des définitions d'attributs, d'opérations, des types exportés, etc... comme n'importe quel autre espace de nommage. C'est en fait un véritable *objet*, au sens des langages objets, dont l'état peut être observé et éventuellement modifié par n'importe quel élément du système.

Dans un premier temps, nous ne nous intéresserons aux événements qu'en tant que structure de données transportée par des ports asynchrones.

4.1.3 Connexion & composition

Une connexion est une relation un-vers-un entre ports de composants *compatibles* : un réceptacle est connecté à une facette et un puits est connecté à une source.

Un *composite* est un ensemble de composants partiellement ou totalement inter-connectés, autrement dit un composant réalisé par *assemblage* de plusieurs autres composants. Nous utiliserons le terme de *système* pour désigner un ensemble quelconque de ports sans identité précise.

Du point de vue de la spécification, un composite peut permettre de masquer certaines facettes et a pour effet principal de rendre inobservable l'ensemble des messages transitant par les connexions réalisées entre ses composants. On notera que la notion de composite, conceptuellement simple et élégante, n'est pas aujourd'hui prise en compte directement dans les plate-formes existantes, à l'exception de .Net sous la forme des *assembly*. Elle est par contre essentielle dans la plupart des modèles formels étudiés.

Les connexions sont réalisées durant la phase de déploiement et peuvent normalement évoluer au court du cycle de vie du composant. Nous considérons ici le cas simple où les connexions des composants sont établies au *déploiement* et ne sont plus modifiées jusqu'à l'arrêt du système.

4.1.4 Exécution & Communication

Ainsi que nous l'avons dit précédemment, aucune hypothèse n'est faite quant aux flots d'exécution des composants. Nous nous intéressons uniquement aux messages échangés entre les différents composants au travers de leurs connexions. Ces messages sont nommés et contiennent des données primitives ou structurées selon le type des interfaces du port concerné. De plus, ces messages identifient de manière unique l'émetteur et le récepteur.

Le médium de communication est supposé fiable : tous les messages envoyés atteignent leur destination en un temps fini, mais nous ne supposons rien quant à l'ordre d'arrivée des messages entre différentes connexions qui peuvent être arbitrairement entrelacées. Sur une même connexion point-à-point, l'ordre des messages est supposé maintenu entre l'émetteur et le récepteur.

Les événements observables d'un système sont les messages échangés entre ses composants au travers de leurs connexions, sachant que l'émission et la réception d'un message constituent deux événements distincts observés à chacun des points de la connexion. Chaque composant possède son propre point de vue sur le système et donc ne peut observer qu'un sous-ensemble des événements globaux, en l'occurrence les messages reçus et émis sur les connexions auxquelles il participe.

4.2 Le langage FIDL

Le langage FIDL est construit par agrégation de plusieurs langages, chacun destiné à définir une partie du système spécifié :

1. le langage IDL3, défini dans le CORBA Component Model[122], permet de décrire les *interfaces* et la structure des *types* de données permettant à un système de communiquer avec son environnement. Cette description indépendante de tout langage de programmation est destinée à être *projetée* dans un langage spécifique puis complétée par l'environnement d'exécution ;
2. le langage FIDL proprement dit, qui décrit le comportement des interfaces et composants IDL3 en termes d'ensemble de traces ;
3. le langage Jaskell, un langage fonctionnel sans effets de bord à évaluation paresseuse, sous-ensemble de Haskell 98, permettant de définir des fonctions sur les types IDL3 et donc de préciser le contenu des messages circulant dans le système.

La partie spécification — FIDL et Jaskell — est incluse dans la description IDL3 sous la forme de commentaires et est donc transparente pour les outils existants de manipulation de modèles IDL3. L'ensemble de la spécification couvre donc les éléments *structuraux*, *comportementaux* et *calculatoires* du système décrit.

L'objectif du langage FIDL est que, en utilisant une seule et même notation, l'on puisse tester, valider et vérifier des composants et assemblages selon différentes technologies, voire même entre plate-formes

hétérogènes. La notation IDL3 est celle qui nous a paru offrir la meilleure couverture des concepts que nous souhaitions voir apparaître et nous l'avons donc choisie comme outil de description structurelle d'une spécification. On verra au chapitre 8 comment ce langage est implanté concrètement dans un outil et comment des applications vers diverses plate-formes techniques peuvent être mises en œuvre.

La notation FIDL s'appuyant sur le langage IDL3, nous en rappellerons ici les principaux éléments avant de détailler la partie proprement originale de FIDL. Le lecteur souhaitant plus de détails pourra consulter les documents de référence de l'OMG tels que Object Management Group [122, 123]. La grammaire EBNF complète du langage FIDL est donnée en annexe A.

L'ensemble de ces langages est fondu dans la sémantique sous la forme d'un seul et même système formel détaillé dans la section 6.1. L'utilisation de ces syntaxes particulières est donc tout à fait contingente et nous avons la possibilité de substituer toute notation équivalente pour l'un ou l'autre de ces éléments.

4.2.1 IDL3

Le langage de description d'interface IDL de CORBA et son extension propre au modèle de composants CCM est une *lingua franca* entre langages de programmation et systèmes de communication permettant d'assurer une interopérabilité au niveau de granularité le plus fin, celui des données et des appels de procédures distants, tout en assurant un minimum de structuration orientée-objet. Un fichier IDL a pour vocation d'être utilisé par un outil qui interprétera son contenu pour produire un ensemble d'éléments dans un certain langage de programmation permettant d'assurer la communication avec un *Objecte Request Broker* donné. Cette opération est souvent appelée *projection* même si elle s'apparente plutôt à une traduction.

Le langage IDL3 permet de définir les éléments suivants :

1. des *modules*, espaces de nommages indépendants et arborescents ;
2. des *types* de données variés : types numériques, structures, unions ou énumérations ;
3. des *interfaces*, pouvant hériter d'une ou plusieurs autres interfaces, chaque interface définissant :
   (a) des *opérations* ou méthodes, avec leurs paramètres, la modalité des paramètres — `in`, `out`, `inout`, leur type de retour, des exceptions éventuelles,
   (b) des *attributs* manipulables au travers d'accesseurs et de mutateurs,
   (c) des *types* ;
4. des *composants* définissant des *ports*, facettes, réceptacles, sources, puits ;
5. des fabriques de composants — *home* — permettant de définir en sus des opérations et attributs usuels des opérations de recherche — *finder* — et de construction — *factory* — de composants ;
6. des types d'*exceptions* contenant éventuellement des données complexes ;
7. des objets-valeurs — `valuetype` — qui sont des classes avec leurs opérations, attributs, types, ... dont les instances sont passées par valeur et non par référence.

Pour des raisons de cohérence du modèle, nous avons choisi de ne pas tenir compte de certains aspects du langage IDL3, en particulier :

– la possibilité pour les composants d'implanter des interfaces. Il nous semble que cette possibilité relève de « l'héritage alimentaire » et n'est présente que pour des raisons de compatibilité ;
– la complexité des objets-valeur qui sont, pour ce qui nous concerne, traités comme de simples structures.

Une spécification IDL3 est normalement accompagnée de l'implantation correspondante et de fichiers de description d'assemblage et de déploiement. Nous discuterons de ce problème dans le chapitre 8 consacré à l'outil FIDL où nous verrons que ces informations propres aux composants CCM seront utilisées dans le cadre du test. Pour plus de détails sur ces opérations de projection, nous renvoyons aux documents de références déjà cités ainsi qu'à la documentation technique du projet OpenCCM.

4.2.2 Spécification

Une *spécification* FIDL est incluse en tant que commentaire dans une description d'interface, de composant ou de maison de composant IDL3. Cette spécification est subdivisée en trois parties :

1. une première partie — optionnelle — déclarative introduisant les noms et types des fonctions utilisées dans l'expression FIDL proprement dite ;
2. une deuxième partie qui est l'expression FIDL ;
3. une troisième partie optionnelle comprenant les définitions de fonctions précédemment déclarées.

Les sections 1 et 3 concernant les fonctions permettent de déclarer et définir des fonctions auxiliaires sans effets de bord pour calculer les valeurs des données transmises dans les messages. Le fait que la déclaration soit séparée de la définition permet de changer le langage d'implantation des fonctions auxiliaires.

4.2.3 Interfaces

La spécification d'une interface définit un *contrat*, au sens de Meyer [109], entre toute *facette* typée par cette interface et tout composant utilisant une facette de ce type. Ce contrat est divisé en trois parties : une partie syntaxique, définissant les structures de messages, écrite en IDL3 ; une partie protocolaire décrivant les séquences d'échanges de messages autorisées par cette interface, l'ensemble de traces représentant le protocole d'utilisation et d'implantation de l'interface ; une partie calculatoire définissant des fonctions utilisées pour l'évaluation des valeurs échangées et écrite actuellement dans le langage Jaskell. On notera que le contrat s'applique à l'*interface* et non à un port donné et donc que sa mise en œuvre concrète dépend des modalités de connexion du port typé par l'interface : une facette `single` qui est partagée entre plusieurs connexions voit les messages de toutes ses connexions arbitrairement entrelacés, ce qui n'est pas le cas pour une facette `multiple` laquelle est instanciée de manière indépendante à chaque connexion.

L'exemple 4.2 décrit une interface simple offrant trois méthodes sans paramètres ni valeur de retour et spécifiant un comportement de type « session » : l'utilisateur doit commencer par appeler la méthode `ouvrir`, puis il peut faire un nombre quelconques d'appels à `m` et il doit terminer avec un appel à `fermer` avant de pouvoir recommencer.

```
module Acces {
  interface Controle {
    void ouvrir ();
    void m ();
    void fermer ();
    /*** FIDL
      (ouvrir()m()*fermer())*
    */
  }
...
}
```

FIG. 4.2 – Interface (sans données).

Une expression FIDL prend la forme d'une *expression rationnelle* dont les lettres sont des expressions dénotant des ensembles de messages. On retrouve les opérateurs usuels que sont la concaténation, l'alternative (+), le produit de mélange ($\|$) et l'itération ($\star$) et comme on peut s'y attendre, les parenthèses permettent de grouper les expressions et de modifier l'ordre de précédence usuel des opérateurs.

Un message $\rightarrow m(x_0, \ldots, x_n)$ dénote un appel de méthode m avec des paramètres `in`- et `inout`- $x_0, \ldots, x_n$; un message $\leftarrow m(x_0, \ldots, x_n : x)$ dénote un retour d'appel de méthode m avec les paramètres `inout`- et `out`- $x_0, \ldots, x_n$ et une valeur de retour x. Les valeurs des paramètres et de retour des messages peuvent être soit des valeurs littérales, soit des variables déclarées dans une expression de liaison englobante, soit un caractère *joker* `'_'`. La notation abrégée $m(x_1, \ldots, x_n : x)$ comprenant l'appel et le retour pour une méthode m est autorisée dans la mesure où il n'y a pas d'ambiguïté possible sur les paramètres — c'est à dire qu'il n'y a pas de paramètre en mode `inout` ou que leur valeur est sans importance dans l'expression (_).

Une exception levée par une méthode est dénotée $\leftarrow m\langle E, x_1, \ldots, x_n\rangle$ où E est le type de l'exception et $x_1, \ldots, x_n$ les valeurs des attributs du type E.

L'exemple 4.3 est un peu plus complexe puisqu'il introduit la notion de fonction et de *contraintes* sur les valeurs des messages. Une expression de contraintes $x_1 : P_1(x_1); \ldots; x_n : P_n(x_n)$ `in` E a pour effet :

1. d'introduire dans la portée de l'expression E les variables déclarées $x_1, \ldots, x_n$;
2. de définir des contraintes $P_i(x_i)$ sur les valeurs possibles de ces variables sous la forme d'un prédicat logique liant une variable x_i. Ce prédicat peut éventuellement être omis auquel cas il est considéré comme la constante **true**.

Le type de chaque variable peut normalement être inféré de son contexte par le *vérificateur de types*, car chaque variable utilisée correspond à un paramètre typé d'une méthode. Si ce n'est pas le cas, l'ambiguïté pourra être levée en insérant une notation de type $x :: T$ après la déclaration de la variable.

Les variables introduites sont liées dans l'environnement après leur définition ce qui les rend disponibles pour les expressions de contraintes suivantes mais interdit les expressions récursives, c'est-à-dire les expressions contraignant deux variables et se faisant mutuellement référence. De fait, la notation $x_1 : P_1(x_1); \ldots; x_n : P_n(x_n)$ `in` E est une facilité syntaxique pour $(x_1 : P_1(x_1)$ `in` $(x_2 : P_2(x_2)$ `in` $\ldots (x_n : P_n(x_n)$ `in` $E))$. Les contraintes sont des expressions fonctionnelles quelconques de type booléen qui sont évaluées à chaque utilisation de la variable avec la valeur effective qu'elle possède au moment de l'évaluation, valeur qui dépend du contenu des messages.

```
interface Accounts {
  long    balance (in long accountNo) ;
  boolean withdraw (in long accountNo, in long amount) ;
  boolean transfer (in long fromAccountNo, in long toAccountNo,
                    in long amount) ;
  void    deposit (in long accountNo,in long amount) ;

    /***
      header
        -- retourne le solde courant pour un numero de compte donne
        bal : Trace,long -> long;
      FIDL
       (
       ((n in ->balance(n) (s : s == bal(Trace,n) in <-balance(:s)))) +
       (n,m in ->withdraw(n,m)
           (ko : ko == (m <= bal(Trace,n)) in <-withdraw(:ko))) +
       (n,o,m in ->transfer(n,o,m)
           (ko : ko == (m <= bal(Trace,n)) in <-transfer(:ko))) +
       deposit(_,_)
       )*

      body
      bal (~->withdraw(num,somme) : ~<-withdraw(True) : h) numc =
          if (num == numc) then (bal h num) - somme else (bal h num);
      bal (~->deposit(num,somme)  : h) numc =
          if (num == numc) then (bal h num) + somme else (bal h num);
      bal (~->transfer(f,t,somme)  : h) numc =
          if (num == f) then
             (bal h num) - somme
          else if (num == t) then
                  (bal h num) + somme
               else
                  (bal h num);
      bal (_ : h) num = bal h num;
      bal [] num = 0

    */
};
```

FIG. 4.3 – Exemple de spécification FIDL : interface Accounts.

L'exemple de la figure 4.3 spécifie le comportement d'une interface permettant de réaliser des opérations sur des comptes bancaires. La spécification de l'interface `Account` est divisée en trois par-

ties dont la signification informelle est la suivante :

- la section principale est introduite par le mot-clé `FIDL`, c'est elle qui contient la spécification comportementale de l'interface sous la forme d'une expression du langage FIDL. Dans le cas présent, cette expression se réduit à `(balance() + withdraw() + deposit())*` si l'on ne tient pas compte du contenu des messages, ce qui signifie que les trois méthodes sont totalement indépendantes — peuvent être appelées dans n'importe quel ordre. L'introduction de variables et d'expressions de contraintes permet de préciser le résultat de l'appel de méthode sur cette interface, en l'occurrence le fait qu'un débit n'est autorisé que si son montant est inférieur au solde courant (calculé par la fonction `bal`), qu'un dépôt est toujours autorisé quel qu'en soit le montant et que la méthode `balance` calcule le solde courant du compte ;
- la section `body` contient la définition de la ou des fonctions utilisées dans la partie principale de la spécification. Ces fonctions peuvent être théoriquement écrites dans n'importe quel langage sans effet de bord. Dans la pratique, on utilisera un sous-ensemble du langage Haskell. La fonction `bal` calcule le solde courant pour un numéro de compte donné en fonction de l'historique des retraits et des dépôts intervenus sur ce compte. Dans l'expression du langage de traces de l'interface, la fonction est utilisée pour contraindre les valeurs possibles de certaines variables, en l'occurrence la valeur de retour des méthodes `withdraw`, `deposit` et `transfer`, en fonction du numéro de compte pour lequel la méthode est invoquée et de la trace courante de l'interface dénotée par la variable globale `Trace` qui représente la séquence des messages émis et reçus par l'interface depuis son « instantiation » ;
- la section `header` permet de s'abstraire du langage concret utilisé pour écrire des fonctions. On y déclare uniquement le type des fonctions selon une syntaxe normalisée, les types utilisés étant ceux du langage IDL3.

Lorsque la spécification d'une interface n'est pas explicitement donnée, elle est inférée automatiquement à partir de sa signature : pour toute interface I offrant les méthodes $m_1, m_2, \ldots, m_n$, le langage induit de l'interface est l'ensemble des appels-retours possibles des méthodes de I sans contrainte sur les paramètres ni ordonnancement des méthodes.

4.2.4 Composants

Comme dans le cas des interfaces, les spécifications FIDL des composants sont insérées sous forme de commentaires à la fin de la description IDL3. Cette spécification est un peu plus compliquée car elle décrit les interactions entre les différents services requis et offerts par le composant, la structure générale est cependant la même. Le composant est sensé respecter les spécifications des interfaces des facettes qu'il offre, ces spécifications font donc implicitement partie de la spécification du composant.

L'expression d'un composant est une conjonction de plusieurs expressions que l'on peut voir comme des descriptions de parties du comportement du composant. Cette opération de « composition » est notée `and` et n'apporte pas de réel pouvoir d'expression supplémentaire (*cf.* section 6.1) mais plutôt des facilités d'écriture appréciables : les comportements peuvent être décrits comme une conjonction de vues différentes.

Un événement d'un composant contient plus d'informations que celui d'une interface : le composant reçoit l'événement par un port et donc le nom du port devient partie intégrante de l'événement car un composant peut offrir ou requérir plusieurs ports d'un même type. D'autre part, les composants peuvent échanger des messages synchrones comme des messages asynchrones et les messages asynchrones peuvent être entrelacés. On utilise à nouveau les symboles $\rightarrow$ et $\leftarrow$ pour désigner les appels et retours de méthodes et le point lorsqu'il s'agit d'un appel suivi du retour correspondant : $a \rightarrow m(), a \leftarrow m(), a.m()$, où a est le nom du port.

La figure 4.4 est un exemple de composant `BankAccount` offrant une interface `Accounts`, émettant des messages de type `Offer` et utilisant une interface `Banker` supposée offrir une méthode `alert()`. Cette spécification précise que :

- chaque fois qu'un dépôt supérieur à 100 est effectué, un message d'alerte est transmis au banquier ;
- lorsqu'un débit est refusé, c'est-à-dire lorsque la méthode `withdraw` retourne la valeur `false`, une offre de crédit est adressée au client (émission d'un message asynchrone de type `Offre`).

Par ailleurs, le composant doit respecter la spécification de l'interface `Accounts` dont la spécification est implicitement ajoutée à la sienne.

```
composant BankAccount {
  provides Accounts a;
  uses Banker b;
  emits Offer o;
  /*** FIDL
    ((n,m:m<=100 in a->deposit(n,m)a<-deposit()) +
     (n,m:m > 100 in a->deposit(n,m) b.alert() a<-deposit()))*
    and
     (a.withdraw(_,_:true) +
      a->withdraw(_,_) o[] a<-withdraw(:false))*
  */
}
```

FIG. 4.4 – Exemple de spécification FIDL : composant BankAccount.

4.2.5 Grammaire

La grammaire complète du langage est donnée dans l'annexe A, nous en donnons dans la figure 4.5 une version abrégée définissant uniquement la syntaxe des expressions de comportement pour faciliter la lecture des exemples et définitions de ce chapitre. La partie gauche décrit la syntaxe des expressions FIDL dites élémentaires pour une spécification d'interface. La partie droite décrit la syntaxe abstraite des expressions de contraintes sur des variables.

(4.1)

$$
\begin{array}{lcl}
Expr & \rightarrow & ExprExpr \mid Expr^* \mid \\
 & & Expr + Expr \mid (Expr) \mid \\
 & & Expr \parallel Expr \mid \\
 & & (Ctr\ \mathbf{in}\ Expr) \mid \\
 & & Msg \mid \texttt{void} \\
Msg & \rightarrow & \leftarrow m(Out) \mid \\
 & & \rightarrow m(In) \mid m(Out) \\
 & & \leftarrow m\langle Exc \rangle \\
In & \rightarrow & \epsilon \mid Param \\
Exc & \rightarrow & t, Param \mid t \\
Param & \rightarrow & Atom \mid Atom, Param \\
Atom & \rightarrow & x \mid l \\
Out & \rightarrow & \epsilon \mid In\ Ret \\
Ret & \rightarrow & \epsilon \mid : Atom
\end{array}
$$

(4.2)

$$
\begin{array}{lcl}
Ctr & \rightarrow & x : Pred \\
Pred & \rightarrow & \texttt{true} \mid \texttt{false} \mid \\
 & & Pred \vee Pred \mid \\
 & & \neg Pred \mid p(x, Fun) \\
Fun & \rightarrow & f(Pars) \mid l \\
Pars & \rightarrow & Pars, Par \mid Par \\
Par & \rightarrow & y \mid l \mid Fun
\end{array}
$$

FIG. 4.5 – Syntaxe des expressions FIDL (interfaces).

Les symboles terminaux x,y désignent des variables, p un prédicat, l un littéral, t un nom de type et f une fonction n-aire. Pour faciliter l'écriture des contraintes, on permettra d'écrire

$$(x : P(x), y : P(y)\ \mathbf{in}\ E)$$

au lieu de

$$(x : P(x)\ \mathbf{in}\ (y : P(y)\ \mathbf{in}\ E)).$$

Par ailleurs, on autorise l'utilisation du symbole _ comme « joker ». Pour chaque utilisation de ce symbole, on substitue implicitement dans l'expression une variable x au symbole _ et on lie x par une contrainte qui est toujours vraie :

$$\leftarrow m(\_) \Leftrightarrow (x : \mathbf{true}\ \mathbf{in}\ m(x)).$$

$$
\begin{array}{lcl}
Comp & \rightarrow & Comp \textbf{ and } Expr \mid Expr \\
Expr & \rightarrow & ExprExpr \mid Expr^* \mid Expr + Expr \mid Expr \parallel Expr \mid \\
 & & (Expr) \mid (Ctr \text{ in } Expr) \mid Msg \\
Msg & \rightarrow & n \leftarrow m(Out) \mid n \rightarrow m(In) \mid n.m(Out) \mid \\
 & & n \leftarrow Event \mid n \rightarrow Event \\
Event & \rightarrow & t[In] \\
In & \rightarrow & \epsilon \mid Param \\
Param & \rightarrow & Atom \mid Atom, Param \\
Atom & \rightarrow & x \mid l \\
Out & \rightarrow & \epsilon \mid InRet \\
Ret & \rightarrow & \epsilon \mid : Atom
\end{array}
\tag{4.3}
$$

FIG. 4.6 – Syntaxe des expressions FIDL (composants).

Un composant est spécifié par un nombre quelconque d'expressions élementaires reliées par l'opérateur **and**. De plus, l'identité des ports mis en œuvre doit être précisée et un composant peut émettre et recevoir des messages au travers de ports asynchrones. La grammaire des expressions FIDL pour les composants est présentée dans la figure 4.6.

4.3 Exemples

4.3.1 Un guichet automatique de banque

Nous détaillons dans cette section un exemple complet de spécification de composants FIDL, un système de gestion de *Guichet Automatique de Banque*. La structure générale du modèle est décrite dans la figure 4.8 : le module `Banque` décrit un certain nombres d'interfaces et un composant. Le composant `ATM` — *Automatic Teller Machine* — est une représentation de la machine physique utilisée par un client. L'architecture du système est schématisée dans la figure 4.7, seule la partie concernant le guichet automatique étant décrite plus précisément.

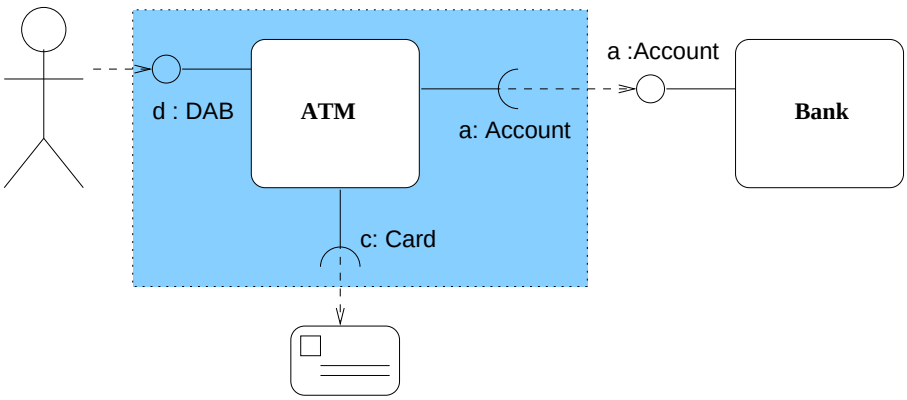


FIG. 4.7 – Schéma du guichet automatique.

Le composant ATM offre une interface — implicitement `unique` — permettant de réaliser diverses opérations et requiert deux ports : un port de type `Card` représentant le lien avec la carte bancaire du client et un autre de type `Account` représentant le lien avec la banque.

L'interface `Card` est détaillée dans la figure 4.9. Le numéro de compte est stocké dans la carte et est donc le même pour toute la durée d'une session, d'où la déclaration de la variables n englobant tous les échanges de messages : le résultat de la méthode `accountNo` est toujours n. Le résultat de la méthode

```
module Banque {
   exception error {};
   exception keep_card {};

  interface Card {...
  interface DAB { ...
  interface Account { ...

   component ATM {
      provides DAB d;
      uses Account a;
      uses Card    c;

   };

};
```

FIG. 4.8 – Exemple de spécification FIDL : module Banque.

`checkCode` est comme on peut s'y attendre dépendant du code « saisi » et est laissé au soin de l'implantation. Enfin la méthode `failedCode` calcule le nombre de fois où un code incorrect a été saisi — en examinant toutes les réponses `false` de la méthode `checkCode` — et retourne cette valeur.

```
  interface Card {
    boolean checkCode(in long code);
    long    failedCode();
    long    accountNo();

    /***
     header
       failed : Trace -> long
    FIDL
       n in (checkCode(_:_) +
        (f: f == failed(Trace) in failedCode(:f)) +
        accountNo(:n))*

     body
       failed (~<-checkCode(False) : h) = (failed h) + 1;
       failed (_ : h) = failed h;
       failed [] = 0
    */
  };
```

FIG. 4.9 – Exemple de spécification FIDL : interface Card.

L'interface DAB, figure 4.10, décrit le comportement du guichet automatique du point de vue du client. Le dialogue commence par l'insertion de la carte, puis la saisie du code PIN et enfin les diverses opérations. Cette interface spécifie en particulier que les opérations ne sont accessibles que si le code saisi est correct et si le nombre de saisies incorrectes est inférieur à 3. L'exception `keep_card` peut être levée par chacune des différentes méthodes : elle signifie que la carte est conservée par le guichet automatique et que plus aucune opération n'est possible avant de refaire une nouvelle opération `insert()` modélisant le fait qu'une nouvelle carte est introduite.

Cet exemple relativement simple permet toutefois d'illustrer les principaux concepts des spécifications FIDL dans le cas de composants sans spécification propre.

4.3.2 Un système de vote électronique

Le deuxième exemple est celui d'un système de vote électronique. L'architecture du système de vote est représentée schématiquement sur la figure 4.11 : il existe un certain nombre de composants individuels

```
interface DAB {
  void    insert() raises (keep_card);
  boolean pinCode(in long code) raises (keep_card);
  void    withdrawCard () raises (keep_card);
  boolean withdrawal (in long amount) raises (keep_card);
  boolean transfer (in long amount,in long toAccountNo) raises (keep_card,error);
  long    balance () raises (keep_card);

  /*** FIDL
     (insert()
      (((pinCode(_:false) + void) ( pinCode(_:false) + void)  pinCode(_:true)
         ( withdrawal(_:_) + balance(:_) +
           -> transfer(_,_) (<-transfer(:_) +  <-transfer<error>)))*
        withdrawCard()) +
      (pinCode(_:false) pinCode(_:false) pinCode(_:false)
        ((->withdrawal(_) <-withdrawal<keep_card>) +
         (->balance() <-balance<keep_card>) +
         (->transfer(_,_) <-transfer<keep_card>) +
         (->withdrawCard() <-withdrawCard<keep_card>))))*
  */
};
```

FIG. 4.10 – Exemple de spécification FIDL : interface DAB

de *vote électronique* qui sont connectés à un *centre de vote* au travers d'une interface `Vote_Center` leur permettant de transmettre le vote — un choix binaire ; et qui par ailleurs peuvent recevoir un événement signifiant la clôture du scrutin — type `Closure` — et contenant le résultat du vote.

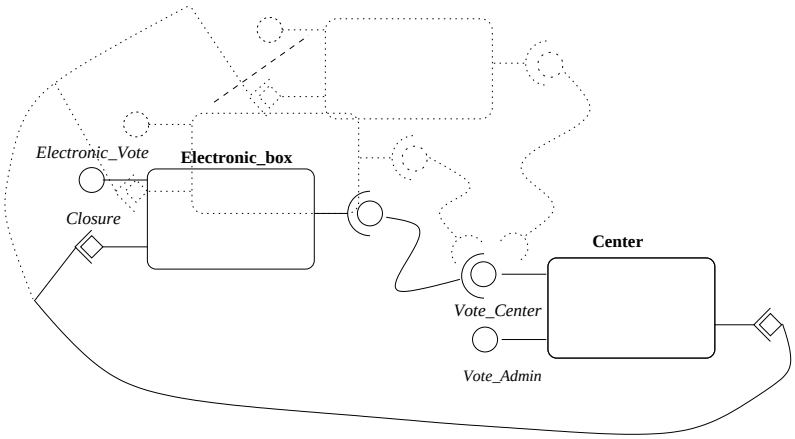


FIG. 4.11 – Architecture du système vote.

Les types de données utilisés sont définis dans le module `Vote`, figure 4.12 : diverses exceptions, la structure de l'événement de fermeture du scrutin et les composants et interfaces.

L'interface `Electronic_Vote` (figure 4.13) représente l'interaction entre chaque boîte individuelle de vote et l'utilisateur. Son fonctionnement est simple : elle permet à l'utilisateur de voter une et une seule fois, et refuse pendant un certain temps de fournir le résultat du scrutin, puis fournit toujours le même résultat. On notera l'utilisation de l'opérateur `||` qui permet d'entrelacer arbitrairement les appels aux différentes méthodes de l'interface — tout en respectant les règles de formation correcte des échanges de messages, et le fait que rien n'est dit dans l'interface quant au moment où le résultat est disponible.

L'interface suivante, `Vote_Center`, est fournie par le centre de vote pour récupérer les différents votes. Son fonctionnement est symétrique de celui de l'interface précédente : elle permet de voter une et une seule fois, et éventuellement elle interdit le vote si celui-ci survient trop tard. L'interface d'administration, enfin, permet de clôre le scrutin une et une seule fois (voir figure 4.14).

```
module Vote {
  exception too_late {};
  exception already_voted {};
  exception already_closed {};
  exception not_closed {};

  eventtype Closure {
    attribute long yes_number;
    attribute long no_number;
  };
  interface Electronic_Vote{ ...
  interface Vote_Center { ...
  interface Vote_Admin { ...
  component Center { ...
  component Electronic_Box { ...

};
```

FIG. 4.12 – Exemple 2 : module Vote.

```
  interface Electronic_Vote{
    boolean vote (in boolean choice);
    boolean results(out long yes,out long no);

    /*** FIDL
       (c in vote(c:true) + void) (c in vote(c : false) )*
     ||(results(_,_:false)*(x,y in results(x,y:true)*))
    */
  };
```

FIG. 4.13 – Exemple 2 : interface de vote électronique.

```
  interface Vote_Center {
    void vote (in boolean choice) raises (too_late,already_voted);

        /*** FIDL
           (x  in vote(x)) (x  in ->vote(x) <-vote<already_voted> + void)*
           (x in ->vote(x) <-vote<too_late> )*
        */
  };

  interface Vote_Admin {
    void close() raises (already_closed);

      /*** FIDL
        close() (->close() <-close<already_closed>)*
      */
  };
```

FIG. 4.14 – Exemple 2 : interfaces centre de vote & administration.

Les spécifications des composants sont ici bien plus intéressantes car elles permettent de *mettre en musique* le comportement des interfaces et de préciser, par exemple, l'ordonnancement des différents événements. Le composant `Center` est détaillé dans la figure 4.15 : c'est lui qui calcule le résultat du vote en fonction des votes reçus et effectivement pris en compte (fonction `result`) et qui précise à l'aide de l'opérateur de composition `and` comment la clôture du vote interagit avec les votes. Le choix fait dans cette spécification est que tout vote survenant après un *appel* de `close()` n'est plus pris en compte, un autre choix possible eût été de prendre quand même en compte ces votes.

```
component Center {

  provides Vote_Center v;
  provides Vote_Admin a;
  publishes Closure c;
  /***
      header
      result : Trace, boolean -> long

      FIDL
        a->close() (x : x == result(Trace,true),
                    y : y == result(Trace,false) in c[x,y])
        a<-close() a.close<already_closed>*
       and
        (v.vote(_) (x in v->vote(x) v<-vote<already_voted>)* + void)
         ((a.close() (a.close<already_closed>* ||
                      (v->vote(_)v<-vote<too_late>)*))
           +
          (a->close() || v->vote(_))
          (v<-vote<too_late> (v->vote(_)v<-vote<too_late>)* ||
           a<-close() (a->close() a<-close<already_closed>)*))

        body
        result [] _ = 0;
        result (~v->vote(x) : ~v<-vote() : h) y =
           if x == y then (result h x) + 1 else (result h x);
        result ( m : ms) _ = result ms;
      */
};
```

FIG. 4.15 – Exemple 2 : composant centre de vote.

Le composant `Electronic_Box` enfin (figure 4.16), permet de préciser le comportement de l'interface de vote électronique en relation avec l'événement de notification de fermeture du scrutin.

On voit donc bien que c'est la spécification des composants qui donne la sémantique du système de vote et qui a pour fonction d'assurer simultanément l'unicité des votes réalisés au moyen de chacune des boîtes électroniques, ce qui est spécifié par l'interface de vote électronique ; et par ailleurs de garantir que tous les votes effectués avant la clôture du scrutin sont pris en compte dans le décompte final des voix qui dépend justement de la clôture du scrutin. On verra au chapitre 6 consacré à la composition comment à partir de ces spécifications on peut espérer vérifier certaines propriétés du système.

La notation que nous avons introduite n'est pas et n'a pas pour objectif d'être un langage de programmation général. Comme tout langage de spécification, il vise à l'abstraction et doit faciliter l'expression de concepts de haut-niveau et améliorer la qualité du système en explicitant des exigences informelles de manière formelle. Cette notation possède donc des limitations volontaires :

- toutes les variables utilisées doivent être présentes dans les messages échangés. Il n'est donc pas possible d'exprimer des « calculs » intermédiaires ou des états qui ne dépendent pas directement du contenu des messages ;
- les contraintes ne peuvent être récursives et ne peuvent porter que sur l'état courant de l'objet contraint. On ne peut donc pas faire dépendre le contenu d'un message du « futur » ;
- par conséquent, l'expression de comportements non rationnels est rendue plus difficile (*p.ex.* le comptage de lettres non borné).

```
component Electronic_Box {
  provides Electronic_Vote e;
  uses Vote_Center v;
  consumes Closure c;

      /*** FIDL
             (x  in e->vote(x) ( v.vote(x) e<-vote(:true) +
                                 v->vote(x) v<-vote<too_late> e<-vote(:false)))
             e.vote(_:false)*
         and
             (e.results(_,_:false)
              (x,y in (c[x,y] + e->results() c[x,y] c[_,_]*
                               e<-results(x,y:true))
                       (e.results(x,y:true)* || c[_,_]*)))
      */
};
```

FIG. 4.16 – Exemple 2 : composant boîte individuelle de vote.

Nous verrons au chapitre suivant que l'expressivité de la notation dépend de la puissance du langage d'expression des contraintes, notre objectif étant de rester dans la mesure du possible dans les limites des langages rationnels.

Chapitre 5

Automates FIDL

Ce chapitre est une présentation formelle de l'outil principal permettant de modéliser les comportements de composants et d'interfaces dans les modèles FIDL. Ces comportements sont donc modélisés par une certaine classe d'automates, appelés automates FIDL, dont la principale caractéristique est de reconnaître un langage sur un certain alphabet de messages, potentiellement infini, à partir d'un alphabet plus restreint comprenant des variables et des contraintes sur ces variables.

La première partie de ce chapitre définit les automates FIDL et surtout leur comportement, c'est-à-dire le langage qu'ils reconnaissent et dont la définition nécessite l'utilisation d'un *environnement*. La deuxième partie définit des expressions, similaires aux expressions rationnelles, grâce auxquelles on peut textuellement décrire un automate FIDL. Il s'agit bien sûr des expressions que nous avons informellement définies dans le chapitre précédent. La troisième section, enfin, est consacrée aux problèmes de la vérification des entités définies au moyen d'automates FIDL et plus particulièrement aux problèmes du traitement des types de données et de la résolution des contraintes.

5.1 Les automates FIDL

Après quelques préliminaires destinés à fixer les notations et conventions mathématiques utilisées dans ce chapitre, nous définissons donc formellement un automate FIDL et le langage qu'il reconnaît. La définition de ce langage prend en compte par ailleurs les problèmes de non-déterminisme de l'automate et s'intéresse aux cas des automates synchronisés. Les contraintes associées à la définition du processus de reconnaissance dans un tel automate nous amènent *in fine* à préciser des propriétés de *bonne formation* des automates.

5.1.1 Préliminaires

Monoïdes, morphismes

Pour tout ensemble X, X^* dénote le monoïde libre engendré par X. Par convention, X sera appelé alphabet et sera considéré comme *a priori* fini et non vide, sauf indication explicite du contraire. Un morphisme de monoïde $\alpha : X^* \rightarrow Y^*$ est une application qui *préserve* la structure de monoïde :

$$\forall u, v \in X^*, \alpha(uv) = \alpha(u)\alpha(v), \tag{5.1}$$

$$\alpha(\epsilon) = \epsilon, \tag{5.2}$$

où la loi de composition interne de chacun des monoïdes est simplement la concaténation des mots et ϵ désigne le mot vide ou élément neutre. Un morphisme est alphabétique si $\alpha(X) \subseteq Y \cup \{\epsilon\}$. Une *projection* $\Pi_Y : X^* \rightarrow Y^*$ est un morphisme alphabétique tel que $Y \subseteq X$ et

$$\forall x \in X, \Pi_Y(x) = \begin{cases} x, \text{ si } x \in Y, \\ \epsilon, \text{ sinon .} \end{cases}$$

Pour tout X, on note $\#$ le morphisme de $(X^*, \epsilon, .)$ dans $(\mathbb{N}, 0, +)$ tel que :

$$\begin{array}{rrll} \#: & X^* & \to \mathbb{N} & \\ & \epsilon & \mapsto & 0 \\ & x & \mapsto & 1 \quad (x \in X) \\ & u.v & \mapsto & \#u + \#v, \end{array}$$

autrement dit $\#$ est la fonction donnant la *longueur* d'un mot.

Soit $h_a^b : X^* \to Y^*$ le morphisme défini pour tout $a \in X, b \in Y$, par

$$h_a^b(x) = \begin{cases} b, \text{ si } x = a, \\ x, \text{ sinon .} \end{cases}$$

On notera $h_{a_1,a_2,\ldots,a_n}^{b_1,b_2,\ldots,b_n}$ le morphisme défini par

$$h_{a_1,a_2,\ldots,a_n}^{b_1,b_2,\ldots,b_n}(x) = \begin{cases} b_i, \text{ si } \exists i, 1 \leq i \leq n, x = a_i, \\ x, \text{ sinon .} \end{cases} .$$

Soit $X_1 \times \cdots \times X_n$, le produit cartésien de n alphabets, $n \geq 2$, on a

$$h_a^b((x_1, \ldots, x_n)) = (h_a^b(x_1), \ldots, h_a^b(x_1)).$$

L'alphabet d'un langage L est noté $\mathrm{alph}(L)$. S'il n'est pas défini explicitement, c'est l'ensemble des lettres qui composent les mots de ce langage. On parlera aussi d'alphabet *induit* par le langage L.

Produits de mélange & synchronisation

Le produit de *mélange — shuffle —* des mots $u \in X^*$ et $v \in Y^*$, noté $u \sqcup\!\sqcup v$ est défini par :

$$u \sqcup\!\sqcup v = \{u_1 v_1 \ldots u_n v_n \mid u = u_1 \ldots u_n, v = v_1 \ldots v_n, \forall 1 \leq i \leq n, u_i \in X^*, v_i \in Y^*\}. \tag{5.3}$$

Le produit de mélange de deux langages L_1 et L_2 est :

$$L_1 \sqcup\!\sqcup L_2 = \bigcup_{u \in L_1, v \in L_2} u \sqcup\!\sqcup v.$$

Le *produit de synchronisation*[54] — ou *mixage* — de deux langages L_1 et L_2 sur les alphabets X_1 et X_2, noté $L_1 \underset{X_1,X_2}{\sqcap\!\sqcap} L_2$ est défini par :

$$L_1 \underset{X_1,X_2}{\sqcap\!\sqcap} L_2 = \{u \in (X_1 \cup X_2)^* \mid \Pi_{X_1}(u) \in L_1 \text{ et } \Pi_{X_2}(u) \in L_2\}. \tag{5.4}$$

Cette opération est associative :

$$\begin{array}{rcl} (L_1) \sqcap\!\sqcap_{X_1,X_2} (L_2) \sqcap\!\sqcap_{X_2,X_3} (L_3) & = & ((L_1) \sqcap\!\sqcap_{X_1,X_2} (L_2)) \sqcap\!\sqcap_{X_1 \cup X_2, X_3} (L_3) \\ & = & (L_1)_{X_1, X_2 \cup X_3} \sqcap\!\sqcap ((L_2) \sqcap\!\sqcap_{X_2,X_3} (L_3)) \end{array}$$

Lorsque plusieurs langages sont mixés ou mélangés, on utilisera les symboles $\sqcap\!\sqcap$ et $\sqcup\!\sqcup$. De plus, si les alphabets utilisés sont les alphabets induits, on écrira plus simplement $L_1 \sqcap\!\sqcap L_2$. Clairement, si X_1 et X_2 sont des ensembles disjoints, alors $L_1 \sqcap\!\sqcap L_2 = L_1 \sqcup\!\sqcup L_2$.

Pour tout langage $L \subseteq X^*$, $Pref(L)$ est la clôture de L par ses facteurs gauches :

$$Pref(L) = \{v \in X^* \mid \exists u \in L, w \in X^*, u = vw\}.$$

Pour un ensemble quelconque X, $\mathcal{P}X$ est l'ensemble des parties de X.

Données & Types $\mathcal{D}$ désigne l'*univers* des valeurs primitives, ses éléments sont appelés aussi littéraux, par opposition aux variables. Un *type* T est un sous-ensemble de $\mathcal{D}$ et nous noterons $\mathcal{D}_T$ le domaine des valeurs de ce type. $Type(v)$ désignera le type d'une variable ou d'un identifiant v. $\mathcal{V}$ est un ensemble dénombrable de noms de variables disjoint de $\mathcal{D}$.

Symétriquement, on peut définir $\mathcal{D}$ comme l'ensemble résultant de l'union de tous les domaines de tous les types T. Nous ne donnons pas ici de description formelle d'un système de type permettant de construire explicitement un type T et l'ensemble des valeurs $\mathcal{D}$ considérant que cette formalisation n'est pas nécessaire pour le reste de la définition du langage FIDL.

Dans l'immédiat, nous considérerons que l'on dispose d'un ensemble de types primitifs tels que les entiers, les booléens, les chaînes de caractères et de types inductifs — sans variables de types — construits à partir de constructeurs de types tels que les structures et les séquences — le système F_1 en théorie des types.

5.1.2 Automates

Définition 5.1 (Automate FIDL) Un automate FIDL est un quintuplet

$$A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta) \ ,$$

où Q est un ensemble d'états, q_0, l'état initial, un état distingué de Q, T un sous-ensemble de Q contenant les *états terminaux*, Σ un alphabet, Λ un ensemble *fini* de variables, $\mathcal{K}$ un ensemble fini de contraintes sur Λ et δ une relation de transition incluse dans $Q \times (\Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}) \times Q \setminus q_0$.

Ce qui distingue fondamentalement un automate FIDL d'un automate classique est qu'il reconnaît un langage qui n'est pas constitué uniquement des lettres de son alphabet. L'automate est construit sur un alphabet contenant des variables, variables qui sont contraintes par prédicats. L'ensemble des substitutions ou interprétations de variables respectant les contraintes définit le langage en terme de lettres « closes ».

La figure 5.1 est une représentation graphique de l'automate FIDL correspondant au comportement de l'interface `Accounts` décrite dans le chapitre précédent. Sur chaque transition sont indiqués les variables déclarées, les contraintes et les messages. Par exemple, l'étiquette de la transition de l'état 4 à l'état 9 indique que :

- la variable s est déclarée sur la transition ;
- elle est contrainte par l'expression $bal(Trace, n)$. La relation d'égalité est implicite ;
- la variable s est utilisée comme valeur de retour du message $\leftarrow$ `balance`(: s).

5.1.3 Alphabet

Les lettres de l'alphabet Σ d'un automate FIDL sont des couples constitués d'une *enveloppe*, représentée généralement par m et appartenant à un ensemble $\mathcal{X}$, et d'un *contenu*, éventuellement vide, qui est un n-uplet dont les éléments sont des valeurs littérales ou des variables. Ces lettres sont appelées *messages* et sont notées $m(x_1, \ldots, x_{ar(m)})$.

Toute enveloppe possède une arité et une signature qui précisent le nombre des éléments du n-uplet contenu dans le message ainsi que leur type ou l'ensemble des valeurs admissibles. Plus formellement, pour tout $m \in \mathcal{X}$, $ar(m) \in \mathbb{N}$ est l'*arité* de m et pour chaque indice $i \in \{1, \ldots, ar(m)\}$, $m[i]$ est le type du $i^{ème}$ paramètre. Si $ar(m) = 0$, m est un message constant sans contenu.

L'ensemble de tous les messages, pour un ensemble d'enveloppes $\mathcal{X}$, est noté $\mathcal{E}$ et se définit comme :

$$\mathcal{E}_\mathcal{X} = \{m(x_1, \ldots, x_n) \mid m \in \mathcal{X}, n = ar(m), x_i \in \mathcal{V} \cup m[i]\}.$$

L'alphabet Σ d'un automate est donc une partie *finie* de $\mathcal{E}_\mathcal{X}$. L'ensemble des *messages clos* est la partie de $\mathcal{E}_\mathcal{X}$ qui contient uniquement les messages avec un contenu sans variables. Cet ensemble est noté $\bar{\mathcal{E}}_\mathcal{X}$ et défini comme :

$$\bar{\mathcal{E}}_\mathcal{X} = \{m(v_1, \ldots, v_n) \mid m \in \mathcal{X}, n = ar(m), v_i \in m[i]\}.$$

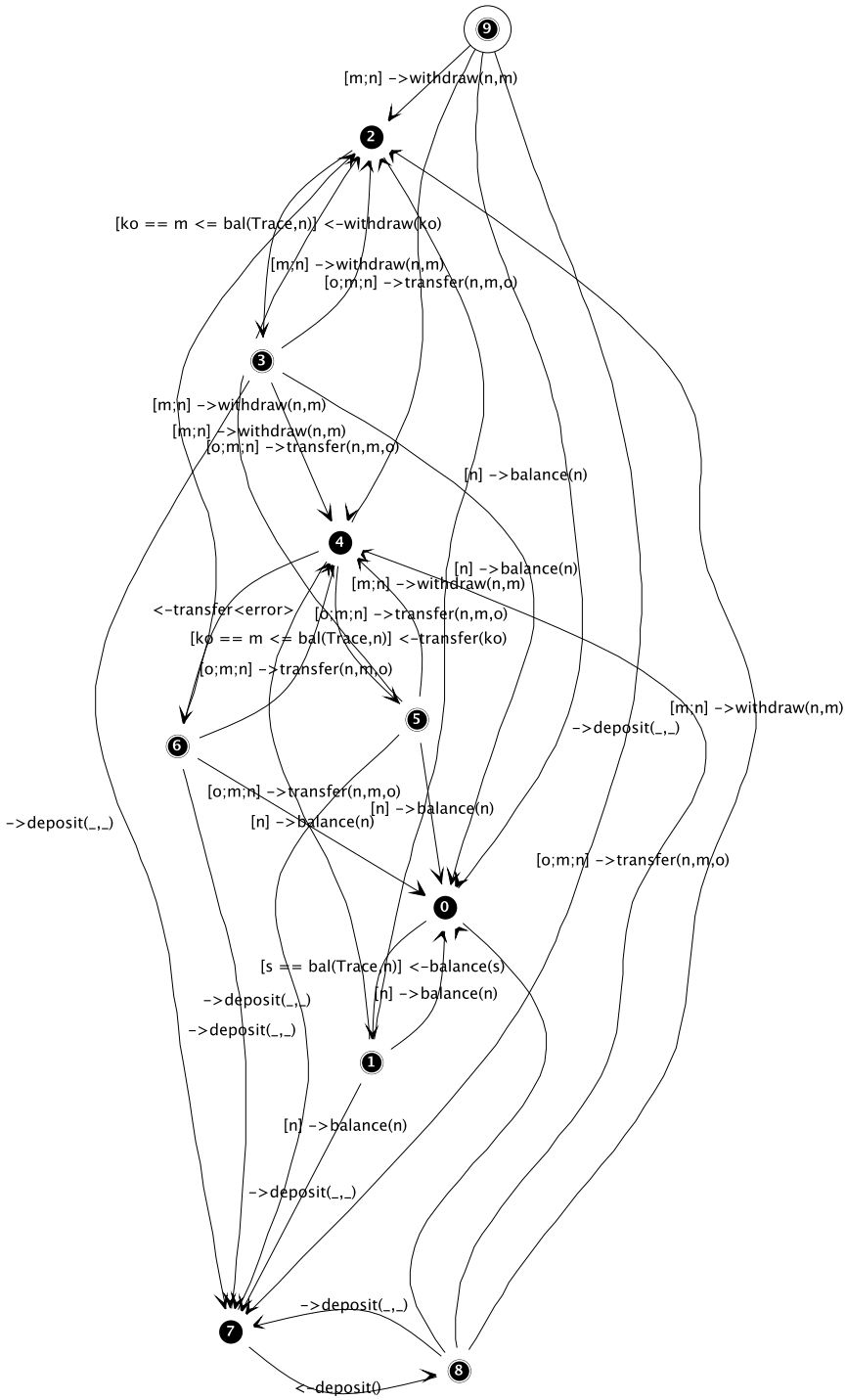


FIG. 5.1 – Exemple d'automate (interface `Accounts`).

On définit $var(\Sigma)$, l'ensemble des variables apparaissant dans les messages d'un alphabet Σ :

$$var(\Sigma) = \bigcup_{m(x_1,\ldots,x_{ar(m)})\in\Sigma} \{x_i \in \mathcal{V},\ \text{pour } 1 \leq i \leq ar(m)\}.$$

5.1.4 Variables & Contraintes

À chaque automate FIDL correspond un ensemble de contraintes $\mathcal{K}$ et un ensemble de variables Λ utilisés dans les étiquettes des transitions de l'automate. Chaque contrainte $c \in \mathcal{K}$ est constituée d'un prédicat $P : \mathcal{D}^r \rightarrow \mathbb{B}$ d'arité r et de deux ensembles de variables, les variables liées $bv(c) \subseteq \Lambda$ et les variables libres $fv(c) \subseteq \Lambda$, tels que pour une contrainte c :

- $bv(c) \cap fv(c) = \emptyset$;
- $var(c) = bv(c) \cup fv(c)$;
- $|bv(c)| = 1$.

Dans la transition citée ci-dessus en exemple :

$$4 \xrightarrow{[s=\text{bal}(\texttt{Trace},n)]\leftarrow\text{balance}(:s)} 9,$$

la contrainte c est l'expression $s = bal(Trace, n)$, la variable s est liée par c, les variables n et `Trace` sont libres. Pour l'automate de la figure 5.1, on a donc :

$$\mathcal{K} = \left\{ \begin{array}{l} \lambda n.\mathbf{true}, \\ \lambda m.\mathbf{true}, \\ \lambda o.\mathbf{true}, \\ \lambda s.(s = \text{bal}(\texttt{Trace}, n)), \\ \lambda ko.(ko = m \leq \text{bal}(\texttt{Trace}, n))) \end{array} \right\},$$

et

$$\Lambda = \{m, n, o, ko, \texttt{Trace}, s\}.$$

La variable liée par chaque contrainte est clairement indiquée par l'opérateur d'abstraction λ, ce qui nous autorisera par la suite à noter la valeur d'une contrainte c en fonction de la valeur de y par $c(y)$: l'application de c sur la valeur de y.

Par extension, on définit pour un automate A l'ensemble des variables liées et libres de A :

$$bv(A) = \bigcup_{c\in\mathcal{K}} bv(c) \qquad \text{et} \qquad \begin{array}{rcl} fv(A) & = & (var(\Sigma) \cup \bigcup_{c\in\mathcal{K}} fv(c)) \setminus bv(A) \\ & = & \Lambda \setminus bv(A). \end{array}$$

L'identifiant `Trace` désigne une variable spéciale, présente dans tous les automates FIDL, et qui a pour fonction de permettre la définition de contraintes et de fonctions dépendant de l'historique de l'exécution de l'automate dans lequel elles sont utilisées. Autrement dit, la *trace* permet de calculer une valeur en fonction de l'état « déplié » de l'automate. Cette variable ne peut apparaître dans le contenu d'un message.

5.1.5 Langage reconnu par un automate FIDL

Le langage reconnu par un automate FIDL est constitué de mots construits sur l'ensemble des messages clos $\bar{\mathcal{E}}$: formellement, le langage associé à un automate A est donc inclus dans $\bar{\mathcal{E}}^*$. Dans un état donné de l'automate, il faut donc faire correspondre à une lettre de l'alphabet formel Σ sur lequel est construit l'automate une ou plusieurs lettres de l'alphabet réel de l'ensemble des mots reconnus par l'automate. Par exemple, à un message noté $m(v_1, \ldots, v_n)$, où $v_1, \ldots, v_n$ est le contenu effectif du message, doit correspondre une transition $(m(x_1, \ldots, x_n), V, C)$ qui peut être déclenchée dans l'état courant.

Dans l'étiquette d'une transition, certains des x_i peuvent être interprétés comme des valeurs littérales, auquel cas v_i doit être la même valeur. D'autres x_j peuvent être des variables, soit introduites dans V et contraintes par C, soit comme partie de l'étiquette d'une transition menant à l'état courant. Dans ce second cas, v_j doit satisfaire la contrainte associée à x_j ou la valeur précédemment « choisie » pour x_j. D'autres

x_i encore peuvent être des variables libres ne correspondant à aucune contrainte. Par conséquent, nous devons mémoriser lors du parcours de l'automate le lien entre un nom de variable et sa valeur dans un *environnement*.

Définition 5.2 (Environnement) L'environnement d'un calcul noté σ — pour *store* — est un triplet $(V, val, pred)$ où :

- $V \subset \mathcal{V}$ est un ensemble fini de variables ;
- $val : V \rightarrow \mathcal{D} \cup \{\perp\}$ est une fonction totale de *valeur* ou *valuation* assignant à chaque variable une valeur dans $\mathcal{D}$, les valeurs indéfinies étant dénotées par $\perp$, où $\perp \notin \mathcal{D}$;
- $pred : V \rightarrow (\mathcal{D} \rightarrow \mathbb{B})$ est une fonction totale de *contrainte* assignant à chaque variable un prédicat sur $\mathcal{D}$ qui est une fonction de l'ensemble des valeurs dans l'ensemble des booléens $\{\texttt{true}, \texttt{false}\}$. Nous désignerons par `true` — resp. `false` — la fonction constante $\lambda x.\texttt{true}$ — resp. $\lambda x.\texttt{false}$.

Pour un environnement σ on note V_σ, val_σ et $pred_\sigma$ les différents composants de l'environnement, l'indice étant omis lorsque l'environnement est clairement précisé par le contexte.

Nous définissons maintenant un pas de l'exécution d'un automate FIDL $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$, à partir d'un couple (q, σ) formé de l'état courant de l'automate $q \in Q$ et d'un environnement associé.

Déclenchement d'une transition Soit un automate A dans un état q associé à un environnement $(V, val, pred)$ noté σ, et une lettre $m(v_1, \ldots, v_n)$ notée a, l'automate peut atteindre l'état q' s'il existe une transition $(q, (m(x_1, \ldots, x_n), W, K), q')$ dans A telle que pour tout i, $1 \leq i \leq n$:

- soit x_i est un littéral et $v_i = x_i$. La transition est définie avec une valeur littérale qui doit être identique à la valeur contenue dans le message ;
- soit x_i est une variable définie dans W et $c(v_i) = \texttt{true}$, pour $c \in K$ la contrainte associée à x_i. La variable x_i est déclarée et contrainte localement et la contrainte est respectée par v_i ;
- soit $x_i \notin W$ est une variable et $x_i \in V$, c'est-à-dire que la variable sur la transition a été déclarée et contrainte précédemment, alors :
  - soit $val_\sigma(x_i) = v_i$,
  - soit $val_\sigma(x_i) = \perp$ et $pred_\sigma(x_i)(v_i) = \texttt{true}$.

Cette propriété est notée

$$(q, \sigma) \xrightarrow{a} (q', \sigma')$$

où $\sigma' = (V_{\sigma'}, val_{\sigma'}, pred_{\sigma'})$ est un nouvel environnement défini par :

$$\begin{array}{rcl}
V_{\sigma'} & = & V_\sigma \cup W, \\
val_{\sigma'} & = & \{x_i \mapsto v_i \mid i \in \{1, \ldots, n\}\} \\
& \cup & \{y \mapsto \perp \mid y \in W, \exists c(y) \in K, \\
& & \quad y \neq x_i, \forall i \in \{1, \ldots, n\}\} \\
& \cup & \{\texttt{Trace} \mapsto val_\sigma(\texttt{Trace}).a\} \\
& \cup & \{y \mapsto z \mid y \notin W, y \in V_\sigma \setminus \{\texttt{Trace}\}, z = val_\sigma(y)\}, \\
pred_{\sigma'} & = & \{x_i \mapsto \texttt{true} \mid i \in \{1, \ldots, n\}\} \\
& \cup & \{y \mapsto c \mid y \in W, \exists c \in K, y \in bv(c), \\
& & \quad y \neq x_i, \forall i \in \{1, \ldots, n\}\} \\
& \cup & \{y \mapsto c \mid y \notin W, y \in V_\sigma \setminus \{\texttt{Trace}\}, c = pred_\sigma(y)\} \\
& \cup & \{\texttt{Trace} \mapsto \texttt{true}\}.
\end{array}$$

La condition de franchissement d'une transition inclut outre une condition sur l'égalité des attributs de l'événement, une vérification de la validité des valeurs des paramètres de messages eu égard à l'environnement courant, donc aux contraintes définies sur les variables. Ce franchissement n'est possible que si, pour chaque paramètre, sa valeur est égale à celle stockée pour la variable correspondante dans l'environnement courant ou compatible avec les contraintes actuellement définies pour cette variable.

La mise à jour de l'environnement — défini ici par création d'un nouvel environnement σ' — comprend les étapes suivantes :

- on étend tout d'abord l'ancien ensemble de variables avec l'ensemble des variables déclarées dans la transition ;
- la valuation des variables est ensuite modifiée : toutes les occurences de variables apparaissant dans le corps du message sur la transition reçoivent la valeur correspondante dans le corps du message de l'événement lu, les autres variables déclarées dans la transition reçoivent la valeur spéciale indéfinie $\perp$ et la trace `Trace` est mise à jour, le reste de l'environnement demeurant inchangé ;
- finalement, la fonction de contrainte est mise à jour : le prédicat associé aux variables devenues définies devient `true` pour signifier qu'il n'est plus nécessaire de le vérifier, les nouvelles contraintes déclarées sur la transition sont ajoutées à l'environnement et les autres contraintes restent inchangées.

Exemple Nous reprenons l'exemple de l'automate 5.1. Le mot u suivant :

$$\rightarrow deposit(123, 100) \leftarrow deposit()$$

mène l'automate dans l'état 7 et dans l'environnement

$$(\{\texttt{Trace}\}, \{\texttt{Trace} \mapsto \rightarrow \text{deposit}(123, 100) \leftarrow \text{deposit}()\}, \{\texttt{Trace} \mapsto \texttt{true}\}).$$

Par application des règles ci-dessus, l'automate acceptera donc de « lire » u suivi de

$$\rightarrow \text{withdraw}(123, 50) \leftarrow \text{withdraw}(\mathbf{true}),$$

ou encore

$$\rightarrow \text{withdraw}(123, 150) \leftarrow \text{withdraw}(\mathbf{false}).$$

Par contre,

$$\rightarrow \text{withdraw}(123, 150) \leftarrow \text{withdraw}(\mathbf{true})$$

n'est évidemment pas un (suffixe de) mot reconnu par l'automate puisque la lecture de $\rightarrow \text{withdraw}(123, 150)$ mène dans l'état 3 avec l'environnement

$$\sigma = \left(\begin{array}{l} \{\texttt{Trace}, m, n\}, \\ \{\texttt{Trace} \mapsto (u \rightarrow \text{withdraw}(123, 150)), m \mapsto 150, n \mapsto 123\}, \\ \{\texttt{Trace} \mapsto \texttt{true}, m \mapsto \mathbf{true}, n \mapsto \mathbf{true}\} \end{array} \right);$$

et dans cet environnement, l'évaluation de

$$(\lambda ko.(ko = (m \leq bal(Trace, n))))(\mathbf{true})$$

est **false** donc la contrainte sur la variable ko n'est pas vérifiée par $\leftarrow \text{withdraw}(\mathbf{true})$

Reconnaissance Enfin, nous pouvons définir la reconnaissance d'un mot et par conséquent l'ensemble des mots reconnus par un automate FIDL.

Définition 5.3 Soient $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$ un automate et $u = e_1 e_2 \dots e_n$ un mot avec $\forall i, 1 \leq i \leq n, e_i \in \bar{\mathcal{E}}$, u est *accepté* par A s'il existe une séquence de couples $(q, \sigma_0) \dots (q_n, \sigma_n)$ tels que

- $q = q_0$, l'état initial de A et $\sigma_0 = (\{\texttt{Trace}\}, \{\texttt{Trace} \mapsto \varepsilon\}, \{\texttt{Trace} \mapsto \texttt{true}\})$;
- quel que soit j, $0 \leq j < n$, $(q_j, \sigma_j) \xrightarrow{e_{j+1}} (q_{j+1}, \sigma_{j+1})$;
- $q_n \in T$.

Le langage reconnu par A, noté $L(A)$, est l'ensemble des mots reconnus par A.

5.1.6 Reconnaissance par synchronisation d'automates

Étant donnés les automates FIDL $A_1, \ldots, A_n$, on définit la reconnaissance d'un mot par *synchronisation des automates* $A_1, \ldots, A_n$. L'état associé à un calcul est un n-uplet d'états élémentaires

$$((q_1, \sigma_1), \ldots, (q_n, \sigma_n)).$$

Un événement $e = m(v_1, \ldots, v_p)$ est reconnu par la machine $\mathcal{A}$ et permet d'atteindre l'état

$$((q'_1, \sigma'_1), \ldots, (q'_n, \sigma'_n)),$$

si pour chaque i, $1 \leq i \leq n$:

1. si l'enveloppe de e appartient à $\mathcal{X}_i$, l'ensemble des enveloppes de l'alphabet de A_i, alors

$$(q_i, \sigma_i) \xrightarrow{e} (q'_i, \sigma'_i) ;$$

2. sinon, $(q_i, \sigma_i) = (q'_i, \sigma'_i)$.

Un mot $u = e_1 e_2 \ldots e_k$ est reconnu par $\mathcal{A}$ si il existe une séquence de n-uplets $Q_0 Q_1 \ldots Q_k$, avec $Q_0 = ((q_1^0, \sigma_1^0), \ldots, (q_n^0, \sigma_n^0)), Q_i = ((q_1^i, \sigma_1^i), \ldots, (q_n^i, \sigma_n^i)), 1 \leq i$, telle que pour tout $j, 1 \leq j \leq k$, pour tout $1 \leq i \leq n$, $(q_j^i, \sigma_j^i) \xrightarrow{m_i} (q_{j+1}^i, \sigma_{j+1}^i)$ et $q_k^i \in T_{A_i}$.

5.1.7 Automate bien-formé

Pour que le processus de *liaison* des variables dans l'environnement en fonction des messages effectivement lus puisse se dérouler correctement, il est nécessaire d'imposer des règles de bonne formation pour les automates :

- toutes les variables, à l'exception de la variable `Trace` doivent être déclarées sur la transition où elles sont utilisées ou sur une transition ayant déjà été franchie par le processus de reconnaissance ;
- lorsque la valeur d'une variable est contrainte par un prédicat dans lequel apparaissent d'autres variables, celles-ci doivent avoir été préalablement déclarées et définies. Cette condition implique que les contraintes ne peuvent être récursivement dépendantes les unes des autres.

Nous définissons tout d'abord une relation entre les différentes contraintes utilisées dans un automate en fonction de leur interdépendance.

Définition 5.4 (Dépendance des contraintes) Étant donné un automate FIDL $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$, la relation de dépendance entre deux contraintes c, c' de $\mathcal{K}$, notée $c \leq c'$ est définie par :

- $c \leq c$;
- s'il existe un chemin $qdq'\mu pd'p'$ dans l'automate A, avec $d = (q, (e, V, K), q') \in \delta$, $d' = (p, (e', V', K'), p') \in \delta$, et des contraintes $c \in K$, $c' \in K'$, telles que $bv(c) \subseteq fv(c')$, alors $c \leq c'$.

Cette relation exprime formellement le fait qu'une variable liée par une contrainte c et utilisée par une autre contrainte c', telles qu'il existe un chemin dans l'automate menant de l'une à l'autre, induit une dépendance de c' envers c.

Les propriétés de *bonne formation* d'un automate s'expriment alors comme :

Définition 5.5 (Automate FIDL bien formé) Étant donné un automate FIDL $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$, on note $\mu_{q,p}$ un chemin de A entre les états q et p formé par une séquence de transitions de δ. A est dit bien formé si et seulement si :

1. la clôture transitive de la relation de dépendance entre contraintes $\leq$ est un ordre partiel : elle est réflexive, transitive et *anti-symétrique* ;
2. pour tout $c \in \mathcal{K}$ et toute transition $(q, (m, V, K), q') \in \delta$ telle que $c \in K$, pour toute variable libre $y \in fv(c)$, alors pour tout chemin partant de q_0 $\mu(q, (m, V, K), q')$, il existe dans μ une transition $(q_1, (m', V', K'), q_2)$ telle que $y \; in V'$;
3. la trace est la seule variable libre de l'automate : $fv(A) = \{\texttt{Trace}\}$.

5.1.8 Non-déterminisme

Un automate FIDL peut être non-déterministe : à partir d'un même état, il peut exister deux transitions étiquetées par le même message telles que les environnements permettant de satisfaire les contraintes apparaissant sur les deux transitions contiennent des valeurs de variables communes. Plus formellement, un automate FIDL $A = (q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$ est non déterministe si, pour deux transitions $d_1 = (q, ((m, x_1, \ldots, x_n), W, K), q')$ et $d_2 = (q, ((m, y_1, \ldots, y_n), W', K'), q'')$, il existe des environnements $\sigma, \sigma'_1, \sigma'_2$ et un mot $u = ve$ tels que :

$$(q_0, \sigma_0) \xrightarrow{u} (q, \sigma) \xrightarrow[d_1]{e} (q', \sigma'_1) \qquad \text{et} \qquad (q_0, \sigma_0) \xrightarrow{u} (q, \sigma) \xrightarrow[d_2]{e} (q'', \sigma'_1).$$

Or nous avons défini la reconnaissance des langages en termes d'automates déterministes. L'extension au cas des automates non-déterministes est toutefois simple : la reconnaissance dans un automate non-déterministe se fait comme dans le cas d'un automate de mots classiques en considérant à chaque pas de l'automate l'ensemble des états accessibles et l'union des environnements associés.

Si l'on note $[q] \subseteq Q$ un état de l'automate non déterministe qui est une classe d'équivalence d'états pour la relation de transition, alors $([q], \sigma) \xrightarrow{e} ([q'], \sigma')$ si et seulement si

$$\forall p \in [q], \exists p' \in [q'], (p, \sigma) \xrightarrow{e} (p', \sigma_p),$$

avec $\sigma' = \cup_p \sigma_p$.

5.2 Expressions FIDL

Les automates FIDL sont construits à partir d'expressions de même nom permettant d'exprimer la structure de l'automate sous une forme textuelle. Une expression FIDL est basée sur les opérateurs des expressions rationnelles usuelles mais contient en plus des contraintes. Nous définissons dans cette section le processus de construction des automates FIDL à partir des expressions. Nous faisons en sorte de contraindre la forme des expressions pour que tous les automates FIDL construits à partir d'expressions syntaxiquement correctes soient bien formés.

5.2.1 Expressions

Nous avons déjà vus les expressions FIDL de manière informelle dans le chapitre 4, figure 4.5. Ces expressions sont semblables aux expressions rationnelles classiques formées à partir de l'alphabet des messages possibles dans le contexte de l'expression. En plus des opérateurs usuels des langages rationnels, union, concaténation et étoile, on utilise des opérateurs binaires pour exprimer le produit de mélange $\|$ et la synchronisation **and**. La principale différence s'exprime dans l'utilisation de contraintes et de variables qui ont un impact important sur la sémantique de ces expressions. Pour simplifier les définitions qui suivent, le symbole $\circ$ désigne l'un quelconque des opérateurs binaires ., $\|$, $+$, **and**.

Syntaxe

La figure 5.2 reprend la syntaxe des expressions FIDL introduite au chapitre précédent.

Les symboles x, y, l, t, p et f sont des symboles terminaux désignant respectivement des variables de l'ensemble $\mathcal{V} \setminus \{\texttt{Trace}\}$, des littéraux de l'ensemble $\mathcal{D}$, un nom de type d'exception, des prédicats n-aires de $\mathcal{D}^n$ dans $\mathbb{B}$ et des fonctions d'arités r de $\mathcal{D}^r$ dans $\mathcal{D}$.

Règles sémantiques

Les expressions FIDL doivent respecter par ailleurs des règles sémantiques ayant trait aux noms des variables pour être bien formés. Ces règles sont détaillées ci-dessous.

(5.5)

$$
\begin{array}{lcl}
Expr & \rightarrow & ExprExpr \mid Expr^* \mid \\
 & & Expr + Expr \mid (Expr) \mid \\
 & & Expr \parallel Expr \mid \\
 & & (Ctr\ \mathbf{in}\ Expr) \mid \\
 & & Msg \mid \texttt{void} \\
Msg & \rightarrow & m(Param) \mid m() \\
 & & m\langle Exc \rangle \\
Exc & \rightarrow & t, Param \mid t \\
Param & \rightarrow & Atom \mid Atom, Param \\
Atom & \rightarrow & x \mid l
\end{array}
$$

(5.6)

$$
\begin{array}{lcl}
Ctr & \rightarrow & x : Pred \\
Pred & \rightarrow & \texttt{true} \mid \texttt{false} \mid \\
 & & Pred \vee Pred \mid \\
 & & \neg Pred \mid p(x, Fun) \\
Fun & \rightarrow & f(Pars) \mid l \\
Pars & \rightarrow & Pars, Par \mid Par \\
Par & \rightarrow & y \mid l \mid Fun
\end{array}
$$

FIG. 5.2 – Syntaxe des expressions FIDL (interfaces).

Variables Les expressions FIDL contiennent des variables qui représentent des données transportées par les messages. Pour une expression E, on définit des ensembles de variables liées $bv(E)$ et libres $fv(E)$. L'ensemble des variables de E, $var(E)$ est simplement défini comme $var(E) = bv(E) \cup fv(E)$. Ces ensembles de variables sont définis inductivement par les règles suivantes :

$$
\begin{array}{rcl}
bv(E^*) & = & bv(E), \\
fv(E^*) & = & fv(E)
\end{array}
$$

$$
\begin{array}{rcl}
bv(E \circ F) & = & bv(E) \cup bv(F), \\
fv(E \circ F) & = & fv(E) \cup fv(F)
\end{array}
$$

$$
\begin{array}{rcl}
fv(Msg) = fv(m(x_1, \ldots, x_n)) & = & \{x_i \in \mathcal{V}, 1 \leq i \leq n\}, \\
bv(Msg) = bv(m(x_1, \ldots, x_n)) & = & \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
bv(Ctr\ \mathbf{in}\ E) & = & bv(E) \cup bv(Ctr), \\
fv(Ctr\ \mathbf{in}\ E) & = & fv(E) \cup fv(Ctr) \setminus bv(Ctr)
\end{array}
$$

$$
\begin{array}{rcl}
fv(Ctr) = fv(x : Pred) & = & var(Pred) \setminus \{x\}, \\
bv(Ctr) = bv(x : Pred) & = & \{x\}
\end{array}
$$

L'opération d'α-renommage des variables permet de s'assurer que si $E = F \circ G$, alors on a $bv(F) \cap var(G) = \emptyset$ et $var(F) \cap bv(G) = \emptyset$. Les variables liées ont ainsi une portée unique limitée à l'arbre d'expression qui suit leur introduction.

Définition 5.6 (α-renommage)

$$
\begin{array}{lrcl}
\alpha : & \alpha(\texttt{void}) & \longmapsto & \texttt{void} \\
 & \alpha(Msg) & \longmapsto & Msg \\
 & \alpha(E \circ F) & \longmapsto & \left\{ \begin{array}{ll} \alpha(E) \circ \alpha(F), & \text{si } bv(E) \cap var(F) = var(E) \cap bv(F) = \emptyset \\ \alpha(h_x^{x'}(E) \circ h_x^{x''}(F)), & \text{pour chaque } x \in bv(E) \cap var(F) \text{ ou} \\ & x \in bv(F) \cap var(E), \\ & \text{avec } x', x'' \notin var(F) \cup var(E) \end{array} \right. \\
 & \alpha(E^*) & \longmapsto & \alpha(E)^* \\
 & \alpha(x : P(x)\ \mathbf{in}\ E) & \longmapsto & x : P(x)\ \mathbf{in}\ \alpha(E))
\end{array}
$$

Une expression de contrainte est dite *non ambigüe* si la variable définie et sa contrainte n'utilisent pas de variables liées dans la sous-expression associée, ce qui est définie comme suit :

Propriété 5.7 (Expression non-ambigüe) Une expression $(x : Pred\ \mathbf{in}\ Expr)$ est non-ambigüe si et seulement si :

$$(x \cup var(Pred)) \cap bv(Expr) = \emptyset.$$

Enfin, nous exigerons que toute expression E ne contienne comme seule variable libre que la variable `Trace` :

Propriété 5.8 (Clôture) Une expression E est *close* si et seulement si :

$$fv(E) \subseteq \{\texttt{Trace}\}.$$

Définition 5.9 (Expression bien-formée) Une expression FIDL E est bien formée si :

1. E est transformée par la fonction de renommage α ;
2. E et toutes ses sous-expressions sont non ambigües ;
3. E est close.

5.2.2 Construction

À partir d'une expression E ne comprenant pas d'opérateur de synchronisation **and** (voir chapitre 4, section 4.2.5), on construit par induction l'automate FIDL associé $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$ de la manière suivante :

- **Mot vide.** Si $E = void$ alors $A = (\{q_0\}, q_0, \{q_0\}, \emptyset, \emptyset)$.
- **Message.** Si $E = m(p_1, \ldots, p_n)$ est un message, avec p_i une variable ou un littéral, alors $A = (\{q_0, q_1\}, q_0, \{q_1\}, \{(E, var(E), \emptyset)\}, \{(q_0, (E, \emptyset, \emptyset), q_1)\}\}$.
- **Produit.** Si $E = F.G$, avec $(Q_F, q_{0_F}, T_F, \Sigma_F \times \mathcal{P}\Lambda_F \times \mathcal{P}\mathcal{K}_F, \delta_F)$ et $(Q_G, q_{0_G}, T_G, \Sigma_G, \mathcal{P}\Lambda_G \times \mathcal{P}\mathcal{K}_G, \delta_G)$ les automates associés respectivement à F et G, alors $Q = Q_F \cup Q_G$, $q_0 = q_{0_F}$, $T = T_G \cup T_F$ si q_{0_G} est dans T_G , $T = T_G$ sinon, $\Lambda = \Lambda_F \cup \Lambda_G$, $\mathcal{K} = \mathcal{K}_F \cup \mathcal{K}_G$, $\Sigma = \Sigma_F \cup \Sigma_G$ et

$$\begin{array}{rl} \delta = & \{(q, (m, V, C), q') \in \delta_F\} \\ \cup & \{(q, (m, V, C), q') \in \delta_G \mid q \neq q_{0_G}\} \\ \cup & \{(q, (m, V, C), q') \mid q \in T_F, (q_{0_G}, (m, V, C), q') \in \delta_G\}. \end{array}$$

- **Union.** Si $E = F + G$ avec $(Q_F, q_{0_F}, T_F, \Sigma_F \times \mathcal{P}\Lambda_F \times \mathcal{P}\mathcal{K}_F, \delta_F)$ and $(Q_G, q_{0_G}, T_G, \Sigma_G, \mathcal{P}\Lambda_G \times \mathcal{P}\mathcal{K}_G, \delta_G)$ les automates associés respectivement à F et G, alors $Q = (Q_F \cup Q_G) \backslash \{q_{0_G}, q_{0_F}\} \cup \{q_0\}$, T est l'union de $T_F \cup T_G$ et de $\{q_0\}$ si q_{0_G} est dans T_G ou q_{0_F} est dans T_F, $\Lambda = \Lambda_F \cup \Lambda_G$, $\mathcal{K} = \mathcal{K}_F \cup \mathcal{K}_G$,$\Sigma = \Sigma_F \cup \Sigma_G$ et

$$\begin{array}{rl} \delta = & \{(q, (m, V, C), q') \in \delta_F \mid q \neq q_{0_F}\} \\ \cup & \{(q, (m, V, C), q') \in \delta_G \mid q \neq q_{0_G}\} \\ \cup & \{(q_0, (m, V, C), q) \mid (q_{0_F}, (m, V, C), q) \in \delta_F\} \\ \cup & \{(q_0, (m, V, C), q) \mid (q_{0_G}, (m, V, C), q) \in \delta_G\}. \end{array}$$

- **Mélange.** Si $E = F \parallel G$ avec $(Q_F, q_{0_F}, T_F, \Sigma_F \times \mathcal{P}\Lambda_F \times \mathcal{P}\mathcal{K}_F, \delta_F)$ and $(Q_G, q_{0_G}, T_G, \Sigma_G, \mathcal{P}\Lambda_G \times \mathcal{P}\mathcal{K}_G, \delta_G)$ les automates associés respectivement à F et G, alors $Q = Q_F \times Q_G$, $q_0 = (q_{0_F}, q_{0_G}), T = T_F \times T_G$, $\Lambda = \Lambda_F \cup \Lambda_G$, $\mathcal{K} = \mathcal{K}_F \cup \mathcal{K}_G$,$\Sigma = \Sigma_F \cup \Sigma_G$ et

$$\begin{array}{rl} \delta = & \{((q, p), (m, V, C), (q', p)) \mid (q, (m, V, C), q') \in \delta_F\} \\ \cup & \{((q, p), (m, V, C), (q, p')) \mid (p, (m, V, C), p') \in \delta_G\}. \end{array}$$

- **Étoile.** Si $E = F^*$ avec $(Q_F, q_{0_F}, T_F, \Sigma_F \times \mathcal{P}\Lambda_F \times \mathcal{P}\mathcal{K}_F, \delta_F)$ l'automate associé à F alors $Q = Q_F$, $q_0 = q_{0_F}$, $T = T_F \cup q_0$, $\Lambda = \Lambda_F$, $\mathcal{K} = \mathcal{K}_F$,$\Sigma = \Sigma_F$,

$$\delta_E = \delta_F \cup \{(q, (m, V, C), q') \in \delta_F \mid q \in T_F, \exists (q_{0_F}, (m, V, C), q') \in \delta_F\}.$$

- **Contrainte.** Si $E = (x : Pred \textbf{ in } F)$ avec $(Q_F, q_{0_F}, T_F, \Sigma_F \times \mathcal{P}\Lambda_F \times \mathcal{P}\mathcal{K}_F, \delta_F)$ l'automate associé à F alors $Q = Q_F$, $q_0 = q_{0_F}$, $T = T_F$, $\Sigma = \Sigma_F$, $\Lambda = \Lambda_F \cup \{x\} \cup var(Pred)$, $\mathcal{K} = \mathcal{K}_F \cup \{Pred\}$ et

$$\begin{array}{rl} \delta = & \{(q, (m, V, C), q') \in \delta_F \mid q \neq q_{0_F}\} \\ \cup & \{(q_{0_F}, (m, \{x\} \cup V', \{Pred\} \cup C', q) \mid (q_{0_F}, (m, V', C'), q) \in \delta_F\}. \end{array}$$

Cette construction se distingue de la construction usuelle des automates de mots à partir d'expressions rationnelles (voir par exemple Sakarovitch [140]) par la prise en compte des contraintes qui ne font pas partie des lettres de l'alphabet et par l'introduction systématique d'un nouvel état initial dans la construction de l'union. Ceci nous assure qu'aucune transition n'a pour état d'arrivée l'état initial et qu'il n'y a toujours qu'un seul état initial ce qui facilite la définition de la clôture par préfixe du langage et la construction des expressions de contraintes.

Expressions & Automates Bien Formés

Pour s'assurer de la correction de cette construction et donc de la validité de la sémantique des expressions FIDL, nous montrons que toute expression écrite selon les règles précédemment définies et proprement renommée engendre un automate FIDL bien-formé.

On notera que la règle de bonne formation des expressions et des automates qui nécessite que toutes les variables soient déclarées n'est pas une propriété inductive : elle peut être vraie sur une expression ou un automate et fausse sur une sous-expression ou une partie de l'automate, même si ceux-ci sont par ailleurs bien-formés en regard des autres règles.

Nous montrons donc tout d'abord un lemme technique qui vérifie l'identité entre les variables des expressions et des automates et le respect des autres règles de bonne formation.

Lemma 5.10 *Si $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$ est un automate construit à partir d'une expression proprement renommée et non-ambiguë E, alors les propriétés suivantes sont vérifiées :*

1. *$fv(E) = fv(A)$ et $bv(E) = bv(A)$;*
2. *la relation $\leq$ sur A est un bon ordre partiel (propriété 1 de la définition 5.5) ;*
3. *pour toute variable x appartenant à $bv(A)$, pour toute transition $c \in \mathcal{K}$ telle que $x \in fv(c)$, alors tout chemin depuis q_0 passant par une transition δ contenant c contient une transition δ' antérieure à δ dans laquelle il y a une contrainte c' liant x — voir propriété 2, définition 5.5.*

Preuve 5.11 Nous montrons le lemme par induction sur la stucture des expressions :

– si $E = m(x_1, \ldots, x_n)$ un message, alors on a
  1. $fv(E) = fv(A) = var(m(x_1, \ldots, x_n))$ et $bv(E) = bv(A) = \emptyset$,
  2. $\leq$ est vide donc est un bon ordre partiel,
  3. la propriété 3 est vérifiée puisqu'aucune variable n'est liée ;

– si $E = F + G$,
  1. $bv(A_E) = \bigcup_{c \in \mathcal{K}_E} bv(c)$ par construction, donc $bv(A_E) = \bigcup_{c \in \mathcal{K}_F \cup \mathcal{K}_G} bv(c) = bv(A_F) \cup bv(A_G)$. Par hypothèse d'induction, $bv(A_F) \cup bv(A_G) = bv(F) \cup bv(G) = bv(F + G)$ et par définition on a $bv(A_E) = bv(E)$. De même pour les variables libres, on a :

$$\begin{aligned} fv(A_E) &= \Lambda_E \setminus \bigcup_{c \in \mathcal{K}_E} bv(c) \\ &= \Lambda_F \cup \Lambda_G \setminus \bigcup_{c \in \mathcal{K}_F \cup \mathcal{K}_G} bv(c) \\ &= \Lambda_F \setminus bv(A_F) \cup \Lambda_G \setminus bv(A_G) \\ &= fv(A_F) \cup fv(A_G) = fv(F) \cup fv(G) = fv(E), \end{aligned}$$

  2. $\leq = \leq_F \cup \leq_G$ est un ordre partiel : pour tout $c \in \mathcal{K}_F$ avec $x \in bv(c)$, on a $bv(F) \cap var(G) = \emptyset$ donc il n'existe pas $c' \in \mathcal{K}_G$ tel que $x \in fv(c)$, et réciproquement pour G et F,
  3. par construction de l'automate associé à $F + G$, tout chemin de A_E

$$q_0 \xrightarrow{x} q_1 \xrightarrow[*]{u} q'$$

est tel que

$$\begin{array}{rll} & q_0 \xrightarrow{x} q_1 \xrightarrow[*]{u} q' & \text{dans } A_F \\ \text{ou} & q_0 \xrightarrow{x} q_1 \xrightarrow[*]{u} q' & \text{dans } A_G. \end{array}$$

Par hypothèse d'induction, la propriété est donc vérifiée.

- si $E = F.G$,
  1. idem,
  2. idem,
  3. pour tout $x \in bv(A_E)$, pour tout $c \in \mathcal{K}_{A_E}$ telle que $x \in fv(c)$, alors :
     - si $c \in \mathcal{K}_{A_F}$, la propriété est vraie par hypothèse d'induction,
     - si $c \in \mathcal{K}_{A_G}$,
       - soit $x \in bv(A_G)$ et la propriété est vraie par hypothèse d'induction,
       - soit $x \in fv(A_G)$ et comme E est proprement renommée, $x \notin bv(A_F)$ donc $x \notin bv(A_E)$ ce qui contredit l'hypothèse ;
- si $E = F \parallel G$,
  1. idem,
  2. idem,
  3. idem ;
- si $E = F^*$,
  1. clairement, on a $bv(A_E) = bv(A_F) = bv(F) = bv(E)$ et de même $fv(A_E) = fv(E)$,
  2. $\leq_F = \leq_E$, puisqu'aucune nouvelle contrainte n'est ajoutée. Par ailleurs, si c est une contrainte apparaissant sur une transition $(q_0^F, (m, V, K), q')$, et c' est telle que $c \leq_F c'$, toute transition de E rajoutée par la construction de l'automate préserve la relation,
  3. vrai par hypothèse d'induction sur F ;
- si $E = (x : Pred \textbf{ in } F)$,
  1. on a $bv(E) = bf(F) \cup \{x\}$ et $fv(E) = fv(F) \cup fv(Pred) \setminus \{x\}$. Par construction de A_E,

$$bv(A_E) = bv(A_F) \cup \{x\} = bv(E),$$

et

$$\begin{aligned} fv(A_E) &= \Lambda_F \cup var(Pred) \setminus (bv(A_F) \cup \{x\}) \\ &= (fv(F) \cup var(Pred)) \setminus \{x\} \\ &= var(E) \setminus bv(F) \cup \{x\} \\ &= fv(E), \end{aligned}$$

  2. $\leq_E = \leq_F \cup (x : Pred, x : Pred) \cup \{(x : Pred, c') \mid c' \in \mathcal{K}_F, x \in fv(c')$ ou $\exists c \in \mathcal{K}_F, c \leq_F c'$ et $x \in fv(c)\}$ est bien une relation d'ordre partiel :
     - $\leq_E$ est réflexive : par hypothèse d'induction, $\leq_F$ est réflexive et $(x : Pred, x : Pred) \in \leq_E$,
     - $\leq_E$ est antisymétrique : quel que soit $c \neq x : Pred$, si $x : Pred \leq c$, alors on ne peut avoir $c \leq x : Pred$ car aucun chemin ne repasse par q_0^E et $\leq_F$ est antisymétrique par hypothèse,
     - $\leq_E$ est transitive par construction si $\leq_F$ est transitive ;
  3. enfin, d'après la construction de l'automate de E, section 5.2.2, toutes les transitions partant de l'état initial de E contiennent la contrainte $x : Pred$ donc la propriété 3 est vérifiée. ■

On peut donc en déduire aisément la proposition suivante :

Proposition 5.12 Si E est une expression bien-formée, alors l'automate A construit à partir de E est bien formé.

Preuve 5.13 On vérifie les propriétés de bonne formation de A :
- $\leq$ est bon ordre partiel d'après le lemme 5.10 ;
- la propriété 2, définition 5.5 est vraie pour les variables liées, or $bv(A) = var(E) \setminus \{\texttt{Trace}\}$ donc elle est vraie pour toutes les variables de A ;
- la propriété 3, définition 5.5, est vraie car $fv(A) = \Lambda \setminus bv(A) = \Lambda \setminus (var(E) \setminus \{Trace\})$. ■

Le langage des expressions FIDL, appelé ensemble de traces, est ainsi défini en fonction du langage reconnu par un automate. Ce langage est en fait la clôture par préfixe du langage de l'automate car cela correspond à la notion d'observation du comportement d'un système en fonction d'événements extérieurs. Cette observation étant finie et pouvant s'interrompre arbitrairement, tout début de comportement correct est un comportement correct.

Définition 5.14 Pour toute expression FIDL E, l'*ensemble de traces* de l'expression E, noté $\mathcal{T}(E)$ est défini comme :

$$\mathcal{T}(E) = Pref(L_{A_E}).$$

5.3 Vérification & Validations

Les automates *FIDL* ont été définis dans un but précis : spécifier le comportement d'éléments d'un système réparti et permettre la vérification d'une implantation concrète de ces spécifications. Cette vérification peut être réalisée de nombreuses manières dont les plus importantes dans le cas de modèles basés sur des systèmes de transitions sont le *contrôle de modèle — model-checking —* et le test de conformité — *conformance testing*. Dans les deux cas, on cherche à atteindre un état particulier ou un représentant d'un ensemble d'états particuliers du système dans lequel une certaine propriété est vraie. Dans le cas du contrôle de modèle, ce parcours est fait sur le modèle lui-même — la spécification — pour valider celle-ci. Dans le cas du test de conformité, ce parcours est fait en parallèle sur la spécification et l'implantation. Clairement, les deux méthodes produisent un résultat en un temps fini si la propriété est vérifiée pour un certain état accessible, ou si le nombre d'états accessibles est fini.

Le modèle général des automates FIDL décrit de manière compacte un ensemble d'états potentiellement infini. L'alphabet d'un automate est obligatoirement fini (voir définition 5.1) mais son langage est potentiellement infini : il utilise des ensembles de données arbitraires, donc potentiellement infini, et des prédicats logiques et des fonctions qui peuvent ne pas se terminer. Nous examinons donc dans cette section deux formes de simplification des automates FIDL permettant de faire en sorte que l'espace d'état à vérifier demeure fini.

5.3.1 Domaines finis

Une première solution consiste à rendre le domaine des valeurs possibles $\mathcal{D}$ fini. Bien qu'elle soit triviale, nous donnons ici la preuve de la reconnaissabilité des langages reconnus par les automates FIDL dans le cas où les ensembles de données sont finis.

Proposition 5.15 (Reconnaissabilité) Si $\mathcal{D}$ le domaine des variables est fini et si toutes les fonctions auxiliaires terminent, alors pour tout automate FIDL A, le langage accepté par A, $L_A \subseteq \bar{\mathcal{E}}^*$ est reconnaissable.

Preuve 5.16 Soit $A = (Q, q_0, T, \Sigma \times \mathcal{P}\Lambda \times \mathcal{P}\mathcal{K}, \delta)$ un automate FIDL déterministe. Si $\mathcal{D}$ est fini alors $\bar{\mathcal{E}}$ est fini donc Σ est fini car il ne contient qu'un nombre fini de variables : étant donné un message m d'arité k, l'ensemble des valeurs possibles des paramètres de m est $\mathcal{D}^k$.

Soit $A' = (Q_A, q_0^A, T_A, \{m(v_1, \ldots, v_n) \in \bar{\mathcal{E}} \mid \exists m(x_1, \ldots, x_n) \in \Sigma\}, \delta_A)$ avec
- $Q_A = Q \times \{\Lambda \rightarrow \mathcal{D} \cup \{\bot\}\}$, le produit des états de l'automate avec l'ensemble des valuations possibles pour toutes les variables déclarées dans A. Nous notons (q, val) un élément de Q_A ;
- $q_0^A = (q_0, \{x \mapsto \bot \mid x \in \Lambda\})$;
- $T_A = \{(q, val) \in Q_A \mid q \in T\}$,
- $\delta_a = \{((q, val), m, (q', val')) \mid m \in \bar{\mathcal{E}}$ et $\exists \sigma = (V, val_\sigma, pred), \sigma' = (V', val'_\sigma, pred'), val_\sigma \subseteq val, val_{\sigma'} \subseteq val', (q, \sigma) \xrightarrow{m} (q', \sigma')\}$.

Q_A et Σ étant finis, A' est un automate d'état fini et son langage $L_{A'}$ est évidemment reconnaissable.

Montrons tout d'abord que $L_A \subseteq L_{A'}$. Par induction sur la longueur des mots, nous montrons que pour tout u tel que $(q_0, \sigma_0) \xrightarrow{u} (q, \sigma)$ dans A, il existe $(q, val) \in Q_A$ tel que $q_0^A \xrightarrow{u} (q, val)$ dans A'.

Soit u un mot de L_A :

- si $\#u = 0$ alors $u = \epsilon$ donc $q_0 \in T$ et $q_0^A \in T_A$, donc $u \in L_{A'}$;
- si $\#u > 0$, $u = v.a$, donc $(q_0, \sigma_0) \xrightarrow{v} (q, \sigma) \xrightarrow{a} (q', \sigma')$ dans A. Par hypothèse d'induction, $q_0^A \xrightarrow{v} (q, val)$ avec $val_\sigma \subseteq val$ et comme $(q, \sigma) \xrightarrow{a} (q', \sigma')$, on peut construire $val' = val_{\sigma'} \cup \{x \mapsto \perp \mid x \in \Lambda \setminus V_{\sigma'}\}$ et l'on a donc $(q, val) \xrightarrow{a} (q', val')$ et l'on déduit $q_0^A \xrightarrow{u} (q', val')$.

On a donc $L_A \subseteq L_{A'}$.

Inversement, pour tout mot $u \in L_{A'}$, $u \in L_A$:

- si $\#u = 0$, $q_0^A \in T_{A'}$ donc $q_0 \in T_A$ et $u \in L_A$;
- si $\#u > 0$, on a $u = va$ donc $q_0^A \xrightarrow{v} (q, val) \xrightarrow{a} (q', val')$ dans A'. Par hypothèse d'induction, on a $(q_0, \sigma_0) \xrightarrow{v} (q, \sigma)$ dans A. On pose $val_{\sigma'} = val'$ et l'on a donc $(q, \sigma) \xrightarrow{a} (q', \sigma')$ d'où $u \in L_A$.

On a donc $L_{A'} \subseteq L_A$. ■

La variable globale `Trace` ne pouvant être utilisée que comme argument d'un prédicat ou d'une fonction, et le codomaine de ces fonctions étant par hypothèse fini, son utilisation dans une contrainte ne pose pas de problème particulier.

Le dépliage de l'automate FIDL, s'il est valide sur le plan théorique puisqu'il nous permet de nous assurer que le nombre d'états à explorer est fini, est peu pratique. On va donc garder la formulation initiale d'un automate avec contraintes et variables, et considérer une solution alternative au dépliage global dans laquelle les mots reconnus sont les solutions de problèmes de satisfaction de contraintes ou CSP — *Constraint Satisfaction Problem* en anglais.

Instantiation dans un automate FIDL

Étant donné un automate FIDL bien formé, une transition candidate

$$d = (q, (m(x_1, x_2, \ldots, x_n), V, K), q'),$$

et un environnement courant σ, le franchissement de la transition est possible si l'on peut trouver une affectation des variables x_i pour $1 \leq i \leq n$ qui satisfasse les contraintes de σ — voir définition 5.1.5.

Soit $\vec{y} = (y_1, y_2, \ldots, y_m)$, $m \leq n$ le vecteur constitué uniquement des variables de $(x_1, x_2, \ldots, x_n)$, alors pour chaque y_i, $1 \leq i \leq m$,

- soit y_i est défini dans σ avec une valeur v et y_i n'est pas déclaré dans V ;
- soit y_i est indéfini — mais déclaré — et il existe $c_i = P(y_i, f(\bar{z}))$, une expression de contrainte dans $pred_\sigma \cup K$ pour y_i.

Soit K' l'ensemble des contraintes sur les variables de $\vec{y}$ telles que $\mathrm{val}_\sigma(y_i) = \perp$. Les contraintes n'étant par construction pas récursives et l'utilisation anticipée de variables étant interdite, on définit l'*ensemble de contraintes* pour la transition d, K_d, comme étant la clôture transitive de l'ensemble K' par la *relation de dépendance* $\rightarrow$ telle que

$$\begin{array}{c} y \text{ op } f(z_1, \ldots, z_n) \rightarrow y' \text{ op } f'(z'_1, \ldots, z'_m) \\ \Updownarrow \\ \exists i, 1 \leq i \leq n, z_i = y'. \end{array}$$

Définition 5.17 (CSP d'instanciation) Le *problème de satisfaction de contrainte* pour l'*instanciation d'un message* sur la transition d, dans un environnement σ, est un triplet (X, K_d, dom) avec :

- X un ensemble fini de variables contraintes ;
- K_d un ensemble fini de contraintes de la forme $P(x, f(\bar{y}))$ avec P un prédicat booléen, x une variable de X et $\bar{y}$ un n-uplet de valeurs et de variables de $X \cup \mathcal{D}$;
- $dom : X \rightarrow \mathcal{P}(\mathcal{D})$ une fonction de *domaine* assignant les valeurs admissibles pour les variables de X.

Une solution à ce problème, si elle existe, est une application $s : X \rightarrow \mathcal{D}$ telle que toutes les contraintes soient évaluées `vrai` et que toutes les images de variables $s(x)$ appartiennent au domaine de celles-ci, $dom(x)$.

Par hypothèse de finitude du domaine des variables, le codomaine de dom est fini et ce problème peut être résolu en utilisant des algorithmes classiques de résolution de CSP (*cf.* infra) ou peut s'avérer insoluble ce qui interdit le franchissement de la transition.

Instantiation dans un automate synchronisé

Dans le cas d'un automate synchronisé (voir section 5.1.6), il est nécessaire de générer un message qui soit susceptible de franchir simultanément plusieurs transitions, plus précisément toutes les transitions des automates dont l'alphabet contient le message. Ce problème se traite simplement en augmentant les CSP individuels de chaque automate de contraintes d'égalité entre les variables communes pour obtenir un CSP global.

Étant donnés n automates $A_1, A_2, \ldots, A_n$ dont les états sont $(q_1, \sigma_1), (q_2, \sigma_2), \ldots, (q_n, \sigma_n)$, et tels qu'il existe des transitions $(q_0, ((c, p, c', p', k, m, \vec{x_i}, d), V, K), q'_0)$, pour $1 \leq i \leq n$, on définit tout d'abord n instances de CSP $(X_i, K_{d_i}, dom_i), 1 \leq i \leq n$ comme précédemment. Ces instances sont jointes en un CSP unique $P = (X, K, dom)$ comme suit :

- on renomme les variables de sorte que les ensembles X_i soient deux à deux disjoints ;
- soit $l = |\vec{x_1}|$ le nombre de paramètres formels de la méthode m, pour chaque $j, 1 \leq j \leq l$:
  - s'il existe x_{qj} et x_{pj}, $1 \leq p, q \leq n$ tels que soit $val_{\sigma_q}(x_{qj}) \neq val_{\sigma_q}(x_{pj})$ soit $x_{qj} \neq x_{pj}$ et $x_{qj}, x_{pj} \in \mathcal{D}$, alors P n'a pas de solution,
  - sinon, on ajoute à X une nouvelle variable z et à K une nouvelle contrainte $x_{ij} = z$, pour $1 \leq i \leq n$.

Le système P obtenu est ensuite résolu par les mêmes techniques que dans le cas d'un automate unique.

Résolution de CSP

Il existe de nombreuses techniques de résolution de CSP dont on trouvera un panorama dans Barták [22] et Kumar [81]. Pour les besoins de cette thèse, nous nous contenterons d'un rapide survol des différentes techniques utilisables. L'implantation concrète choisie est détaillée dans le chapitre 8.

Les différentes techniques de résolution de systèmes de contraintes se répartissent en trois grandes familles aux frontières éminemment perméables :

1. les *techniques de recherche* dans lesquelles on explore l'espace des solutions jusqu'à trouver la ou les solutions ;
2. les *techniques de consistance* qui *a contrario* vont élaguer l'espace des solutions possibles en explorant les incompatibilités ;
3. les *techniques de recherche locale*, le plus souvent basées sur des heuristiques et/ou des processus stochastiques.

Tous les algorithmes de la première famille se basent sur la technique de *Générer-et-Tester* (GT) : on génère une solution candidate — une valuation des variables — et on teste si le candidat est effectivement solution du CSP. Cet algorithme extrêmement inefficace est amélioré par le *Backtracking* (BT) qui élague de la recherche les valuations partielles incohérentes. En combinant cette technique de base avec la deuxième famille d'algorithmes, on obtient les algoritmes de *Backjumping* (BJ), *Backmarking* (BM) et *Backchecking* (BC) qui augmentent l'élagage par examen des contraintes effectivement violées.

Les techniques de consistance sont basées sur une représentation en graphe du CSP où un nœud représente une variable — et son domaine de valeurs possibles — et un arc une contrainte unaire ou binaire. Les algorithmes *Consistance-de-Nœuds* (NC), *Consistance-d'Arcs* (AC) et *Consistane-de-Chemins* (PC) parcourent répétitivemant le graphe pour élaguer des domaines de variables les valeurs qui ne peuvent satisfaire les contraintes auxquelles cette variable est liée. La différence réside essentiellement dans la profondeur de la recherche qui est menée après chaque mise à jour d'une variable. Ces techniques étant généralement incomplètes, elles se combinent avec la première famille comme on l'a déjà vu pour

l'élagage. Le Backtracking présentant l'inconvénient de détecter tardivement les incohérences, on peut lui adjoindre une technique de Lookahead qui va élaguer les incohérences introduites par une certaine valuation avant la poursuite de la recherche d'une solution en appliquant une des techniques de maintien de la consistance.

Enfin, les techniques de recherches locales dont la plus connue est la *Descente de Gradient — Hill-Climbing* — cherchent à améliorer une solution partielle en accroissant par étape le nombre de contraintes satisfaites. La *Minimisation des Conflits* cherche en plus les solutions voisines qui minimisent le nombre de conflits. Ces méthodes de base ayant tendance à se retrouver « piégées » dans des minima locaux, on peut les améliorer soit en introduisant des sauts aléatoires — technique du Random-Walk — soit en maintenant une liste de solutions provisoirement interdites — méthode de la *Recherche Taboue* avec ou sans critère d'aspiration. On trouvera dans Michalewicz et Fogel [111] une étude très complète et récente des différentes méthodes heuristiques permettant de résoudre efficacement des problèmes d'exploration d'espaces d'états tels que les CSP.

5.3.2 Domaines reconnaissables

Une autre solution consiste à considérer l'ensemble des domaines des variables de manière symbolique. On peut ainsi espérer pouvoir manipuler une représentation finie d'un ensemble infini et montrer ou mettre en évidence des propriétés sur des ensembles infinis de valeurs en n'explorant qu'une partie finie des représentations possibles. Les langages reconnaissables sont une classe particulière de représentation finie d'ensembles infinis que l'on peut utiliser pour représenter un domaine de valeurs possibles sous une forme symbolique. Cette approche a été utilisée, entre autres, avec succès dans Boigelot et Godefroid [32] pour la représentation de files de messages non bornées dans les protocoles de communication, et dans Wolper et Boigelot [163] pour la représentation de formules de l'arithmétique de Presburger et donc la résolution de systèmes d'équations linéaires dans les entiers ou la représentation d'ensembles d'entiers linéairement contraints.

C'est un fait bien connu qu'il existe une relation intime entre diverses catégories de logiques et diverses formes d'automates, au sens où les modèles des unes sont les langages des autres. Par exemple la logique MSO, *logique monadique du second ordre*, a pour modèle les langages reconnaissables par un automate fini. L'arithmétique de Presburger, c'est-à-dire les formules du premier ordre sur les entiers utilisant le prédicat $\leq$ et la fonction $+$, est, elle aussi, liée aux automates. Dernier exemple classique, les modèles de formules de la logique temporelle linéaire sont des mots infinis reconnaissables par un *automate de Büchi*.

Sous l'hypothèse que les ensembles représentant des types soient reconnaissables et que les fonctions et prédicats utilisés dans les expressions de contraintes soient limités aux fonctions exprimables sous forme de transductions rationnelles, alors tout environnement d'un automate FIDL contient une valuation assignant à chaque variable un ensemble reconnaissable. On peut donc voir un automate FIDL comme définissant une relation entre des séquences d'enveloppes de messages et des valeurs contenues dans ces messages, valeurs qui sont restreintes à des ensembles reconnaissables donc représentés par un automate d'états finis. Cette représentation nous permettrait de spécifier et vérifier des comportements d'automates FIDL potentiellement infinis en restreignant la vérification ou la validation aux états de l'automate représentant les valeurs. Si l'on est de plus capable d'évaluer ou d'approcher le résultat de l'itération d'une certaine séquence de messages sur l'environnement, autrement dit si l'on sait évaluer le résultat de l'itération des transductions représentant les contraintes de l'automate, alors on est à même d'étudier de manière séparées les parties *contrôle* et *données* de l'automate FIDL, réunies dans une structure commune qui est le graphe de l'automate. On aurait ainsi la possibilité de définir aisément les messages en fonction des valeurs, ou les valeurs en fonction des messages.

5.4 Conclusion

Les automates que nous avons définis dans ce chapitre sont proches des *Machines d'États Fini Étendues — Extended Finite State Machines* ou EFSM — couramment utilisées pour la modélisation de sémantiques formelles de langages de haut-niveau comprenant des interactions entre différentes entités dans un système,

tels que les Statecharts. Les EFSM posent toutefois des problèmes en termes de composition dans la mesure où ils comprennent une notion explicite d'événements émis et reçus qui peuvent permettre à plusieurs EFSM de communiquer de manière synchrone ou asynchrone. La sémantique des déclenchements de transition n'est toutefois pas toujours très claire lorsque les situations deviennent un peu complexes du fait de questions de priorité entre événements et de différences entre sémantiques de petit pas et sémantiques de grand pas.

Nous avons choisi une formalisation qui est proche de la théorie classique des langages formels et plus particulièrement des langages rationnels pour lesquels il existe une vaste littérature, de nombreux outils et techniques de vérification et une sémantique de la composition simple basée sur des opérations élémentaires internes à la théorie des langages. Nous verrons dans la suite de ce travail que ces automates s'apparentent plutôt à des IOLTS symboliques.

Chapitre 6

Sémantique du modèle FIDL

Les automates FIDL détaillés au chapitre précédent constituent les briques de base de la modélisation des entités dans le modèle d'architecture de composants sur lequel nous travaillons. Nous n'avons toutefois encore rien précisé de la formalisation de ces entités, de la manière dont elles peuvent s'agencer et comment une spécification décrit leur comportement.

Ce chapitre présente donc la sémantique formelle des éléments modélisés : interfaces, composants, connexions, assemblages et composites. Par sémantique, nous entendons définir le *sens* précis de chacune des constructions syntaxiques qu'il est possible d'écrire selon la grammaire FIDL. Ce sens est défini en termes de *langages* clos par préfixes appelés aussi *ensembles de traces*, sur des alphabets dont les lettres possèdent une structure particulière. La première partie décrit la sémantique des éléments atomiques d'un modèle : événements, interfaces, composants primitifs. La deuxième partie s'attachera à définir le comportement d'un système lorsqu'il est constitué de plusieurs éléments mis en relation. Nous démontrerons en particulier un résultat de préservation de la correction des composants lors de leur composition. La dernière partie sera consacrée aux propriétés de sous-typage et d'héritage et à la manière dont celles-ci sont définies dans notre modèle.

6.1 Éléments atomiques

À partir de la définition de la structure de l'alphabet — des événements — nous définissons les alphabets et langages associés à une interface et à un composant. Ces événements présentent la particularité d'encapsuler dans chaque lettre la connectique du système dans lequel est produit l'événement sous la forme de variables identifiant précisément l'émetteur et le récepteur d'un message. Au fur et à mesure du processus de composition, ces variables sont instanciées avec les identités concrètes des éléments qu'elles représentent.

6.1.1 Événements

Nous précisons dans cette section la structure de l'ensemble $\mathcal{X}$ (voir chapitre 5, section 5.1.3) des *enveloppes* des lettres reconnues par un automate FIDL. Chacune de ces enveloppes représente une partie d'un *message* envoyé d'un émetteur vers un récepteur et est appelée *événement*.

Définition 6.1 (Événement) Un événement est un n-uplet $(\gamma_1, \varrho_1, \gamma_2, \varrho_2, k, m, v)$ où :

- $\gamma_1, \varrho_1, \gamma_2, \varrho_2$ identifient une connexion, γ_1, γ_2 étant respectivement l'identité des composants client et serveur et ϱ_1, ϱ_2 identifiant les ports connectés dans les composants respectifs ;
- $k \in \{\texttt{call}, \texttt{return}, \texttt{exception}, \texttt{async}\}$ le *genre* de l'événement, respectivement un appel de méthode, un retour d'appel, une exception ou un message asynchrone ;
- m le nom de l'événement, qui peut être un nom de méthode ou de message asynchrone. À chaque nom d'événement m est attachée une signature qui est un n-uplet de types $(T_1, \ldots, T_n)$ où n dépend de m ;

– $v \in \{\texttt{emit}, \texttt{receive}\}$ le *point de vue* sur l'événement, respectivement une *émission* et une *réception*.

L'ensemble de tous les messages est noté comme précédemment $\mathcal{E}$ et il est construit à partir de la structure des événements et d'un contenu dépendant de l'arité et de la signature des messages. Un message est noté $(\gamma_1, \varrho_1, \gamma_2, \varrho_2, k, m, \vec{x}, v)$ où $\vec{x}$ est le *contenu* du message, les autres éléments du n-uplet constituant son enveloppe. Le nombre et le type des éléments du vecteur $\vec{x}$ dépendent de la signature associée à m.

L'ensemble de tous les *messages clos* qui sont tous les messages dont le corps ne contient que des littéraux, est noté $\bar{\mathcal{E}}$:

$$\bar{\mathcal{E}} = \{(\gamma_1, \varrho_1, \gamma_2, \varrho_2, k, m, (x_1, x_2, \ldots, x_n), v) \in \mathcal{E} \mid \forall i, 1 \leq i \leq n, x_i \in \mathcal{D}_{T_i}\}.$$

On définit $h_\lambda : \mathcal{E} \rightarrow \mathcal{X}$ le *morphisme d'abstraction* comme :

$$h_\lambda((\gamma_1, \varrho_1, \gamma_2, \varrho_2, k, m, \vec{x}, v)) = \begin{cases} (\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{return}, m, v), \text{ si } k = \texttt{exception}, \\ (\gamma_1, \varrho_1, \gamma_2, \varrho_2, k, m, v) \text{ sinon}, \end{cases} \tag{6.1}$$

et $\bar{h} : \mathcal{E} \rightarrow \mathcal{E}$ le *morphisme miroir* :

$$\bar{h} = h^{\texttt{receive},\texttt{emit}}_{\texttt{emit},\texttt{receive}}. \tag{6.2}$$

Dans la définition de h_λ, nous abandonnons la distinction entre les messages de retour d'appels de méthodes et les messages d'exception, considérant que les exceptions ne sont qu'un type de retour possible différent d'un retour normal.

Pour simplifier la notation, nous pourrons être amenés à dénoter le n-uplet de variables de connexions $(\gamma_1, \varrho_1, \gamma_2, \varrho_2)$ par le symbole unique κ lorsque le détail des connexions n'est pas nécessaire.

6.1.2 Types de ports

Interfaces

Une interface est un *contrat* syntaxique et comportemental sur un canal de communication entre deux entités d'un système. Elle décrit l'alphabet des messages pouvant être échangés entre le client et le fournisseur du service défini par l'interface et le langage associé à cet alphabet.

Définition 6.2 (Interface) Une interface I est un couple noté $\langle meth(I), \mathcal{T}(I)\rangle$ où $meth(I)$ est un ensemble fini de *méthodes* et $\mathcal{T}(I)$ un langage clos par préfixe sur un certain alphabet $\mathrm{alph}(I)$.

Toute méthode m appartenant à $meth(I)$ est définie par :
– son arité $\mathsf{ar}(m) \in \mathbb{N}$;
– sa signature $\mathsf{sig}(m)$ qui est un vecteur de taille $\mathsf{ar}(m)$ dont les éléments sont des types $T_1, \ldots, T_{ar(m)}$;
– ses exceptions, $\mathsf{exceptions}(m)$, un vecteur de taille k finie.

En supposant que les signatures sont ordonnées selon l'ordre usuel : d'abord les paramètres en mode **in**, ensuite les **inout** enfin les paramètres en mode **out**, $in(m)$ est l'indice du dernier paramètre en mode **in** ou **inout** et $out(m)$ est l'indice du premier paramètre en mode **inout** ou **out**.

Pour une méthode m de signature $\mathsf{sig}(m)$, on définit les ensembles suivants :
– $\mathsf{inpar}(m)$ est l'ensemble des vecteurs de paramètres en mode **in** et **inout** de la méthode m comprenant des valeurs et des variables :

$$\mathsf{inpar}(m) = \{\vec{x} \in (\mathcal{V} \cup \mathcal{D})^{in(m)} \mid x[i] \in (\mathcal{D}_{\mathsf{sig}(m)[i]} \cup \mathcal{V})\};$$

– $\mathsf{outpar}(m)$ l'ensemble des vecteurs de paramètres en mode **inout** et **out** de la méthode m comprenant des valeurs et des variables ainsi éventuellement qu'une valeur de retour de la méthode appelée

$$\mathsf{outpar}(m) = \{\vec{x} \in (\mathcal{V} \cup \mathcal{D})^{\mathsf{ar}(m)-out(m)+1} \mid x[i] \in (\mathcal{D}_{\mathsf{sig}(m)[i+out(m)-1]} \cup \mathcal{V})\};$$

– $\overline{\mathsf{inpar}}(m)$ l'ensemble des vecteurs de paramètres en mode **in** et **inout** de la méthode m ne contenant que des valeurs littérales :

$$\overline{\mathsf{inpar}}(m) = \{\vec{x} \in \mathcal{D}_{T_1} \times \cdots \times \mathcal{D}_{T_{in(m)}}\};$$

– $\overline{\mathsf{outpar}}(m)$ l'ensemble des vecteurs de paramètres en mode **inout** et **out** de la méthode m ne contenant que des valeurs littérales plus la valeur de retour si elle est différente de `void` :

$$\overline{\mathsf{outpar}}(m) = \{\vec{x} \in \mathcal{D}_{T_{out(m)}} \times \cdots \times \mathcal{D}_{T_{\mathsf{ar}(m)}}\}.$$

Définition 6.3 (Alphabet d'une interface) L'alphabet d'une interface $I = \langle meth(I), \mathcal{T}(I)\rangle$ est défini comme :

$$\begin{aligned}\mathrm{alph}(I) = \{(\gamma_1, \varrho_1, \gamma_2, \varrho_2, d, m, \vec{x}, v) \mid\ & m \in meth(I),\\ & (d, \vec{x}, v) \in \{\texttt{call}\} \times \overline{\mathsf{inpar}}(m) \times \{\texttt{receive}\}\\ & \cup\{\texttt{return}\} \times \overline{\mathsf{outpar}}(m) \times \{\texttt{emit}\}\\ & \cup\{\{\texttt{exception}\} \times (E \times \mathcal{D}_E) \times \{\texttt{emit}\}, \forall E \in \mathsf{exceptions}(m)\}\}.\end{aligned}$$

Événements asynchrones

Le cas des événements asynchrones est le plus simple : un événement asynchrone est simplement un n-uplet de valeurs correctement typées, donc respectant la structure de la définition de l'événement.

Définition 6.4 (Type d'événement asynchrone) Un type d'événement asynchrone S est défini par un n-uplet de types $(T_1, \ldots, T_n)$. L'alphabet associé à un événement asynchrone S, $\mathrm{alph}(S)$, est défini simplement comme suit :

$$\mathrm{alph}(S) = \{(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{async}, S, (x_1, \ldots, x_n), \texttt{emit}) \mid \forall x_i, 1 \leq i \leq n, x_i \in \mathcal{D}_{T_i}\}.$$

Il est constitué de tous les messages *émis* identifiés par le nom de S contenant des valeurs correctes pour la structure de l'événement. Le langage associé à un type d'événement asynchrone est simplement

$$\mathcal{T}(S) = \mathrm{alph}(S)^*.$$

6.1.3 Composants

Un composant définit une entité d'un système susceptible d'être réalisée à l'exécution par une — ou plusieurs dans le cas des composites — instances de composants réels. C'est donc la description d'un ensemble de services *offerts* et requis au travers de ports *identifiés* et *typés*, soit par une interface, soit par un type d'événement asynchrone.

Nous définissons donc tout d'abord la notion de port avant de définir la sémantique d'un composant.

Définition 6.5 (Port) Un port p est un quadruplet (n, c, T, g) où n est l'identité du port, c l'identité du composant auquel ce port appartient, T son type qui peut être une interface ou un type d'événement asynchrone et g son *genre* (`receptacle`, `facet`, `source` ou `sink`).

L'alphabet et l'ensemble de traces associés à un port $p = (n, c, T, g)$, notés respectivement $\mathrm{alph}(p)$ et $\mathcal{T}(p)$, sont définis comme suit :

– si $p = (f, c, I, \texttt{facet})$, alors $\mathrm{alph}(p) = h^{c,f}_{\gamma_2,\varrho_2}(\mathrm{alph}(I))$ et $\mathcal{T}(p) = h^{c,f}_{\gamma_2,\varrho_2}(\mathcal{T}(I))$;
– si $p = (r, c, I, \texttt{receptacle})$, alors $\mathrm{alph}(p) = \bar{h}(h^{c,r}_{\gamma_1,\varrho_1}(\mathrm{alph}(I)))$ et $\mathcal{T}(p) = \bar{h}(h^{c,r}_{\gamma_1,\varrho_1}(\mathcal{T}(I)))$;
– si $p = (s, c, S, \texttt{source})$, alors $\mathrm{alph}(p) = h^{c,s}_{\gamma_1,\varrho_1}(\mathrm{alph}(S))$ et $\mathcal{T}(p) = h^{c,s}_{\gamma_1,\varrho_1}(\mathcal{T}(S))$:
– si $p = (s, c, S, \texttt{sink})$, alors $\mathrm{alph}(p) = \bar{h}(h^{c,s}_{\gamma_2,\varrho_2}(\mathrm{alph}(S)))$ et $\mathcal{T}(p) = \bar{h}(h^{c,s}_{\gamma_2,\varrho_2}(\mathcal{T}(S)))$.

Un port est simplement une instance nommée et orientée d'un type d'interface ou d'événement asynchrone, dans un certain contexte englobant. Rappelons que $\bar{h}$ désigne le *morphisme miroir* qui inverse les messages émis et les messages reçus.

Définition 6.6 (Composant) Un composant C est défini par un couple $\langle Port(C), \mathcal{T}(C)\rangle$ où $Port(C)$ est un ensemble fini de ports (p, γ, T, g) dont les noms sont deux à deux distincts et $\mathcal{T}(C)$ un langage clos par préfixe sur l'alphabet $\mathrm{alph}(C)$ défini simplement :

$$\mathrm{alph}(C) = \bigcup_{p \in Port(C)} \mathrm{alph}(p).$$

L'alphabet d'un composant est donc l'alphabet des différents ports qui le composent. L'identité du composant est la variable γ qui représente n'importe quelle instance du composant C.

Pour un composant donné C, nous noterons :
- $\mathcal{F}(C) = \{(f, \gamma, I, \texttt{facet}) \in Port(C)\}$, l'ensemble de ses facettes ;
- $\mathcal{R}(C) = \{(r, \gamma, I, \texttt{receptacle}) \in Port(C)\}$, l'ensemble de ses réceptacles ;
- $\mathcal{S}ource(C) = \{(s, \gamma, S, \texttt{source}) \in Port(C)\}$, l'ensemble de ses sources ;
- et $\mathcal{S}ink(C) = \{(s, \gamma, S, \texttt{sink}) \in Port(C)\}$, l'ensemble de ses puits.

On peut alors décrire une instance particulière du composant C en instanciant simplement la variable γ avec l'identité de l'instance de composant.

Définition 6.7 (Instance de composant) Une instance du composant $C = \langle Port(C), \mathcal{T}(C)\rangle$ dont l'identité est c est définie par le couple :

$$c = \langle Port(C), h^c_\gamma(\mathcal{T}(C))\rangle,$$

et son alphabet est :

$$\mathrm{alph}(c) = h^c_\gamma(\mathrm{alph}(C)).$$

6.1.4 Expressions FIDL

Les comportements des interfaces et des composants sont décrits au moyen d'expressions FIDL dont la syntaxe a été introduite dans la section 4.2.5. Nous donnons ici (tableau 6.1) le lien entre la définition formelle des messages et la notation utilisée. Dans la notation $r.m(\bar{x})$, le n-uplet de paramètres $\bar{x}$ représente « l'union » des deux vecteurs $\vec{y} \in inpar(m)$ et $\vec{z} \in outpar(m)$ en identifiant $\vec{z}_1 \ldots \vec{z}_k$ et $x_{out(m)} \ldots x_{ar(m)}$. Le type des paramètres du vecteur $\vec{x}$ se déduit immédiatement de son utilisation dans les messages.

Composant	Notation	Événement
Appel sur réceptacle	$r \rightarrow m(\bar{x})$	$(\gamma, r, \gamma_2, \varrho_2, \texttt{call}, m, \vec{x}, \texttt{emit})$
Retour correspondant	$r \leftarrow m(\bar{x})$	$(\gamma, r, \gamma_2, \varrho_2, \texttt{return}, m, \vec{x}, \texttt{receive})$
Appel/retour sur réceptacle	$r.m(\bar{x})$	$(\gamma, r, \gamma_2, \varrho_2, \texttt{call}, m, \vec{y}, \texttt{emit})$ $(\gamma, r, \gamma_2, \varrho_2, \texttt{return}, m, \vec{z}, \texttt{receive})$
Appel reçu sur facette f	$f \rightarrow m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma, f, \texttt{call}, m, \vec{x}, \texttt{receive})$
Retour correspondant	$f \leftarrow m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma, f, \texttt{return}, m, \vec{x}, \texttt{emit})$
Appel/retour sur facette f	$f.m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma, f, \texttt{call}, m, \vec{y}, \texttt{receive})$ $(\gamma_1, \varrho_1, \gamma, f, \texttt{return}, m, \vec{z}, \texttt{emit})$
Envoi d'événement	$so.[\bar{x}]$	$(\gamma, so, \gamma_2, \varrho_2, \texttt{event}, Type(so), \vec{x}, \texttt{emit})$
Réception d'événement	$si.[\bar{x}]$	$(\gamma_1, \varrho_1, \gamma, si, \texttt{event}, Type(si), \vec{x}, \texttt{receive})$
Interface		
Appel	$\rightarrow m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{call}, m, \vec{x}, \texttt{receive})$
Retour	$\leftarrow m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{return}, m, \vec{x}, \texttt{emit})$
Exception	$\leftarrow m\langle E, \bar{x}\rangle$	$(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{exception}, m, (E, \bar{x}), \texttt{emit})$
Appel/retour	$m(\bar{x})$	$(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{call}, m, \vec{y}, \texttt{receive})$ $(\gamma_1, \varrho_1, \gamma_2, \varrho_2, \texttt{return}, m, \vec{z}, \texttt{emit})$

FIG. 6.1 – Correspondance alphabet/événements.

Par ailleurs, nous définissons précisément l'alphabet d'une expression car il n'est pas égal à l'alphabet induit.

Définition 6.8 (Alphabet d'une expression) L'alphabet d'une expression E, $\mathrm{alph}(E)$, est défini inductivement par :

$$\begin{aligned}
\mathrm{alph}(\texttt{void}) &= \emptyset \\
\mathrm{alph}((\kappa, \texttt{call}, m, \vec{x}, v)) &= \{(\kappa, \texttt{call}, m, \vec{y}, v) | \vec{y} \in \overline{inpar}(m)\} \\
\mathrm{alph}((\kappa, \texttt{return}, m, \vec{x}, v)) &= \{(\kappa, \texttt{return}, m, \vec{y}, v) | \vec{y} \in \overline{outpar}(m)\} \\
\mathrm{alph}((\kappa, \texttt{exception}, m, (E, \bar{x}), v)) &= \{(\kappa, \texttt{exception}, m, (E, \bar{y}), v) | \forall y_i \in \bar{y}, y_i \in \mathcal{D}\} \\
\mathrm{alph}((\kappa, \texttt{event}, E, \vec{x}, v)) &= \{(\kappa, \texttt{event}, E, \vec{y}, v) | \forall y_i \in \bar{y}, y_i \in \mathcal{D}\} \\
\mathrm{alph}(EF) &= \mathrm{alph}(E) \cup \mathrm{alph}(F) \\
\mathrm{alph}(E + F) &= \mathrm{alph}(E) \cup \mathrm{alph}(F) \\
\mathrm{alph}(E \parallel F) &= \mathrm{alph}(E) \cup \mathrm{alph}(F) \\
\mathrm{alph}(E^*) &= \mathrm{alph}(E) \\
\mathrm{alph}(x : Pred \text{ in } E) &= \mathrm{alph}(E)
\end{aligned}$$

On remarque que cet alphabet contient, quelles que soient la structure de l'expression et les contraintes associées aux variables, l'ensemble des messages qu'il est possible de construire pour chaque instance de message apparaissant dans l'expression. La principale utilité de cet alphabet apparaîtra dans la définition de l'ensemble de traces associé à un composant.

6.1.5 Ensemble de traces d'une interface

Dans le cas des interfaces, on cherche à spécifier une communication point-à-point entre deux correspondants sous la forme d'échanges d'appel-retour de méthodes. Il est donc nécessaire de contraindre la forme de la trace pour maintenir une sémantique d'appel de méthode correcte : chaque émission d'un retour ou d'une exception doit immédiatement être précédée d'un appel correspondant.

Définition 6.9 (Mots bien formées) Soit I une interface. On note $WF(I)$ l'ensemble des mots *biens formés* sur l'alphabet de I, $\mathrm{alph}(I)$:

$$WF(I) = \left(\bigcup_{\substack{m \in meth(I), \\ \vec{x} \in inpar(m), \\ \vec{y} \in outpar(m) \\ (E, \bar{y}), E \in exceptions(m)}} \begin{array}{l} (\kappa, \texttt{call}, m, \vec{x}, \texttt{receive})(\kappa, \texttt{return}, m, \vec{y}, \texttt{emit}) \\ +(\kappa, \texttt{call}, m, \vec{x}, \texttt{receive})(\kappa, \texttt{exception}, m, (E, \bar{y}), \texttt{emit}) \end{array} \right)^* .$$

Définition 6.10 (Ensemble de traces d'une interface) Soit $I = \langle meth(I), \mathcal{T}(I) \rangle$ une interface dont l'expression FIDL est E, alors :

$$\mathcal{T}(I) = Pref(\mathcal{T}(E) \cap WF(I)).$$

Cet ensemble de traces est celui des traces valides de la spécification de l'interface. Toutefois, nous n'avons émis aucune hypothèse quant au comportement des clients qui sont susceptibles de ne pas respecter le « contrat » représenté par cet ensemble de traces. Pour obtenir la spécification réelle du comportement d'une interface, il sera donc nécessaire de compléter cet ensemble de traces valides par toutes les traces non valides mais possibles, c'est-à-dire par toutes les traces contenant un appel de méthode invalide. Comme il est usuel dans les spécifications de type *rely-guarantee*, en présence d'un client ne respectant pas le protocole d'utilisation de l'interface, on obtient un comportement divergent de celle-ci qui peut désormais émettre et recevoir n'importe quel message syntaxiquement correct. Cette complétion implicite du comportement est laissée en suspens pour être précisée en fonction du contexte d'utilisation de l'ensemble de traces.

6.1.6 Ensemble de traces d'un composant

La spécification du comportement d'un composant est dépendante de la spécification des ports qu'il offre et utilise — facettes, réceptacles, sources et puits — et d'une spécification explicite complémentaire

sous la forme d'une **and**-expression : des expressions FIDL reliées par l'opérateur **and**. L'expression FIDL sert normalement à préciser les dépendances entre ports du composant ou à éliminer du non-déterminisme dans la spécification des interfaces.

Définition 6.11 (Ensemble de traces d'un composant) Soit $C = \langle Port(C), \mathcal{T}(C)\rangle$ un composant dont l'expression FIDL est E_1 **and** ... **and** E_n. Le langage induit par cette expression est défini comme :

$$L_E = \left(\mathcal{T}(E_1) \underset{\text{alph}(E_1),\text{alph}(E_2)}{\sqcap\!\sqcap} \mathcal{T}(E_2) \underset{\text{alph}(E_2),\text{alph}(E_3)}{\sqcap\!\sqcap} \cdots \underset{\text{alph}(E_{n-1}),\text{alph}(E_n)}{\sqcap\!\sqcap} \mathcal{T}(E_n)\right).$$

Soit L_F le langage de ses ports offerts

$$L_F = \bigsqcup\!\bigsqcup_{p\in\mathcal{F}(C)\cup\mathcal{S}ink(C)} \mathcal{T}(p)$$

et L_R celui de ses ports requis

$$L_R = \bigsqcup\!\bigsqcup_{p\in\mathcal{R}(C)\cup\mathcal{S}ource(C)} \mathcal{T}(p).$$

On note Σ_E et $\Sigma_{F,R}$ les alphabets de synchronisation respectivement définis par :

$$\Sigma_E = \bigcup_{1\leq i\leq n} \text{alph}(E_i)$$

et

$$\Sigma_{F,R} = \bigcup_{p\in\mathcal{R}(C)\cup\mathcal{S}ource(C)} \text{alph}(p) \cup \bigcup_{q\in\mathcal{F}(C)\cup\mathcal{S}ink(C)} \text{alph}(q).$$

L'ensemble de traces de C, $\mathcal{T}(C)$, est défini comme :

$$\mathcal{T}(C) = L_E \underset{\Sigma_E,\Sigma_{F,R}}{\sqcap\!\sqcap} (L_F \sqcup\!\sqcup L_R).$$

La distinction entre les différents alphabets utilisés dans le produit de synchronisation est importante. Elle permet de vérifier qu'un composant *respecte* la spécification de ses ports au sens où il ne peut définir de comportement qui ne soit prévu par sa spécification. Par ailleurs, elle autorise à alléger la spécification du composant en n'imposant pas de redéfinir le comportement de celui-ci explicitement pour chacun de ses ports et chacun des messages possibles de ceux-ci. Une alternative eut été de considérer que le comportement du composant fût uniquement l'ensemble de traces induit par les expressions de son invariant. Ce choix eût obligé le concepteur à repréciser dans l'expression du composant les fragments des expressions d'interfaces permettant de conserver un comportement compatible avec les spécifications des interfaces.

Compte tenu d'une part de la définition des alphabets des expressions (définition 6.8) qui contient tous les messages possibles de même structure pour chaque occurrence de message dans l'expression, et d'autre part de la définition du produit de synchronisation (section 5.1.1), le comportement d'un composant est simultanément une spécialisation du comportement de chacune de ses interfaces et l'expression des liens existant entre les différents ports offerts par le composant.

6.2 Propriétés

Nous avons défini les spécifications des interfaces en terme de *contrats* entre un client et un fournisseur de l'interface et la spécification des composants en terme d'une restriction des entrelacements possibles de messages sur l'ensemble de ses ports. Nous nous intéressons maintenant à définir les propriétés que doit vérifier un composant ou sa spécification : d'une part le respect des contrats relatifs aux différents ports offerts et requis par le composant, d'autre part l'indépendance des services offerts relativement aux autres services du composant.

6.2.1 Système

Les langages qui nous intéressent ne sont pas arbitraires mais possèdent une structure particulière : pour chaque occurrence d'un message représentant un *retour* d'appel de méthode, il doit exister un message représentant l'*appel* de méthode précédant le retour. De plus, nous nous intéressons à l'observation finie du comportement de systèmes donc les langages observés doivent être clos par préfixe. Nous appellerons ces langages des langages consistants.

Définition 6.12 (Langages consistants) Un langage $\mathcal{L} \subseteq \mathcal{E}^*$ est dit *consistant* si et seulement si

$$h_\lambda(\mathcal{L}) \subseteq \left(\coprod_{m \in \mathcal{X}} \left(\begin{array}{l} (\kappa, \texttt{call}, m, \texttt{emit})(\kappa, \texttt{return}, m, \texttt{receive}) \\ + \\ (\kappa, \texttt{call}, m, \texttt{receive})(\kappa, \texttt{return}, m, \texttt{emit}) \end{array} \right) \right)^* \tag{6.3}$$

et

$$\mathcal{L} = Pref(\mathcal{L}) \tag{6.4}$$

Pour prendre en compte les développements qui suivront cette section sur la construction de systèmes et d'assemblages, nous définirons les propriétés citées pour un « système » : un ensemble de ports et un langage sur les alphabets induits par cet ensemble de ports.

Définition 6.13 (Système) Un système est un couple $(P, \mathcal{L})$ où P est un ensemble de *ports* (n, c, T, g) tel que les couples (n, c) soient deux à deux disjoints et $\mathcal{L}$ un *langage consistant* sur l'union des alphabets des ports de P.

Pour la bonne compréhension de ce qui suit, et au risque d'anticiper légèrement sur des notions qui ne sont pas encore définies, précisons que l'ensemble de ports d'un système est *toujours* considéré comme un ensemble de ports *externes*, c'est-à-dire sans aucune connexion entre eux.

On peut remarquer que d'après la définition de l'ensemble de traces qui lui est associé et celle de sa structure, un composant *est* un système :

- l'ensemble de ses $Port(C)$ contient bien des éléments dont les identités sont deux à deux disjointes (définition 6.6) ;
- et $\mathcal{T}(C) \subseteq (L_F \sqcup\!\sqcup L_R)$ est évidemment consistant (définition 6.11).

6.2.2 Respect des contrats

Nous définissons donc formellement ce que signifie pour un système $S = (P, \mathcal{L})$ de respecter le contrat de ses ports. Le cas des sources d'événements est trivial : puisque que le langage $\mathcal{L}$ est supposé consistant, la projection de $\mathcal{L}$ sur l'alphabet d'une source d'événement est incluse nécessairement dans le langage du port par définition de la consistance.

Le cas des puits d'événements est plus intéressant. En effet, un puits d'événements est un service offert par le système à d'éventuels clients qui s'attendent à pouvoir émettre arbitrairement des messages vers ce puits d'événements.

Définition 6.14 (Respect des puits d'événements) Soit $S = (P, \mathcal{L})$ un système et $s = (n, c, T, \texttt{sink})$ un port de P. S respecte le port s, si et seulement si

$$\forall x \in \text{alph}(s), \forall u \in \mathcal{L}, \exists v \in \left(\bigcup_{\{p=(n,c,T,g) \in P | g \in \{\texttt{receptacle}, \texttt{source}\}\}} \text{alph}(p) \right)^*, uvx \in \mathcal{L}.$$

Le respect par un système du contrat d'un puits s est noté $S \sim s$.

Dans le cas des facettes et des réceptacles, la situation est bien entendu plus complexe puisqu'ici, le port impose des restrictions sur l'ordre des messages et que de plus, la spécification n'impose de respecter le contrat que si l'autre partie le respecte aussi. Nous introduisons donc une notion de *relation contractuelle* entre les langages qui nous permet de définir le respect d'un port par un composant en fonction de leurs ensembles de traces respectifs.

Définition 6.15 (Relation contractuelle) Soient L_1 et L_2 deux langages consistants tels que $\text{alph}(L_1) \cap \text{alph}(L_2) = X \neq \emptyset$. Une *relation* $\mathcal{S} \in \{(u,u) \mid u \in L_1 \cap L_2\}$ est *contractuelle* si pour tout $(u,u) \in \mathcal{S}$ alors

$$\forall v = ui \in L_1, i \in In(X), v \in L_2 \text{ et } (v,v) \in \mathcal{S}, \tag{6.5}$$

$$\forall v = uo \in L_2, o \in Out(X), v \in L_1, \text{ et } (v,v) \in \mathcal{S}, \tag{6.6}$$

avec $Out(X) = \{(\kappa, k, m, \vec{x}, \texttt{emit}) \in X\}$ et $In(X) = \{(\kappa, k, m, \vec{x}, \texttt{receive}) \in X\}$.

Définition 6.16 (Fiabilité contractuelle) Soient L_1 et L_2 deux langages consistants , L_2 est *contractuellement fiable* pour L_1, ce qui est noté $L_1 \lesssim L_2$ si et seulement si il existe une relation contractuelle $\mathcal{S} \subseteq (L_1 \cap L_2)^2$ telle $(\epsilon, \epsilon) \in \mathcal{S}$.

Cette définition est une adaptation au contexte de composants et d'interfaces contractuellement spécifiés de la relation bien connue de *bisimulation*[112]. Elle s'interprète simplement comme le fait que L_2 respecte le contrat de L_1 s'il accepte toutes les entrées spécifiées par L_1 et ne produit pas plus de sorties, et ce pour tout préfixe contractuellement correct.

Propriété 6.17 La relation $\lesssim$ entre deux langages est réflexive et transitive.

Preuve 6.18 La réflexivité est immédiate.

Soit L_1, L_2 et L_3 des langages tels que $L_1 \lesssim L_2$ et $L_2 \lesssim L_3$. Alors il existe des relations contractuelles $\mathcal{S}_1 \in (L_1 \cap L_2)^2$ et $\mathcal{S}_2 \in (L_2 \cap L_3)^2$ telles que $L_1 \lesssim L_2$ et $L_2 \lesssim L_3$.

Soit $\mathcal{S}_3 \subseteq (L_1 \cap L_3)^2$ la relation définie comme

$$\mathcal{S}_3 = \{(u,u) \in (L_1 \cap L_3)^2 \mid (u,u) \in \mathcal{S}_1, (u,u) \in \mathcal{S}_2\}.$$

Soit u tel que $(u,u) \in \mathcal{S}_3$, alors par définition $(u,u) \in \mathcal{S}_1$ et $(u,u) \in \mathcal{S}_2$. Pour tout $u' = ui, i \in In(\text{alph}(L_1))$ tel que $ui \in L_1$, $u' \in L_2$ car $L_1 \lesssim L_2$ et $(u',u') \in \mathcal{S}_2$. Donc par définition de S_3, $(u',u') \in S_3$.

Pour tout $u' = uo \in L_3, o \in Out(\text{alph}(L_3))$, par la relation $L_2 \lesssim L_3$, $u' \in L_2$, $(u',u') \in \mathcal{S}_2$; et par conséquent si $o \in Out(\text{alph}(L_1))$, $u' \in L_1$.

Donc $\mathcal{S}_3$ est bien une *relation contractuelle* contenant (ϵ, ϵ) et l'on a

$$L_1 \lesssim L_2 \wedge L_2 \lesssim L_3 \implies L_1 \lesssim L_3.$$

■

Nous pouvons désormais définir le respect d'un port p de type `facet` ou `receptacle` par un système.

Définition 6.19 (Respect des Réceptacles) Soit $S = (P, \mathcal{L})$ un système et $p = (r, c, I, \texttt{receptacle})P$ un réceptacle de P. S respecte le réceptacle p, ce qui est noté $S \sim p$, si et seulement si

$$\mathcal{T}(p) \lesssim \Pi_{\text{alph}(p)}(\mathcal{L}).$$

Définition 6.20 (Respect des Facettes) Soit $S = (P, \mathcal{L})$ un système et $p = (f, c, I, \texttt{facet})$ une facette de P. S respecte la facette p, ce qui est noté $S \sim p$, si et seulement si

$$\mathcal{T}(p) \lesssim \Pi_{\text{alph}(p)}(\mathcal{L}).$$

Rappelons que dans le cas des réceptacles, le sens des messages est inversé (voir définition 6.5), par conséquent la relation contractuelle *ne doit pas* être inversée comme on pourrait s'y attendre. Dans tous les cas, le langage $\mathcal{L}$ doit effectuer moins de sorties que prévues par la spécification du port ce qui pour les réceptacles signifie émettre moins d'appels, et inversement pour les entrées.

6.2.3 Indépendance des facettes

Dans le cadre de systèmes ouverts et potentiellement répartis, les clients utilisant différentes facettes d'un même composant n'ont *a priori* aucune raison de se connaître. Un composant ou sa spécification ne doivent donc pas supposer une relation de dépendance causale entre les différents clients et donc entre les facettes qu'ils offrent. Cela signifie que la fourniture d'un service sur une facette doit être *indépendante* du comportement d'un autre client sur une autre facette. Cette propriété, que nous appelons *indépendance des facettes*, est plus formellement définie comme suit.

Définition 6.21 (Indépendance de facette) Soit $S = (P, \mathcal{L})$ un système et $f = (p, c, I, \texttt{facet}) \in P$ une facette du système. S respecte l'indépendance de sa facette f, ce qui est noté $S \perp f$ si et seulement si

$$\begin{array}{c} \forall u \in h_\lambda(\mathcal{L}), \forall x \in h_\lambda(Out(\mathcal{E})) \text{ tel que } \Pi_{h_\lambda(\mathrm{alph}(f))}(u)x \in h_\lambda(\mathcal{T}(f)), \\ \exists v \text{ tel que } uvx \in h_\lambda(\mathcal{L}) \text{ et } \forall p = (\varphi, c, T, g) \in P \text{ avec } g \in \{\texttt{facet}, \texttt{sink}\}, \\ \Pi_{h_\lambda(\mathrm{alph}(p))}(v) = \epsilon, \end{array} \tag{6.7}$$

et

$$\begin{array}{c} \forall u \in \mathcal{L}, \forall x \in In(\mathcal{E}) \text{ tel que } \Pi_{\mathrm{alph}(f)}(u)x \in h_\lambda(\mathcal{T}(f)), \\ \exists v \text{ tel que } uvx \in \mathcal{L} \text{ et } \forall p = (\varphi, c, T, g) \in P \text{ avec } g \in \{\texttt{facet}, \texttt{sink}\}, \\ \Pi_{\mathrm{alph}(p)}(v) = \epsilon. \end{array} \tag{6.8}$$

Cette propriété s'énonce plus simplement comme le fait que tout mot abstrait du langage d'une facette f, toute séquence d'enveloppes de messages, doit appartenir au langage du système sans qu'interfèrent dans le mot des lettres appartenant au langage d'une autre facette ou d'un puits du système. Autrement dit, un système doit toujours pouvoir « répondre » à un client indépendamment de ce que font les autres clients du système.

Deux remarques importantes concernant cette propriété :

1. l'indépendance exprime l'absence de blocage général d'une facette par une autre, mais n'interdit pas de faire dépendre le *contenu* des messages sur la facette de messages reçus sur une autre facette ;
2. l'indépendance concerne uniquement les langages des facettes et puits et ne dit rien sur les dépendances possibles entre facettes et autre ports du composant : réceptacles et sources.

Cette propriété peut paraître très restrictive et elle est souvent difficile à spécifier correctement. Elle permet toutefois de concevoir des systèmes réellement ouverts et de s'assurer par construction de l'absence de ce type de blocages dans un système. De plus, dans une approche de type *rely-guarantee*, un fournisseur ne peut faire aucune hypothèse sur la structure ou l'identité de ses clients, tant que ceux-ci respectent le contrat de service.

6.2.4 Fiabilité d'un système

Nous pouvons maintenant définir une notion de *fiabilité* pour un système S. Un système est *fiable* s'il respecte les contrats de l'ensemble de ses ports et l'indépendance de ses facettes.

Définition 6.22 (Système fiable) Un système $S = \langle P, \mathcal{L} \rangle$ est *fiable* si et seulement si :

$$\begin{array}{c} \forall (p, c, T, g) \in P, S \sim p \\ \text{et} \\ \forall (f, c, I, \texttt{facet}) \in P, S \perp f. \end{array}$$

6.3 Compositionnalité

Dans les modèles formels de composants — ou assimilés — que nous avons étudiés au chapitre 2, la composition de composants, lorsque leur comportement est exprimé sous la forme de systèmes de transitions étiquetés, est réalisée par identification des messages d'entrée et de sortie reçus et émis sur chaque port, résultat d'un modèle de communication synchrone. Le cadre que nous avons choisi implique que

le modèle de communication des composants FIDL soit *asynchrone* : émission et réception sont deux messages différents distingués dans la structure des événements par le *point de vue*. La sémantique des interactions entre composants est donc dépendante de la sémantique des connexions entre les composants qui assure la *synchronisation*, au sens des langages, entre les différents points de vue.

Dans cette section, nous construisons donc les différents langages résultant de la composition de composants, en partant des *connexions* entre ports compatibles, puis en examinant le comportement d'un système quelconque résultant d'un ensemble de composants et de connexions, pour enfin définir un composite qui, normalement, devrait avoir les mêmes propriétés qu'un composant pour ce qui est de construire des systèmes.

6.3.1 Connexions

Une connexion est la réalisation d'un lien entre deux ports, l'un fournissant un service, l'autre le requérant, de sorte que les messages émis sur un port soient reçus sur l'autre port. Dans un premier temps, nous considérons uniquement le cas de la connexion de deux ports du même type. La section 6.4 introduira une notion de sous-typage comportemental nous permettant de traiter la connexion de deux ports de types différents.

Définition 6.23 (Connexion) Une connexion χ entre deux ports $p = (n, c, T, g)$ et $p' = (n', c', T, g')$ appartenant respectivement à deux instances distinctes de composants c et $c', c \neq c'$, et tels que $(g, g') \in \{(\texttt{receptacle}, \texttt{facet}), (\texttt{source}, \texttt{sink})\}$ est notée $\chi = (c, n, c', n')$ et induit un langage L_χ sur un alphabet $\mathrm{alph}(\chi)$ respectivement définis comme suit :

- $\mathrm{alph}(\chi) = h_\chi(\mathrm{alph}(p)) \cup h_\chi(\mathrm{alph}(p'))$;
- $L_\chi = \left(\bigcup_{(\kappa,k,m,\vec{x},v)\in \mathrm{alph}(p)\cup\mathrm{alph}(p')} (h_\chi((\kappa, k, m, \vec{x}, \texttt{emit}))h_\chi((\kappa, k, m, \vec{x}, \texttt{receive}))) \right)^*$.

avec h_χ le *morphisme de connexion* induit par $\chi = (c, n, c', n')$ défini comme :

$$\begin{array}{rlcl} h_\chi : & \mathcal{E} & \longrightarrow & h_\chi(\mathcal{E}) \\ & (c, n, \gamma_2, \varrho_2, k, m, \vec{x}, v) & \longmapsto & (c, n, c', n', k, m, \vec{x}, v) \\ & (\gamma_1, \varrho_1, c', n', k, m, \vec{x}, v) & \longmapsto & (c, n, c', n', k, m, \vec{x}, v) \\ & x & \longmapsto & x \text{ sinon.} \end{array}$$

Le langage d'une connexion est un ordonnancement des messages susceptibles d'être transportés par la connexion de telle sorte que tout message dénotant une réception soit précédé dans le langage de la connexion du message dénotant l'émission du même événement. De plus, ce langage concerne des messages où toutes les variables représentant les identités des ports et composants parties prenantes de la connexion sont instanciées avec les identités réelles de ces entités, ce qui est réalisé par le morphisme de connexion h_χ.

Soit $X = \{\chi_1, \chi_2, \ldots, \chi_n\}$ un ensemble de connexions. Nous noterons $elem(X)$ l'ensemble des ports connectés de X :

$$elem(X) = \{(c, n) \mid \exists (c, n, c', n') \in X \vee \exists (c', n', c, n) \in X\}.$$

Définition 6.24 (Ensemble de connexions) Soit X un ensemble de connexions. Si pour tout couple de connexions $\chi_i = (c, n, d, m), \chi_j = (c', n', d', m') \in X$, on a $c \neq c'$ ou $n \neq n'$, et $d \neq d'$ ou $m \neq m'$, alors $h_{\chi_i} \circ h_{\chi_j} = h_{\chi_j} \circ h_{\chi_i}$ et l'on définit le morphisme h_X comme la composition de tous les morphismes h_{χ_i}.

De même, $\mathrm{alph}(X)$ l'alphabet de X est défini comme l'union des alphabets des connexions de X.

Rappelons que nous ne traitons ici que des facettes dont la modalité est **unique** et que par conséquent un port ne peut être utilisé que par une seule connexion. Cela signifie donc que les inégalités nécessaires dans la définition 6.24 sont toujours satisfaites. Dans le reste de ce chapitre, tous les ensembles de connexions considérés possèdent cette propriété.

6.3.2 Assemblages

Nous introduisons maintenant la notion fondamentale d'*assemblage* de composants et le langage associé qui résulte d'un ensemble de connexions sur un ensemble d'instances de composants.

Définition 6.25 (Assemblage) Un assemblage $\mathcal{A} = \langle B, X \rangle$ est construit à partir d'un ensemble d'instances de composants $B = \{c_1, \ldots, c_n\}$ et d'un ensemble de connexions X sur B. Le langage de l'assemblage $L_{\mathcal{A}}$ et son alphabet sont définis par :

$$\text{alph}(\mathcal{A}) = \bigcup_{c \in B} h_X(\text{alph}(c)), \tag{6.9}$$

$$L_{\mathcal{A}} = \left(\coprod_{c \in B} h_X(L_c) \right) \sqcap_{\text{alph}(\mathcal{A}),\text{alph}(X)} \left(\coprod_{\chi \in X} L_\chi \right). \tag{6.10}$$

On notera $Port(\mathcal{A})$ l'ensemble des ports de B qui ne sont pas connectés dans $\mathcal{A}$:

$$Port(\mathcal{A}) = \{(n, c, T, g) \mid \nexists (c, n) \in elem(X)\}.$$

Le langage d'un assemblage résulte du produit de synchronisation des différents langages des composants avec les langages des connexions. Cette synchronisation induit donc un ordonnancement des messages transitant par les connexions de l'assemblage.

Remarquons que les alphabets des composants étant disjoints deux à deux, les produits de mélange dans l'équation (6.10) peuvent être remplacés par des produits de synchronisation (voir section 5.1.1). On en déduit aisément qu'un assemblage considéré uniquement au travers de ses ports non connectés est un système.

Proposition 6.26 Soit $\mathcal{A} = \langle B, X \rangle$ un assemblage. Alors $\langle Port(A), L_{\mathcal{A}} \rangle$ est un système.

Preuve 6.27 Un composant est un système et les morphismes de connexion induits par X maintiennent la propriété de consistance. Enfin, les propriétés du produit de synchronisation induisent immédiatement l'inclusion (6.3). ■

6.3.3 Propriétés d'un assemblage

La question principale qui se pose à nous est de vérifier qu'un assemblage de composants fiables respecte les ports qui n'apparaissent pas dans l'ensemble de connexions X. Nous produisons tout d'abord un exemple qui montre que l'on ne peut pas assembler les composants de manière arbitraire.

Soient les composants suivants :

$$c = \langle \{(f, c, I, \texttt{facet}), (g, c, J, \texttt{facet}), (r, c, K, \texttt{receptacle})\}, \mathcal{T}(c) \rangle$$
$$d = \langle \{(h, d, K, \texttt{facet}), (q, d, I, \texttt{receptacle}), (p, d, L, \texttt{receptacle})\}, \mathcal{T}(d) \rangle,$$

avec $meth(I) = \{mi()\}, meth(J) = \{mj()\}, meth(K) = \{mk()\}$ et $meth(L) = \{ml()\}$. Par ailleurs, on a :

$$\mathcal{T}(c) = \begin{array}{ll} (& (f \rightarrow mi() r \rightarrow mk() r \leftarrow mk() f \leftarrow mi())+ \\ & (g \rightarrow mj() r \rightarrow mk() r \leftarrow mk() g \leftarrow mj()))^* \end{array}$$

$$\mathcal{T}(d) = (h \rightarrow mk() q \rightarrow mi() q \leftarrow mi() p \rightarrow ml() p \leftarrow ml() h \leftarrow mk())^*.$$

Soit l'assemblage $\mathcal{A}$ défini comme suit :

$$\mathcal{A} = \langle \{c, d\}, X = \{(d, q, c, f), (c, r, d, h)\} \rangle.$$

et représenté dans la figure 6.2.

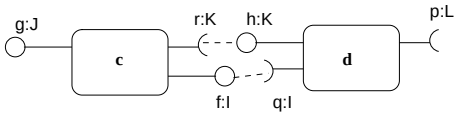


FIG. 6.2 – Contre-exemple.

On vérifie immédiatement que c et d, vus comme des systèmes, sont bien fiables pour leurs facettes et réceptacles respectifs. Rappelons qu'en l'absence de spécification explicite, le comportement d'une interface est toujours défini comme l'ensemble de toutes les séquences d'appels-retours des méthodes déclarées dans l'interface. Le langage de $L_{\mathcal{A}}$, selon la définition 6.25, est :

$$L_{\mathcal{A}} = h_X(\mathcal{T}(c)) \underset{h_X(\mathrm{alph}(c)),\mathrm{alph}(X)}{\sqcap\!\sqcap} L_X \underset{\mathrm{alph}(X),h_X(\mathrm{alph}(d))}{\sqcap\!\sqcap} h_X(\mathcal{T}(d)),$$

que l'on peut développer en

$$((f \rightarrow mi()r \rightarrow mk()r \leftarrow mk()f \leftarrow mi()) + (g \rightarrow mj()r \rightarrow mk()r \leftarrow mk()g \leftarrow mj()))^*$$
$$\underset{\sqcap\!\sqcap h_X(\mathrm{alph}(c)),\mathrm{alph}(X)}{}$$
$$((q \rightarrow mi()f \rightarrow mi())^* \sqcup\!\sqcup (f \leftarrow mi()q \leftarrow mi())^* \sqcup\!\sqcup (r \rightarrow mk()h \rightarrow mk())^* \sqcup\!\sqcup (h \leftarrow mk()r \leftarrow mk())^*)$$
$$\underset{\sqcap\!\sqcap \mathrm{alph}(X),h_X(\mathrm{alph}(d))}{}$$
$$(h \rightarrow mk()q \rightarrow mi()q \leftarrow mi()p \rightarrow ml()p \leftarrow ml()h \leftarrow mk())^*.$$

Le langage résultant est :

$$(g \rightarrow mj()r \rightarrow mk()h \rightarrow mk()q \rightarrow mi()),$$

rappelons que les ensembles de traces de composants sont des langages clos par préfixes et que par conséquent le produit de synchronisation obtenu contient des préfixes corrects se terminant en blocage.

Clairement

$$h_X(L_f) \not\lesssim \Pi_{h_X(\mathrm{alph}(f))}(L_{\mathcal{A}}) \text{ et} \qquad h_X(L_p) \not\lesssim \Pi_{h_X(\mathrm{alph}(p))}(L_{\mathcal{A}}),$$

donc $\mathcal{A}$ ne respecte pas ses facettes et réceptacles au sens de 6.19 et 6.20 et n'est donc pas fiable.

Le cycle de dépendances entre les facettes et réceptacles connectés de c et d ne pose pas en soi de problèmes : ce qui pose problème ce sont les dépendances existant dans les langages de c et d entre leurs facettes et réceptacles connectés respectifs. Ce problème peut-être levé de deux manières :

- soit en s'assurant qu'il n'existe pas de mots dans le langage des composants assemblés induisant un ou plusieurs cycles de dépendances ;
- soit en restreignant les opérations d'assemblage correctes à celles qui sont garanties ne pas introduire de cycles.

Nous allons donc montrer que l'absence de cycle est une condition suffisante pour obtenir un assemblage fiable.

Assemblage de deux composants

Nous considérons un assemblage formé de deux composants c et d tel que les connexions de l'assemblage relient uniquement des ports requis de c à des ports fournis par d. Nous montrons alors que si les deux composants, vus comme des systèmes, sont fiables pour leurs ports respectifs, alors l'assemblage résultant est fiable pour les ports non connectés.

Théorème 6.28 (Composition deux à deux) *Soit $c = \langle Port(c), \mathcal{T}(c)\rangle$ et $d = \langle Port(d), \mathcal{T}(d)\rangle$ deux instances de composants et X un ensemble de connexions ne concernant que des ports requis de c avec des ports fournis de d.*

Alors, si c et d sont des composants fiables, *l'assemblage $\mathcal{A} = \langle\{c, d\}, X\rangle$ est un système fiable — définition 6.22 — pour l'ensemble de ports $Port(\mathcal{A})$:*

1. *$\mathcal{A}$ respecte l'indépendance des facettes de $Port(\mathcal{A})$ (définition 6.21) ;*

2. *$\mathcal{A}$ respecte les contrats des puits et sources de $Port(\mathcal{A})$ (definition 6.14) ;*
3. *$\mathcal{A}$ respecte les contrats des facettes et réceptacles de $Port(\mathcal{A})$ (définitions 6.19 et 6.20).*

La figure 6.3 est une représentation schématisée de l'assemblage $\mathcal{A}$ pour deux composants : les ports des deux composants concernés qui ne sont pas dans X sont représentés en traits pleins dans la partie droite et l'on verra ultérieurement que l'on obtient ainsi un nouveau composant ou composite.

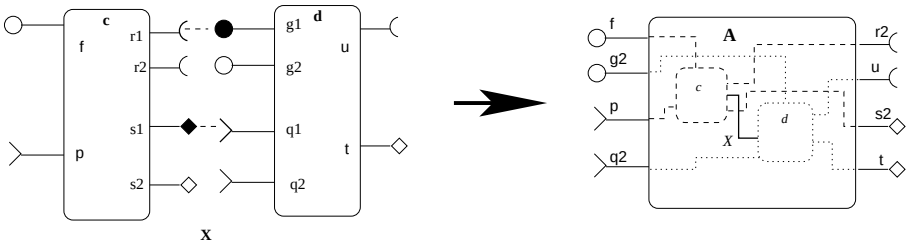


FIG. 6.3 – Composition de c et d.

Nous montrons tout d'abord par induction sur la longueur des mots de $L_{\mathcal{A}}$ que la propriété d'indépendance des facettes est préservée. Si c et d respectent l'indépendance de leurs facettes, alors $\mathcal{A}$ respecte l'indépendance des facettes de $Port(\mathcal{A})$.

Preuve 6.29 (Préservation de l'indépendance des facettes) Supposons que pour tous les mots de $L_{\mathcal{A}}$ de longueur inférieure ou égale à n la propriété soit vraie et soit w un tel mot. Par construction de $L_{\mathcal{A}}$, nous savons qu'il existe $u_c \in h_X(\mathcal{T}(c))$, $u_d \in h_X(\mathcal{T}(d))$ et $\chi \in L_X$ tels que $w \in (u_c \sqcup\!\sqcup u_d \sqcap \chi)$.

Soit f une facette de $Port(\mathcal{A})$ et $x \in In(\mathrm{alph}(f))$ une lettre telle que $\Pi_{\mathrm{alph}(f)}(w)x \in \mathcal{T}(f)$. Remarquons que $h_X(\mathrm{alph}(f)) = \mathrm{alph}(f)$ car $f \in Port(\mathcal{A})$.

Nous distinguerons deux cas :

1. Si $f = (n, d, I, \texttt{facet})$ est une facette de d. Par hypothèse du respect de l'indépendance des facettes de d, pour tout mot u de $\mathcal{T}(d)$ et toute lettre $x \in In(\mathrm{alph}(d))$ tels que $\Pi_{\mathrm{alph}(f)}(u)x$, il existe un mot $v_d \in \mathrm{alph}(d)^*$ tel que pour tout port $p = (\phi, d, T, g)$ de $Port(d)$ avec $g \in \{\texttt{facet}, \texttt{sink}\}$, $\Pi_{\mathrm{alph}(p)}(v_d) = \epsilon$. En particulier, cette propriété est vraie pour u_d, donc il existe un mot $u_d v_d x$ de $\mathcal{T}(d)$. Comme f n'est pas connectée dans $\mathcal{A}$ et qu'aucun des réceptacles ou sources de d ne l'est, on a $h_X(v_d) = v_d$ et par conséquent

$$wv_dx \in (u_c \sqcup\!\sqcup u_d v_d x \sqcap \chi) \subseteq L_{\mathcal{A}}.$$

2. Si $f = (n, c, I, \texttt{facet})$ est une facette de c. Par hypothèse, c respecte l'indépendance de ses facettes donc il existe v_c tel que $u_c v_c x \in \mathcal{T}(c)$ et $\Pi_{\bigcup_{p \in \mathcal{F}(c) \cup \mathcal{S}ink(c)} \mathrm{alph}(p)}(v_c) = \epsilon$: v_c est constitué uniquement de lettres appartenant à l'alphabet des réceptacles et sources de c (cependant ces réceptacles et sources peuvent être connectés à des ports de d).

   Nous posons $v_c = a_1 a_2 \ldots a_n$ avec pour tout $1 \leq i \leq n$, $a_i \in h_X(\mathrm{alph}(c))$ et nous cherchons à montrer par induction qu'il existe un mot $v_d = v_1 w_1 v_2 w_2 \ldots v_n w_n$, avec pour tout $1 \leq i \leq n$, $v_i, w_i \in h_X(\mathrm{alph}(d)^*)$ et $u_d v_d \in h_X(\mathcal{T}(d))$, tel que pour tout $p \in (\mathcal{F}(d) \cup \mathcal{S}ink(d)) \cap Port(A)$, $\Pi_{\mathrm{alph}(p)}(v_d) = \epsilon$ et un mot $\chi' \in \mathrm{alph}(X)^*$ tels que

$$wv_1a_1w_1v_2a_2w_2 \ldots v_n a_n w_n x \in (u_c v_c x \sqcup\!\sqcup u_d v_d \sqcap \chi') \in L_{\mathcal{A}}.$$

   La propriété est évidemment vraie pour $v_c = z = \chi' = \epsilon$.

   Supposons que la propriété soit vraie pour un certain j, $1 \leq j < n$. Il existe donc $v_1 w_1 \ldots v_j w_j$ et χ'_j tels que

$$wv_1a_1w_1v_2a_2w_2 \ldots a_{j-1}w_{j-1}v_j a_j w_j \in (u_c a_1 \ldots a_j \sqcup\!\sqcup u_d v_1 w_1 \ldots v_j w_j \sqcap \chi'_j) \in L_{\mathcal{A}}.$$

   Nous examinons les différents cas pour a_{j+1} :

(a) si $a_{j+1} \in h_X(\text{alph}(r))$ avec $r = (n, c, T, g) \in Port(\mathcal{A}), g \in \{\texttt{receptacle}, \texttt{source}\}$ un port requis non connecté dans $\mathcal{A}$, alors $a_{j+1} \notin h_X(\text{alph}(d))$ et $a_{j+1} \notin \text{alph}(X)$, on a donc immédiatement

$$\begin{array}{rl} & wv_1a_1w_1v_2a_2w_2 \ldots a_{j-1}w_{j-1}v_ja_jw_jv_{j+1}a_{j+1}w_{j+1} \\ \in & (u_ca_1 \ldots a_ja_{j+1} \sqcup\!\sqcup u_dv_1w_1 \ldots v_jw_jv_{j+1}w_{j+1} \sqcap\!\sqcap \chi'_j) \\ \subseteq & L_{\mathcal{A}}. \end{array}$$

avec $v_{j+1} = w_{j+1} = \varepsilon$ et $\chi'_{j+1} = \chi'_j$ et la propriété est vraie ;

(b) si $a_{j+1} \in h_X(\text{alph}(p))$ avec $p = (n, c, T, \texttt{receptacle})$, un réceptacle connecté de c tel que $(c, n, d, n') \in X$ et $a_{j+1} = (c, n, d, n', t, m, \vec{x}, \texttt{receive})$. Comme par hypothèse, d respecte ses puits et respecte l'indépendance de ses facettes, il existe un mot w tel que $u_dv_1w_1 \ldots v_jw_jwb \in h_X(\mathcal{T}(d))$ avec $b = \bar{h}(a_{j+1})$ et $\Pi_{\bigcup_{p \in \mathcal{F}(d) \cup \mathcal{S}ink(d)} \text{alph}(p)}(w) = \varepsilon$. Donc, $w \notin \text{alph}(X)^*$ et on a immédiatement

$$\begin{array}{rl} & wv_1a_1w_1v_2a_2w_2 \ldots a_{j-1}w_{j-1}v_ja_jw_jv_{j+1}a_{j+1}w_{j+1} \\ \in & (u_ca_1 \ldots a_ja_{j+1} \sqcup\!\sqcup u_dv_1w_1 \ldots v_jw_jv_{j+1}w_{j+1} \sqcap\!\sqcap \chi'_j) \\ \subseteq & L_{\mathcal{A}}. \end{array}$$

avec $v_{j+1} = wb$, $w_{j+1} = \epsilon$ et $\chi'_{j+1} = \chi'_j a_{j+1} b$ et la propriété est vraie ;

(c) si $a_{j+1} \in h_X(\text{alph}(p))$ avec $p = (n, c, T, g)$, $g \in \{\texttt{source}, \texttt{receptacle}\}$, un port requis connecté de c tel que $(c, n, d, n') \in X$ et $a_{j+1} = (c, n, d, n', t, m, \vec{x}, \texttt{emit})$. Comme par hypothèse, d respecte ses puits, l'indépendance de ses facettes et les contrats de ses facettes, il existe des mots w' et v' tels que $u_dv_1w_1 \ldots w_jv'bw' \in h_X(\mathcal{T}(d))$ avec $b = \bar{h}(a_{j+1})$ et $\Pi_{\bigcup_{p \in \mathcal{F}(d) \cup \mathcal{S}ink(d)} \text{alph}(p)}(v'w') = \varepsilon$. Donc, $v', w' \notin \text{alph}(X)^*$ et on a immédiatement

$$\begin{array}{rl} & wv_1a_1w_1v_2a_2w_2 \ldots a_{j-1}w_{j-1}v_ja_jw_jv_{j+1}a_{j+1}w_{j+1} \\ \in & (u_ca_1 \ldots a_ja_{j+1} \sqcup\!\sqcup u_dv_1w_1 \ldots v_jw_jv_{j+1}w_{j+1} \sqcap\!\sqcap \chi'_j) \\ \subseteq & L_{\mathcal{A}}. \end{array}$$

avec $v_{j+1} = v'$, $w_{j+1} = bw'$ et $\chi'_{j+1} = \chi'_j a_{j+1} b$ et la propriété est vraie.

Comme le mot vide vérifie trivialement la propriété, on vérifie donc par induction que $L_{\mathcal{A}}$ préserve l'indépendance de ses facettes non connectées.

Nous avons montré la propriété pour les lettres appartenant à l'alphabet d'entrée (équation (6.8)). Pour ce qui concerne les lettres de l'alphabet de sortie (équation (6.7)), la preuve est similaire en restreignant l'alphabet de x aux lettres de l'alphabet abstrait du port considéré. ■

Nous avons donc montré que l'opération d'assemblage de deux composants préservait la propriété d'indépendance des facettes et nous supposerons donc désormais que c et d respectent l'indépendance de leurs facettes respectives, $\mathcal{A}$ fait de même. Nous allons maintenant montrer que le respect des puits et sources d'événements est lui aussi préservé dans l'assemblage.

Preuve 6.30 (Respect des puits et sources d'événements) La propriété est évidemment vérifiée dans le cas des sources d'événements puisque par construction du langage de l'assemblage, aucun nouvel événement n'appartenant pas déjà aux langages des composants c et d n'est créé donc si ces derniers ne produisent que des événements corrects pour l'alphabet d'une source, ces événements resteront corrects dans $\mathcal{A}$.

Dans le cas des puits, nous cherchons à montrer que si $s = (n, \gamma, T, \texttt{sink})$, pour $\gamma = c$ ou $\gamma = d$, est un puits d'événements de $Port(\mathcal{A})$, alors pour tout $x \in \text{alph}(s)$, pour tout $u \in L_{\mathcal{A}}$, $\exists v \in \left(\bigcup_{\{p=(n,\gamma,T,g) \in Port(\mathcal{A}) | g \in \{\texttt{receptacle},\texttt{source}\}\}} \text{alph}(p)\right)^*, uvx \in L_{\mathcal{A}}$. Cette propriété étant fortement similaire à l'indépendance des facettes, nous utiliserons la même technique pour la démontrer.

Supposons que pour tous les mots de $L_{\mathcal{A}}$ de longueur inférieure ou égale à n la propriété soit vraie et soit w un tel mot. Par construction de $L_{\mathcal{A}}$, nous savons qu'il existe $u_c \in h_X(\mathcal{T}(c))$, $u_d \in h_X(\mathcal{T}(d))$ et

$\chi \in L_X$ tels que $w \in (u_c \sqcup\!\sqcup u_d \sqcap \chi)$ ce qui par définition du produit de synchronisation et du langage $L_{\mathcal{A}}$ nous permet d'écrire w sous la forme :

$$w = u_c^1 u_d^1 u_c^2 u_d^2 \dots u_c^m u_d^m$$

avec $u_c = u_c^1 u_c^2 \dots u_c^m$ et $u_d = u_d^1 u_d^2 \dots u_d^m$.

Soit s un puits de $Port(\mathcal{A})$ et $x \in \mathrm{alph}(s)$. Remarquons que $h_X(\mathrm{alph}(s)) = \mathrm{alph}(s)$ car $s \in Port(\mathcal{A})$.

1. si $s = (n, d, T, \texttt{sink})$ est un puits de d, alors par hypothèse de respect par d de ses puits, il existe $u_d^{m+1} \in (\bigcup_{\{p=(n,d,T,g) \in Port(d) | g \in \{\texttt{receptacle},\texttt{source}\}\}} \mathrm{alph}(p))^*$ tel que $u_d u_d^{m+1} x \in \mathcal{T}(d)$. Aucun réceptacle ni source de d n'étant connecté dans $\mathcal{A}$, on a donc

$$w u_d^{m+1} x = u_c^1 u_d^1 u_c^2 u_d^2 \dots u_c^m u_d^m u_d^{m+1} x \in (u_c \sqcup\!\sqcup u_d^1 \dots u_d^{m+1} x \sqcap \chi) \subseteq L_{\mathcal{A}}.$$

2. si $s = (n, c, T, \texttt{sink})$ est un puits de c, alors par hypothèse de respect par c de ses puits, il existe $u_c^{m+1} \in (\bigcup_{\{p=(n,c,T,g) \in Port(c) | g \in \{\texttt{receptacle},\texttt{source}\}\}} \mathrm{alph}(p))^*$ tel que $u_c u_c^{m+1} x \in \mathcal{T}(c)$.

   Nous posons $u_c^{m+1} = a^1 a^2 \dots a^j$ et nous montrons par induction qu'il existe un mot $v_d = v^1 w^1 v^2 w^2 \dots v^j w^j \in h_X(\mathrm{alph}(d))^*$ tel que $v_d \in (\bigcup_{\{p=(n,d,T,g) \in Port(d) | g \in \{\texttt{receptacle},\texttt{source}\}\}} \mathrm{alph}(p))^*$ et que

$$w v^1 a^1 w^1 \dots v^j a^j w^j x \in L_{\mathcal{A}}.$$

   La propriété est évidemment vraie pour $v_c = v_d = \epsilon$.

   Supposons que la propriété soit vraie pour un certain i, $1 \leq i < j$, on a donc

$$w v^1 a^1 w^1 \dots v^j a^i w^i \in L_{\mathcal{A}},$$

   et nous examinons les différents cas pour a^{i+1} :

   (a) si $a^{i+1} \in h_X(\mathrm{alph}(r))$ avec $r = (n, c, T, g) \in Port(\mathcal{A}), g \in \{\texttt{receptacle}, \texttt{source}\}$ un réceptacle non connecté dans $\mathcal{A}$, on a $v^{i+1} = w^{i+1} = \epsilon$ qui vérifient

$$w v^1 a^1 w^1 \dots v^i a^i w^i v^{i+1} a^{i+1} w^{i+1} \in L_{\mathcal{A}},$$

   (b) si $a_{j+1} \in h_X(\mathrm{alph}(p))$ avec $p = (n, c, T, \texttt{receptacle})$, un réceptacle connecté de c tel que $(c, n, d, n') \in X$ et $a_{j+1} = (c, n, d, n', t, m, \vec{x}, \texttt{receive})$. Comme par hypothèse, d respecte ses puits et respecte l'indépendance de ses facettes, il existe un mot w tel que $u_d v_1 w_1 \dots v_j w_j w b \in h_X(\mathcal{T}(d))$ avec $b = \bar{h}(a_{j+1})$ et $\Pi_{\bigcup_{p \in \mathcal{F}(d) \cup Sink(d)} \mathrm{alph}(p)}(w) = \varepsilon$. Donc, $w \notin \mathrm{alph}(X)^*$ et on a immédiatement

$$\begin{array}{rl} & w v_1 a_1 w_1 v_2 a_2 w_2 \dots a_{j-1} w_{j-1} v_j a_j w_j v_{j+1} a_{j+1} w_{j+1} \\ \in & (u_c a_1 \dots a_j a_{j+1} \sqcup\!\sqcup u_d v_1 w_1 \dots v_j w_j v_{j+1} w_{j+1} \sqcap \chi'_j) \\ \subseteq & L_{\mathcal{A}}. \end{array}$$

   avec $v_{j+1} = wb$, $w_{j+1} = \epsilon$ et $\chi'_{j+1} = \chi'_j a_{j+1} b$ et la propriété est vraie ;

   (c) si $a_{j+1} \in h_X(\mathrm{alph}(p))$ avec $p = (n, c, T, g)$, $g \in \{\texttt{source}, \texttt{receptacle}\}$, un port requis connecté de c tel que $(c, n, d, n') \in X$ et $a_{j+1} = (c, n, d, n', t, m, \vec{x}, \texttt{emit})$. Comme par hypothèse, d respecte ses puits, l'indépendance de ses facettes et les contrats de ses facettes, il existe des mots w' et v' tels que $u_d v_1 w_1 \dots w_j v' b w' \in h_X(\mathcal{T}(d))$ avec $b = \bar{h}(a_{j+1})$ et $\Pi_{\bigcup_{p \in \mathcal{F}(d) \cup Sink(d)} \mathrm{alph}(p)}(v' w') = \varepsilon$. Donc, $v', w' \notin \mathrm{alph}(X)^*$ et on a immédiatement

$$\begin{array}{rl} & w v_1 a_1 w_1 v_2 a_2 w_2 \dots a_{j-1} w_{j-1} v_j a_j w_j v_{j+1} a_{j+1} w_{j+1} \\ \in & (u_c a_1 \dots a_j a_{j+1} \sqcup\!\sqcup u_d v_1 w_1 \dots v_j w_j v_{j+1} w_{j+1} \sqcap \chi'_j) \\ \subseteq & L_{\mathcal{A}}. \end{array}$$

   avec $v_{j+1} = v'$, $w_{j+1} = bw'$ et $\chi'_{j+1} = \chi'_j a_{j+1} b$ et la propriété est vraie.

ce qui termine l'induction.
La propriété étant vraie pour le mot vide, $\mathcal{A}$ respecte le contrat de ses puits non connectés. ■

Nous augmentons notre stock d'hypothèses en considérant désormais que $\mathcal{A}$ respecte l'indépendance des facettes de $Port(\mathcal{A})$ et qu'il est fiable pour ses sources et ses puits.

Enfin, nous montrons que la propriété de respect contractuel des facettes et des réceptacles est préservée par l'assemblage.

Preuve 6.31 (Respect des facettes et réceptacles) Soit $p = (n, \gamma, T, g)$ un port de c ou d appartenant à $Port(\mathcal{A})$. Par hypothèse $\gamma \sim p$, donc $\mathcal{T}(p) \lesssim \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma))$ et donc par définition de la fiabilité, il existe une relation $\mathcal{S}_p \in \{(u,u) \mid u \in \mathcal{T}(p) \cap \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma))\}$ telle que $(u,u) \in \mathcal{S}$ implique :

$$\forall v = ui \in \mathcal{T}(p), i \in In(\mathrm{alph}(p)), v \in \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma)) \text{ et } (v,v) \in \mathcal{S},$$
$$\forall v = uo \in \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma)), o \in Out(\mathrm{alph}(p)), v \in \mathcal{T}(p), \text{ et } (v,v) \in \mathcal{S},$$

et contenant (ϵ, ϵ). Remarquons que p n'étant pas connecté, on a $\mathrm{alph}(p) = h_X(\mathrm{alph}(p))$ et de même $\mathcal{T}(p) = h_X(\mathcal{T}(p))$.

Soit $S'_p = \{(u,u) \mid u \in \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma)) \cap \Pi_{\mathrm{alph}(p)}(L_{\mathcal{A}})\}$ une relation, montrons que cette relation est contractuelle dans le cas où p est une facette.

- soit $i \in In(\mathrm{alph}(p))$ tel que $ui \in \Pi_{\mathrm{alph}(p)}(\mathcal{T}(\gamma))$. Il existe $v \in L_{\mathcal{A}}$ tel que $\Pi_{\mathrm{alph}(p)}(v) = u$. Par respect de l'indépendance des facettes par $\mathcal{A}$, il existe un mot $vwi \in L_{\mathcal{A}}$ avec $\Pi_{\mathrm{alph}(p)}(w) = \epsilon$, donc $\Pi_{\mathrm{alph}(p)}(vwi) = ui$ et par conséquent $(ui, ui) \in \mathcal{S}'_p$.
- soit $o \in \mathrm{alph}(p)$ tel $uo \in \Pi_{\mathrm{alph}(p)}(L_{\mathcal{A}})$. D'après la définition de $L_{\mathcal{A}}$ (équation (6.10)) et les propriétés du produit de synchronisation (équation (5.4)), on a nécessairement $uo \in \Pi_{\mathrm{alph}(p)}(\mathcal{T}(c))$.

Nous avons donc montré que pour toute facette f de $Port(\mathcal{A})$, $\Pi_{\mathrm{alph}(f)}(\mathcal{T}(\gamma)) \lesssim \Pi_{\mathrm{alph}(f)}(L_{\mathcal{A}})$. Par transitivité de la relation contractuelle (propriété 6.17), on donc

$$\mathcal{T}(f) \lesssim \Pi_{\mathrm{alph}(f)}(L_{\mathcal{A}}).$$

Le cas des réceptacles se résoud de manière identique et l'on montre donc que si c et d respectent les contrats de leurs facettes et réceptacles, il en sera de même pour un assemblage $\mathcal{A}$. ■

Composite

Pour pouvoir étendre ce résultat de manière inductive d'un assemblage de deux composants à un assemblage de n composants, il est nécessaire de pouvoir construire un composant à partir d'un assemblage. Nous définissons donc dans cette section la notion de *composite* qui est un composant formé à partir d'un assemblage. Pour ce faire, il faut fournir au nouveau composant une identité propre et veiller à ce que les ports non connectés soient correctement renommés, afin d'éviter des collisions de noms avec des ports de mêmes noms fournis par d'autres composants participant du même assemblage. On obtient ainsi une entité autonome et opaque qui se comporte comme n'importe quel autre composant.

Définition 6.32 (Composite) Étant donné un assemblage $\mathcal{A} = \langle B, X \rangle$, on définit l'instance de composant $c = \langle P, L_c \rangle$ comme suit :

- $P = \bigcup_{c_i \in B} \{(c_i^p, c, T, g) \mid (p, c_i, T, g) \in Port(\mathcal{A})\}$ est l'ensemble des ports de c ;
- $L_c = h_c(L_{\mathcal{A}})$ est le langage de c ;

avec le morphisme h_c défini comme :

$$\begin{array}{lll} h_c : & \mathrm{alph}(\mathcal{A}) & \longrightarrow \quad \mathrm{alph}(c) \\ & (c_i, p, \gamma_2, \varrho_2, k, m, \vec{x}, v) & \longmapsto \quad (c, c_i^p, \gamma_2, \varrho_2, k, m, \vec{x}, v) \text{ si } (c_i, p) \notin elem(X) \\ & (\gamma_1, \varrho_1, c_i, p, k, m, \vec{x}, v) & \longmapsto \quad (\gamma_1, \varrho_1, c, c_i^p, k, m, \vec{x}, v) \text{ si } (c_i, p) \notin elem(X) \\ & x & \longmapsto \quad \epsilon \text{ sinon.} \end{array}$$

Classiquement, l'opération d'encapsulation que représente la formation d'un composite à partir d'un assemblage a pour effet de supprimer du langage de ce dernier les messages relatifs aux ports connectés. Les ports restants, non connectés, sont alors renommés et l'on obtient une nouvelle instance de composant d'identité c.

Assemblage fiable

Une dernière condition pour étendre la composition de deux composants à n composants assemblés sur un ensemble de connexions est liée à la *topologie* des connexions. Pour tout ensemble de connexions X, on peut définir un graphe dirigé $G_X = (S, A)$ où S est l'ensemble des sommets du graphe et $A \subseteq S \times S$ est l'ensemble des arcs du graphe, construit comme suit :

- $S = elem(X)$;
- $A = \{((c,p),(c,p')) \mid p \in \mathcal{F}(c) \cup \mathcal{S}ink(c) \wedge p' \in \mathcal{R}(c) \cup \mathcal{S}ource(c)\} \cup \{((c,p),(c',p')) \mid (c,p,c'p') \in X\}$.

Les arcs du graphe G_X matérialisent non seulement toutes les connexions de X mais aussi la « dépendance » pouvant exister entre les ports offerts — facettes et puits — d'un composant et ses ports requis — réceptacles et sources : tous les nœuds des premiers sont reliés aux nœuds des seconds. On pourrait envisager d'affiner cette relation de dépendance en analysant le langage du composant et en identifiant les ports qui sont réellement causalement dépendants.

Définition 6.33 (Ensemble de connexions consistant) Soit X un ensemble de connexions, X est *consistant* si et seulement si le graphe de connexion induit par X, G_X, est acyclique.

Théorème 6.34 (Compositionnalité) *Soit $\mathcal{A} = \langle B, X\rangle$ un assemblage tel que l'ensemble X des connexions soit* consistant *(définition 6.33) et tel que tout composant $c \in B$ soit un* système fiable *(définition 6.22), alors $\mathcal{A}$ est un système* fiable *pour l'ensemble de ses ports non connectés $Port(\mathcal{A})$.*

Preuve 6.35 Le graphe $G_X = (S, A)$ induit par l'ensemble de connexions X (définition 6.33) étant dirigé et acyclique, on peut énumérer $elem(X)$, l'ensemble des sommets de G_X, suivant un tri topologique associé à l'ordre partiel induit par les arcs de G_X. D'où

$$\forall s, s' \in S, s \leq s' \Leftrightarrow \begin{cases} s = s' \\ \text{ou } (s,s') \in A \\ \text{ou } \exists s'' \in S, s \leq s'' \text{ et } s'' \leq s'. \end{cases}$$

En particulier, si $(c, r, d, f) \in X$, $(c, r) \leq (d, f)$ et par acyclicité du graphe, il n'existe pas de ports p dans $Port(c)$, q dans $Port(d)$ tels que $(d, q) \leq (c, p)$. Soit $n = |B|$, alors l'ordre défini ci-dessus permet de définir une *bijection* $\iota : B \to \{1, \ldots, n\}$ qui associe un indice à chaque composant. On a donc $\iota(c) = \iota(d) \implies c = d$ et

$$\iota(c) < \iota(d) \implies \forall (c, p, d, q) \in X, p \in \mathcal{R}(c) \cup \mathcal{S}ource(c) \text{ et } q \in \mathcal{F}(d) \cup \mathcal{S}ink(d).$$

Par induction sur l'ordre de ι :

- pour c tel que $\iota(c) = 1$, par hypothèse c est fiable et donc $\mathcal{A} = \langle\{c\}, \emptyset\rangle$ est fiable ;
- soit $k \in \{1, \ldots n\}$, on suppose que le composite $\mathcal{C}_k$ produit à partir de l'assemblage $\mathcal{A}_k$ tel que $\mathcal{A}_k = \langle\{\iota^{-1}(1), \ldots, \iota^{-1}(k)\}, X_k = \{(c, p, d, q) \mid \iota(c) \leq k \text{ et } \iota(d) \leq k\}\rangle$ est fiable. Soit $\mathcal{A}_{k+1}$ l'assemblage défini par $\mathcal{A}_{k+1} = \langle\{\mathcal{C}_k, \iota^{-1}(k+1)\}, X_{k+1} = \{(\mathcal{C}_k, c_j^p, d, q) \in X \mid d = \iota^{-1}(k+1), (\iota(j), p, d, q) \in X\}\rangle$:
  - si $X_{k+1} = \emptyset$, alors $\mathcal{A}_{k+1}$ est l'union sans connexion de deux composants fiables et par conséquent est fiable,
  - sinon, par application du lemme 6.28 avec $c = \mathcal{C}_k$ et $d = \iota^{-1}(k+1)$, on a bien $\mathcal{A}_{k+1}$ est fiable ce qui termine l'induction. ■

6.3.4 Composition & Modèles

Nous avons dans les précédentes sections défini la composition en termes d'*instances de composants*. On peut tout aussi bien considérer qu'il est intéressant de spécifier directement un assemblage. Pour ce faire, nous étendons la syntaxe et la sémantique du langage FIDL (voir chapitre 4).

La syntaxe — au niveau IDL3 du langage — de la définition d'un *composite* est décrite par la grammaire dans la figure 6.4. Elle introduit un mot-clé `assembly` et permet de décrire le composite correspondant en

$$
\begin{array}{rcl}
Composite & \rightarrow & \texttt{assembly}\ \mathsf{id}\ \{\ Assembly\ \}; \\
Assembly & \rightarrow & Assembly\ Component \mid Assembly\ Connection \mid Assembly\ Hide \mid \epsilon \\
Component & \rightarrow & \texttt{component}\ Type\ \mathsf{id}\ ; \\
Connection & \rightarrow & \texttt{connection}\ Port\ Port\ ; \\
Hide & \rightarrow & \texttt{hide}\ Port\ ; \\
Port & \rightarrow & \mathsf{id.id} \\
Type & \rightarrow & \mathsf{fqname}
\end{array} \tag{6.11}
$$

FIG. 6.4 – Syntaxe des composites.

nommant les composants qui le réalisent et en définissant les connexions entre ces composants. Les lexèmes `id` et `fqname` désignent respectivement un identifiant valide en IDL3 et un nom qualifié.

Cette syntaxe permet de définir un nouveau — type de — composant par assemblage de composants plus primitifs. Les définitions relatives aux assemblages (définition 6.25) et aux composites (définition 6.32) doivent être simplement adaptées : le terme instance de composant désigne alors simplement un composant nommé de l'assemblage. Les définitions de langages et d'alphabets sont identiques.

L'exemple de la figure 6.3 peut ainsi être défini comme le fragment FIDL de la figure 6.5.

```
1  component C {
2     provides F f;
3     consumes P p;
4     uses R1 r1,R2 r2;
5     emits S1 s1, S2 s2;
6     ...
7  }
8
9  component D {
10    provides R1 g1, G2 g2;
11    consumes P q1,Q2 q2;
12    uses U u;
13    emits T t;
14    ...
15 }
16
17 assembly A {
18    component C c;
19    component D d;
20    connection c.r1 d.g1;
21    connection c.s1 d.q1;
22 }
```

FIG. 6.5 – Exemple d'assemblage.

La seule modification importante concerne le *masquage* de ports ce qui implique de redéfinir l'ensemble des ports d'un composite.

Définition 6.36 (Masquage de ports) Soit C un composant et $H \subseteq \mathcal{F}(C) \cup \mathcal{S}ink(C)$ un ensemble de ports *masqués*, alors on définit le composant $C' = \langle Port(C) \setminus H, \mathcal{T}(C) \rangle$.

On notera que le langage de C' est identique au langage de C. La suppression des ports masqués de l'ensemble de ports implique nécessairement l'impossibilité de se connecter sur les ports masqués donc l'impossibilité de « produire » les mots induits par un appel ou un message sur une facette ou un puits masqué. Par indépendance des facettes, on est assuré toutefois de conserver un comportement fiable du composant dont les ports sont masqués.

6.4 Héritage

Dans le langage FIDL, les interfaces représentent des contrats proposés par des composants au travers de facettes et utilisés par ceux-ci au travers de réceptacles. Au niveau purement syntaxique de l'IDL3, les problèmes posés par l'héritage sont extrêmement réduits, d'autant plus que le langage n'autorise pas la surcharge de méthodes. Une interface héritant d'une ou plusieurs autres interfaces *importe* simplement des méthodes — signatures — des super-interfaces et offre une garantie minimale au client.

Lorsqu'on s'intéresse aux interfaces en tant que spécification comportementale des communications entre deux composants au travers de ports identifiés, l'héritage pose divers problèmes :

- peut-on dans une sous-interface redéfinir le comportement d'une méthode, c'est-à-dire changer le contenu des paramètres attendus en entrée et retournés en sortie par la méthode — bien entendu sans changer leur type ?
- peut-on étendre le comportement d'une super-interface dans une sous-interface avec d'autres méthodes et de quelle manière ?
- peut-on restreindre le comportement d'une super-interface dans une sous-interface, la spécialiser et de quelle manière ?
- que se passe-t-il lorsqu'une interface hérite de plusieurs interfaces définissant les mêmes méthodes — signature identique — mais avec des comportements différents ?
- comment est défini — et calculé — l'ensemble de traces d'une sous-interface par rapport à ses super-interfaces et, corollaire, que doit-on réécrire et que peut-on réutiliser ?

La relation de sous-typage la plus communément utilisée dans le domaine des langages orientés objets est la notion de *sous-typage comportementale* introduite dans Liskov et Wing [90] et qui s'énonce comme suit (traduction par nos soins) :

Définition 6.37 (Sous-typage Comportemental) Si $\phi(x)$ est une propriété prouvable pour tous les objets x de type T, alors $\phi(y)$ doit être vraie pour tous les objets y de type S où S est un sous-type de T.

Dans le contexte de Liskov et Wing [90], les propriétés auxquelles on s'intéresse sont des propriétés de *sûreté* : invariants sur l'état observable des objets d'un type et comportement (propriétés sur l'historique), les spécifications étant exprimées sous la forme d'assertions logiques (invariants de type, pré- et post-conditions sur les méthodes). Ceci revient à dire que l'on peut toujours substituer un y pour un x sans remettre en cause le fonctionnement des *clients* de x.

6.4.1 Relation de sous-typage

On va chercher à définir dans le cadre de spécifications FIDL une notion de sous-typage qui préserve les propriétés de compositionnalité des composants et donc le contrat existant entre un fournisseur et un utilisateur d'une interface donnée : pour tout réceptacle r de type I, on doit pouvoir définir une connexion avec une facette f de type I ou d'un sous-type de (une interface descendante de) I.

Pour ce faire, on va naturellement utiliser la *relation contractuelle* définie précédemment (voir définition 6.15).

Définition 6.38 (Sous-typage comportemental) Soient deux interfaces $I = \langle meth(I), \mathcal{T}(I)\rangle$ et $J = \langle meth(J), \mathcal{T}(J)\rangle$, J est un sous-type comportemental de I, ce qui est noté $I \sqsubseteq J$ si et seulement si :

1. $\mathrm{alph}(I) \subseteq \mathrm{alph}(J)$;
2. et $\mathcal{T}(I) \lesssim \mathcal{T}(J)$.

Autrement dit, une sous-interface acceptera au moins toutes les entrées de sa super-interface et produira au plus toutes les sorties qu'aurait produite celle-ci, ce qui correspond aux propriétés usuelles d'affaiblissement des pré-conditions et de renforcement des post-conditions. Cette définition autorise une sous-interface à redéfinir le comportement de sa super-interface tant que la transparence est respectée pour les clients de la super-interface.

Types de données

La relation de sous-typage comportemental s'étend naturellement aux types des paramètres des messages et des valeurs de retours. Il est alors nécessaire de respecter la contravariance pour les paramètres en mode **in** et la covariance pour les paramètres en mode **out** et les valeurs de retours. Les paramètres en mode **in-out** ne peuvent donc varier.

Nous supposons l'existence d'une relation de sous-typage sur l'ensemble des données $\mathcal{D}$ qui peut être simplement la relation d'inclusion des ensembles en considérant que les types sont des parties de $\mathcal{D}$. Pour tout couple de types T et S, S est un *sous-type* de T si et seulement si $\mathcal{D}_T \subseteq \mathcal{D}_S$.

Soient deux interfaces I et J telles que J est un sous-type de I, et deux méthodes $m_I = T\ m(T_1, \ldots, T_n) \in meth(I)$, $m_J = T'\ m'(T'_1, \ldots, T'_n) \in meth(J)$. Alors, m_J est une *redéfinition* valide de m_I pour la relation de sous-typage si $m' = m$ et si :

– les types des paramètres en mode **in** sont plus « larges » dans m_J :

$$\forall i, i \leq in(m_I), \mathcal{D}_{T_i} \subseteq \mathcal{D}_{T'_i};$$

– les types des paramètres en mode **out** sont plus « larges » dans m_I :

$$\forall i, i \geq out(m_I), \mathcal{D}_{T'_i} \subseteq \mathcal{D}_{T_i}.$$

Notons que le langage IDL interdit la *surcharge* des méthodes — l'utilisation d'un même nom pour deux méthodes différentes.

Exceptions

L'ensemble des exceptions susceptibles d'être levées par une méthode m est noté $exceptions(m)$. Les messages d'exception constituent une classe différente des appels et des retours de méthode, mais qui est prise en compte dans le langage de l'interface. Un appel de méthode et un retour d'exception sont autorisés par les règles de formation des langages d'interfaces (voir la définition 6.10).

Par conséquent, pour deux interfaces I et J avec $I \sqsubset J$ et deux méthodes $m \in meth(I)$, $m' \in meth(J)$, m' est une redéfinition valide de m pour la relation de sous-typage si $exceptions(m') \subseteq exceptions(m)$: une méthode peut choisir de lever moins d'exceptions.

Exemple

Nous illustrons ces notions de sous-typage par un exemple classique, celui des comptes bancaires. La figure 6.6 reprend la spécification d'une interface de compte bancaire définie dans le chapitre 4 : tout retrait dépassant le solde courant du compte est interdit.

On peut étendre cette interface par un compte gérant les découverts, `CreditAccounts`, représentée dans la figure 6.7. Une nouvelle méthode permet de fixer un niveau de crédit pour un compte bancaire donné. Le comportement de l'interface est modifié pour tenir compte du niveau de crédit lors d'une rentrée. `CreditAccounts` est bien un sous-type comportemental de `Accounts` puisque dans ce cas précis, le langage de `Accounts` est inclus dans le langage de `CreditAccounts`.

L'interface `InfiniteCreditAccounts` est aussi un sous-type comportemental de `Accounts` alors même que le langage de celle-ci n'est pas inclus dans le langage de celle-là puisque la méthode `withdraw` ne retourne jamais la valeur `False` : il s'agit d'un cas — financièrement dangereux ! — de réduction du non-déterminisme de sortie induit par une spécification.

6.4.2 Sous-typage & Connexions

Le principal intérêt de pouvoir définir une relation de sous-typage est bien entendu que cette relation s'intègre harmonieusement dans les mécanismes de connexion et de composition du langage. Nous modifions donc la définition 6.23 d'une connexion pour autoriser la mise en relation de ports dont les types respectifs appartiennent à une relation de sous-typage comportemental.

```
interface Accounts {
  long    balance (in long accountNo) ;
  boolean withdraw (in long accountNo, in long amount) ;
  void    deposit (in long accountNo,in long amount) ;

    /***
      header
        -- retourne le solde courant pour un numero de compte donne
        bal : Trace,long -> long;
      FIDL
       (
       ((n in ->balance(n) (s : s == bal(Trace,n) in <-balance(:s)))) +
       (n,m in ->withdraw(n,m) (ko : ko == (m <= bal(Trace,n)) in <-withdraw(:ko))) +
       deposit(_,_)
       )*

       ...
     */
 };
```

FIG. 6.6 – Compte bancaire standard.

```
interface CreditAccounts : Accounts {
    boolean setCreditLevel(in long accountNo, in long montant);

    /***
      header
        -- retourne le dernier niveau de credit du compte courant
        credit : Trace,long -> long;
      FIDL
       (
       ((n in ->balance(n) (s : s == bal(Trace,n) in <-balance(:s)))) +
       (n,m in ->withdraw(n,m) (ko : ko == (m <= bal(Trace,n)+credit(Trace,n))
              in <-withdraw(:ko))) +
       deposit(_,_) +
       setCreditLevel(_,_:_)
       )*

      body
      credit h numc =
          let
            f ( ~<-setCreditLevel(:True) :
                ~->setCreditLevel(num,somme) :  h) numc
              = if (num == numc)
                then somme
                else (f h num);
            f [] _ = 0
          in
             f (reverse h) numc
     */
 };
```

FIG. 6.7 – Compte bancaire avec découvert.

```
interface InfiniteCreditAccounts : Accounts {

    /***
      FIDL
       (
       ((n in ->balance(n) (s : s == bal(Trace,n) in <-balance(:s)))) +
        ->withdraw(_,_:True) +
       deposit(_,_)
       )*
     */
};
```

FIG. 6.8 – Compte bancaire avec crédit illimité.

Définition 6.39 (Connexion) Une connexion χ entre deux ports (p, c, I, g) et (p', c', J, g') appartenant respectivement à deux instances de composants c et c' et tels que
- $(g, g') \in \{(\texttt{receptacle}, \texttt{facet}), (\texttt{source}, \texttt{sink})\}$;
- et $I \sqsubseteq J$;

est notée $\chi = (c, p, c', p')$. Son alphabet et son langage sont définis comme précédemment (définition 6.23).

6.4.3 Substitution de composants

Du point de vue de la conception d'une architecture à base de composants, une question des plus intéressante est de savoir s'il est possible de remplacer un composant C par un composant D. Si le composant D est le résultat d'un assemblage, on définit ainsi un processus de *raffinement* d'architecture. Cette substitution doit d'une part préserver la fiabilité du composant substitué, d'autre part préserver sa sémantique.

De manière assez évidente, un composant peut-être substitué à un autre composant s'il offre au moins les mêmes services. Par contre, il n'est pas requis que le composant de substitution possède les mêmes dépendances que le composant substitué. Si C et D sont des composants fiables, au sens de la définition 6.22, et si l'on substitue C à D dans un assemblage fiable, alors la substitution préserve la fiabilité.

Du point de vue de la conception d'une architecture, le fait de préserver la sémantique du composant est une propriété bien plus intéressante que la simple préservation de la fiabilité. La préservation du langage d'un composant substitué ne peut être maintenue par la seule définition de règles de décomposition et il est alors nécessaire de s'intéresser au langage du composant de substitution et de s'assurer de sa « conformité » avec celui du composant qui est remplacé. Cette problématique sort du cadre strict de ce travail et constituera un axe de recherche ultérieur.

6.5 Conclusion

Le modèle que nous avons défini présente un certain nombre de caractéristiques intéressantes pour raisonner sur des architectures de composants ouvertes. Le fait qu'il soit basé intégralement sur des langages et des opérations préservant la rationalité des langages — morphismes, produits de synchronisation et de mélange, intersections, ... — nous permet, dans le cas où les types de données transportées par les messages sont finis, de raisonner avec des outils classiques tels que des *contrôleurs de modèles* sur des modèles arbitrairement complexes. La structure particulière de l'alphabet nous permet de travailler à différents niveaux de composition avec les mêmes méthodes et raisonnements et surtout nous permet de choisir le niveau de détail auquel on souhaite s'intéresser.

La propriété de fiabilité des composants dont nous avons montré qu'elle était préservée par la composition moyennant certaines restrictions sur la topologie des connexions est intéressante car elle permet dans un grand nombre de cas et en particulier dans celui du test de composants, de se passer d'une vérification globale d'un système réalisé par assemblage de plusieurs composants. Cette possibilité peut être particulièrement utile pour le raisonnement dans les problèmes d'adaptation ou de substitution d'architectures de composants.

Chapitre 7

Vérification & test

Le modèle d'architecture de composants que nous avons détaillé dans les chapitres précédents s'appuie sur une sémantique exprimée sous forme d'automates dont les alphabets distinguent les entrées et les sorties. Nous avons vu dans le chapitre 3 que cette catégorie de système constituait les bases d'une théorie du test de protocoles de communication. L'objectif de ce chapitre est de montrer comment cette théorie du test d'IOLTS peut être utilisée pour tester la conformité d'une implantation d'un composant par rapport à une spécification.

Si ses principes de base peuvent être directement appliqués, les algorithmes utilisés nécessitent d'être adaptés au modèle particulier que nous avons défini. Le problème principal auquel nous sommes confrontés est celui de la fragmentation de la spécification qui se présente comme une collection d'automates synchronisés. Un deuxième problème est celui de la multiplicité des ports de communication et du parallélisme intrinsèque des composants modélisés. Enfin, la notion même de conformité ne peut être directement transposée.

Nous avons aussi dans le chapitre 6 défini des propriétés que devaient posséder interfaces, composants et assemblages. Ces propriétés devraient pouvoir être vérifiées sur les modèles FIDL. La première partie de ce chapitre sera donc consacrée au problème de la vérification de modèles FIDL eu égard aux propriétés de respect des contrats et d'indépendance des facettes.

La seconde partie de ce chapitre présente un algorithme de base séquentiel pour le test de composants modélisés comme des IOLTS, synthèse de plusieurs algorithmes de la littérature. À partir de cet algorithme, nous définissons plus précisément ce que l'on entend par la notion de conformité d'un composant et les problèmes spécifiques posés par sa vérification au moyen du test. Nous proposons enfin un processus de test intégrant ces différents éléments et permettant la validation par le test de composants concrets par rapport à une spécification FIDL.

7.1 Vérification

Dans le chapitre 6, section 6.3, les propriétés des composants — indépendance des facettes, respect des contrats des ports — et des assemblages — acyclicité du graphe de connexion — permettent de produire *par construction* des *composites* possédant ces propriétés sans qu'il soit nécessaire de redémontrer leur existence. Le modèle FIDL est donc réellement *compositionnel* pour ces propriétés fondamentales qui sont préservées par l'opération de composition de composants et de production de composites.

Il reste bien sûr à vérifier que les composants atomiques utilisés dans un assemblage possèdent effectivement ces propriétés. Cette vérification relève *a priori* des techniques de *vérification de modèles* — *model-checking* en anglais — qui constitue un champ de recherche particulièrement actif. Ceci ne relève toutefois pas du cœur du sujet de cette thèse et nous nous contentons dans cette section d'esquisser une stratégie permettant de construire un problème de *model-checking* à partir d'un modèle FIDL de composant pour vérifier les deux propriétés nécessaires à la compositionnalité des composants :

– l'indépendance des facettes ;
– le respect des contrats liés aux différents ports.

7.1.1 Indépendance des facettes

La vérification de l'indépendance des facettes est *a priori* plus simple que celle du respect des ports. La propriété que nous avons défini dans le chapitre 6, section 6.2.3, distingue les entrées des sorties. Nous utilisons ici une autre formulation de cette propriété, plus faible puisqu'elle ne considère pas toutes les entrées mais uniquement les entrées abstraites. Cette formulation présente l'avantage de permettre une vérification incrémentale en utilisant des algorithmes classiques sur les automates de mots.

Proposition 7.1 Soit $C = \langle Port(C), \mathcal{T}(C)\rangle$ un composant et soit $\Lambda_C = h_\lambda(\mathcal{T}(C))$ le langage associé à C sur l'*alphabet abstrait* des messages. Alors, pour toute facette f de C, C respecte l'indépendance de f ce qui est noté $C \perp f$ si et seulement si :

$$\forall u \in \Lambda_C, \quad \prod_{\bigcup_{p\in\mathcal{F}(C)\cup Sink(C)} h_\lambda(\mathrm{alph}(p))}(u^{-1}\Lambda_C) = \coprod_{p\in\mathcal{F}(C)\cup Sink(C)} \Pi_{h_\lambda(\mathrm{alph}(p))}(u^{-1}\Lambda_C). \tag{7.1}$$

Pour toute séquence u de messages abstraits — voir équation (6.1), p.98 — du langage du composant, la projection sur l'union des alphabets abstraits des facettes et puits du composant du langage résiduel de u est égale au produit de mélange de la projection du langage résiduel de u sur l'alphabet abstrait de chacune de ses facettes et puits. Autrement dit, pour chaque facette et puits du composant, il existe toujours dans le langage du composant une séquence de messages où n'intervient aucun autre facette ou puits du composant.

Algorithmique

Pour calculer Λ_C, on pourrait envisager de construire explicitement le produit de synchronisation entre les *abstractions* des différents automates composant $\mathcal{T}(C)$. Malheureusement, dans le cas général, pour un morphisme alphabétique α et deux langages L_1 et L_2 sur les alphabets Σ_1 et Σ_2, on n'a pas l'égalité

$$\alpha(L_1 \underset{\Sigma_1,\Sigma_2}{\sqcap\!\sqcap} L_2) = \alpha(L_1) \underset{\alpha(\Sigma_1),\alpha(\Sigma_2)}{\sqcap\!\sqcap} \alpha(L_2),$$

comme le montre le contre-exemple simple suivant :

$$\Sigma_1 = \Sigma_2 = \{a, b\} \quad L_1 = \{ab\} \quad L_2 = \{bb\} \quad \alpha(a) = \alpha(b) = b$$
$$\alpha(L_1 \sqcap\!\sqcap_{\Sigma_1,\Sigma_2} L_2) = \emptyset \quad \neq \quad \alpha(L_1) \sqcap\!\sqcap_{\alpha(\Sigma_1),\alpha(\Sigma_2)} \alpha(L_2) = \{bb\}.$$

Cette égalité nécessiterait que h_λ fût un morphisme bijectif, ce qu'il n'est pas. On est donc contraint de calculer le produit de synchronisation entre les automates FIDL complets ce qui est une opération complexe — voire indécidable — dans le cas général. La section 5.3, p.92, contient deux approches de réduction du problème en fonction d'hypothèses sur les domaines de valeurs des variables.

Le calcul de $u^{-1}\Lambda_C$, pour tout $u \in \Lambda_C$, se ramène à la déterminisation de l'automate représentant Λ_C : dans un automate déterministe, le langage reconnu à partir d'un état q, noté L_q, est précisément le langage résiduel de L ou quotient à gauche de L pour tous les mots u tels qu'il existe un chemin dans l'automate entre q_0 et q. Remarquons que l'on pourrait aussi calculer l'*Automate Fini à État Résiduel* ou AFER de Λ_C qui présente les mêmes propriétés avec l'avantage dans certains cas d'obtenir une réduction significative du nombre d'états nécessaires (voir Denis *et al.* [52] pour plus de détails). Ce calcul n'est toutefois pas moins complexe que celui de la déterminisation et l'on ne sait pas prédire lequel des deux automates possédera le moins d'états. Les opérations de projection et de produit de mélange sont quant à elles triviales à réaliser.

Ce calcul peut être réalisé de manière incrémentale à partir des états initiaux des automates FIDL de la spécification de C. D'un point de vue algorithmique, on voit donc que le problème de la vérification de l'indépendance des facettes, quoiqu'en apparence plus simple, repose sur celui du calcul du produit de synchronisation d'automates FIDL.

7.1.2 Respect des ports

La définition de la propriété de relation contractuelle nous donne immédiatement un algorithme de vérification de l'existence de cette relation en la construisant effectivement par un parcours en parallèle des automates reconnaissant les langages mis en œuvre dans la relation.

Ce parcours s'apparente lui aussi à un algorithme de synchronisation d'automates ou à un calcul de bisimulation. La figure 7.1 est une représentation de celui-ci sous la forme d'une fonction prenant en paramètre deux automates finis non déterministes A_1 et A_2 sur l'alphabet $\bar{\mathcal{E}}$ et retournant **true** si et seulement L_{A_2} est contractuellement fiable pour L_{A_1}. Notons que les automates, reconnaissant des langages clos par préfixe ne contiennent pas d'états terminaux distingués.

Classiquement, l'algorithme utilise un « marquage » des ensembles d'états — les automates étant non-déterministes — déjà traités représenté par un ensemble done. Les couples d'ensembles d'états à analyser sont stockés dans une pile todo, les états étant explorés en profondeur d'abord.

```
1  function contractual(A1,A2) : boolean
2  Input : A1 = (Q1, q0^1, Σ1, δ1)
3          A2 = (Q2, q0^2, Σ2, δ2)
4  Output :
5          true si L_A1 ≲ L_A2
6  todo ← ∅
7  done ← ∅
8  todo.push((({q0^1}, {q0^2})))
9
10 do
11     (S1, S2) ←todo.pop()
12     if (S1, S2) ∈ done then
13         continue
14     else
15         done ← done ∪{(S1, S2)}
16     end
17     for each {(s1, a, s1') ∈ δ1 | s1 ∈ S1}
18         if a ∈ In(Σ1) then
19             t2' ←{t' ∈ Q2 | ∃t ∈ S2, (t, a, t')}
20             if t2' = ∅ then
21                 return false
22             else
23                 todo.push(({s1'}, t2'))
24             end
25         end
26     end
27
28     for each {(s2, a, s2') ∈ δ2 | s2 ∈ S2}
29         if a ∈ Out(Σ2) then
30             t1' ←{t' ∈ Q1 | ∃t ∈ S1, (t, a, t')}
31             if t1' = ∅ then
32                 return false
33             else
34                 todo.push((t1', {s2'}))
35             end
36         end
37     end
38 while todo ≠ ∅
39 return true
```

FIG. 7.1 – Calcul de la relation contractuelle pour deux automates.

Cet algorithme est adapté au cas où les automates sont totalement dépliés et sont donc bien des automates de mots classiques, éventuellement non déterministes. Dans le cas général des automates FIDL, on se trouve bien évidemment confronté au problème de l'identification de transitions contenant des variables contraintes par des prédicats dont les valeurs possibles peuvent être des ensembles infinis. De toute évidence, sans hypothèse supplémentaire sur ces éléments, cette vérification est impossible.

Un certain nombre de techniques existent pour le *model-checking* de programmes concurrents, y compris avec des variables. L'approche introduite dans McMillan [103] et qui est à la base de l'outil SMV est une des plus connues. Cette approche permet de gérer le problème de l'explosion du nombre d'états dans l'analyse d'accessibilité d'un graphe par l'abstraction d'ensembles d'états sous forme de *Diagrammes de Décision Binaires*. Une autre approche plus récente et prometteuse, basée sur la *sémantique de jeu*, est proposée dans Abramsky *et al.* [2] et détaillée dans Ghica [65]. Dans une sémantique de jeu[3], l'exécution d'un programme est modélisée comme un *jeu à 2 joueurs* entre le programme et son environnement. Le principal intérêt de cette approche est sa compositionnalité qui permet d'interpréter des fragments de langage comme des *stratégies* et de construire une sémantique par la composition des interprétations de ces fragments. Un autre intérêt est qu'une stratégie est définie comme un langage rationnel. De notre point de vue, l'utilisation de cette sémantique présenterait l'intérêt d'unifier dans un même cadre — celui des langages rationnels — les différentes parties du langage FIDL.

Dans un certain nombre de cas simples, par exemple lorsque les données sont contraintes par des prédicats de base, en l'absence de fonctions auxiliaires complexes et/ou avec des types de données finis, la vérification peut être faite directement par traduction vers un *model-checker*.

7.2 Algorithme de test

Nous définissons dans cette section le cadre général pour le test de composants FIDL à partir des travaux sur le test de protocole étudiés dans le chapitre 3 consacré à l'état de l'art sur le test de conformité. Notre objectif est d'identifier les problèmes posés par le test de conformité de composants et les principales stratégies permettant de résoudre ces problèmes et de s'assurer de la conformité d'un composant eu égard à sa spécification.

7.2.1 Rappels

La théorie du test de conformité d'automates à entrées/sorties est basée sur les principes du *contrôle de modèle* entre l'implantation $\mathcal{I}$ et le testeur — ou observateur — $\mathcal{O}$. Informellement, on peut modéliser $\mathcal{I}$ et $\mathcal{O}$ comme étant deux langages sur un alphabet partitionné en ensembles disjoints de lettres — ou actions, ou messages — dénotant soit une entrée, soit une sortie, soit une action interne.

La structure de $\mathcal{I}$ étant inconnue, le processus de test est alors le calcul d'un *produit de synchronisation* $\mathcal{I}\|\mathcal{O}$. Dans un modèle de communication *synchrone*, chaque lettre de sortie de $\mathcal{O}$ se synchronise avec une lettre d'entrée de $\mathcal{I}$ et vice-versa. Dans un modèle de communication *asynchrone*, on introduit un ou plusieurs langages supplémentaires modélisant une *file de messages*, bornée ou non, permettant à chacun des deux langages de se synchroniser avec les files de messages de manière indépendante.

Le test est considéré comme réussi si le langage résultant du calcul du produit de synchronisation $\mathcal{I}\|\mathcal{O}$ est dans une certaine *relation de conformité* avec un langage de référence, la spécification $\mathcal{S}$. De fait, l'observateur $\mathcal{O}$ est construit en fonction de cette relation.

Si l'observateur est un *langage*, alors son pouvoir d'observation est au mieux celui de l'équivalence entre les langages ou équivalence de traces. Si l'observateur est un automate ou plus généralement un système de transition, alors des équivalences plus fines peuvent éventuellement être observées, liées à la structure du graphe de transitions.

7.2.2 Test d'IOLTS

L'algorithme générique de test pour la relation de conformité **ioco** est introduit dans Tretmans [156]. Cet algorithme présente la caractéristique d'être valide et non-biaisé (voir chapitre 3, p.46) : il détecte toutes les implantations erronées et ne rejette pas les implantations conformes. Il produit une suite de test *complète* au sens de UIT-T [158]. Son principal inconvénient est que si le comportement spécifié pour l'implantation est infini, l'algorithme ne se termine pas et produit un ensemble de tests infini.

Il est donc évident que dans le cadre d'un processus de test concret, il est nécessaire de restreindre le test à une partie finie du comportement spécifié. Cette restriction est nécessairement basée sur des méthodes heuristiques dépendant de la connaissance spécifique du testeur et des objectifs du processus de test. Toute

méthode heuristique se ramène *in fine* à valoriser certains comportements possibles par rapport à d'autres, que ce soit sous la forme d'un *objectif de test*[75, 118, 158] ou d'un critère de couverture inspiré des pratiques du test structurel[9, 56, 79, 134].

La figure 7.2 présente les bases d'un algorithme, inspiré de Pyhälä [134], permettant de vérifier la conformité d'une IUT dont le comportement est supposé être un IOLTS par rapport à une spécification. Les paramètres d'entrée sont :

- une spécification S sous la forme d'un automate FIDL ;
- une implantation à tester I supposée compatible avec S, c'est-à-dire possédant un alphabet complémentaire et disposant d'une opération reset correctement implantée.

Les fonctions auxiliaires non détaillées sont :

- la fonction output(I) (ligne 20) attend une sortie de la part de l'IUT, la variable out contenant le résultat de la sortie qui peut éventuellement être l'observation d'un blocage θ ;
- la fonction input(I,action) (ligne 43) génère une entrée vers l'IUT et retourne **false** si l'action est refusée par l'IUT.

L'algorithme de test collecte les traces ayant généré des erreurs dans la variable failures.

Le cœur de l'algorithme est constitué de la fonction select (ligne 10) — voir figure 7.3 — qui comme son nom l'indique sélectionne la prochaine action parmi toutes celles possibles dans l'état courant. Le choix est fonction de l'état courant, de l'ensemble de test généré et de la trace de test courante. Cette fonction produit l'un des résultats suivants :

- terminate : le processus de test se termine car l'objectif fixé est atteint ;
- reset : l'IUT doit être replacée dans son état initial et une nouvelle séquence de test est construite ;
- output : le testeur attend une sortie de la part de l'IUT. Le message précis n'est pas spécifié ce qui autorise les comportements non-déterministes ;
- $a \in In(\Sigma)$: le testeur doit générer une entré spécifique vers l'IUT.

Le choix de ces différentes actions est subordonné aux deux fonctions objectifAtteint et evaluation. La première décide si l'objectif global du processus de test est atteint et donc l'arrête, la seconde évalue les différentes actions possibles dans l'état courant en fonction des tests déjà réalisés et bien sûr des critères heuristiques propres au processus du test courant. Notons que dans tous les états, l'évaluation des actions possibles dans cet état est comparée aux actions possibles depuis l'état initial ce qui introduit la possibilité de choisir un reset.

Le principal problème dans le choix des actions à effectuer est celui de la prise en compte du non-déterminisme de la spécification qui peut prendre deux formes :

- dans un même état, des sorties et des entrées sont possibles ;
- dans un même état, plusieurs sorties différentes sont possibles,

ces deux formes pouvant bien sûr se combiner. Plusieurs solutions sont envisagées dans la littérature, soit au travers d'hypothèses quant aux probabilités de telle ou telle action de sortie, soit plus simplement par un choix aléatoire dans le deuxième cas d'indéterminisme. L'évaluation que nous présentons ici tend à favoriser les entrées sur les sorties dans la mesure où sont comparés le maximum de la valorisation des entrées avec le minimum de la valorisation des sorties.

Bien évidemment, nous n'avons encore rien dit sur la fonction d'évaluation elle-même ni sur la fonction déterminant l'atteinte de l'objectif du test qui en est un cas particulier.

Évaluation

La fonction d'évaluation des états ou des actions possibles est paramétrée par la spécification S, l'ensemble des tests déjà réalisés noté T et la séquence de test courante comprenant l'action choisie. Cette fonction est fortement liée à la celle d'évaluation d'atteinte de l'objectif dans la mesure où elle doit valoriser les actions concourant à rapprocher l'objectif plutôt que les autres.

Cette fonction est basée sur des critères heuristiques dont la justification réside *in fine* dans le choix d'une stratégie de test et d'un modèle de faute. Parmi la variété de critères existants, on trouvera :

- la sélection d'un sous-ensemble fini du comportement de la spécification (objectif de test). La fonction d'évaluation sélectionne les entrées en fonction de leur appartenance ou non à l'objectif de test et la fonction objectifAtteint retourne vrai si l'ensemble du comportement défini dans l'objectif de test est réalisé ;

```
1  Input : S = (Q, q0, Σ, δ)
2          I
3  Output : failures
4  T ← ∅
5  Trace ← ϵ
6  State ← {q0}
7  Failures ← ∅
8  while true
9     //  select an action
10    action ← select(S, State, T, Trace)
11    switch action
12       case terminate:
13            return Failures
14       case reset:
15            T ← T ∪{Trace}
16            State ← {q0}
17            Trace ← ϵ
18            break
19       case output:
20            out ← output(I)
21            if out != θ then
22               Trace ← Trace.out
23               //  unexpected output
24               if out ∉ δ(State) then
25                  Failures ← Failures ∪ Trace
26                  T ← T ∪{Trace}
27                  State ← {q0}
28                  Trace ← ϵ
29               else
30                  State ← fire(S, State, out)
31               end
32            else
33               //  a deadlock
34               Trace ← Trace.θ
35               Failures ← Failures ∪ Trace
36               T ← T ∪{Trace}
37               State ← q0
38               Trace ← ϵ
39            end
40            break
41       default:
42          //  select an input
43          if ! input(I, action)
44               //  refused input
45               Trace ← Trace.θ
46               Failures ← Failures ∪ Trace
47               T ← T ∪{Trace}
48               State ← {q0}
49               Trace ← ϵ
50          else
51               Trace ← Trace.action
52               State ← fire(S, State, action)
53          end
54       end
55 end
```

FIG. 7.2 – Algorithme de test séquentiel.

```
1  function select(S,q,T,Trace)
2  Input : S = (Q, q_0, Σ, δ)
3          q ∈ Q
4          T ⊂ L_S
5          Trace ∈ L_S
6  Output :
7          action ∈ {terminate, reset, output} ∪ In(Σ)
8
9  if objectifAtteint(T,S)
10     action ← terminate
11 else
12     action ← reset
13     evalin ← {(a, eval(S, Trace.a, T)) | (q, a, q') ∈ δ, a ∈ In(Σ)}
14     evalout ← {(a, eval(S, Trace.a, T)) | (q, a, q') ∈ δ, a ∈ Out(Σ)}
15     evalreset ← {(a, eval(S, a, T)) | (q_0, a, q') ∈ δ}
16     if max(evalin) < min(evalout)
17       action ← output
18     else if max(evalreset) > max(evalin)
19       action ← reset
20     else
21       action ← max(evalin)
22 end
23 return action
```

FIG. 7.3 – Fonction de sélection.

- la sélection en fonction d'une notion de distance sur les séquences de test. L'existence d'une métrique sur les mots de la spécification dans un espace continu et borné permet de définir une notion de couverture à partir de la notion de limite. La fonction d'évaluation va donc favoriser les actions améliorant la couverture et le processus s'arrête lorsqu'un certain taux de couverture est atteint ;
- la sélection en fonction de divers critères de couverture du graphe sous-jacent au système de transitions spécifié : couverture des transitions, des états, des paires de transitions, ... La fonction d'évaluation comme précédemment valorise les actions permettant d'améliorer la couverture.

Intuitivement, on voit bien que cette fonction d'évaluation ne peut se restreindre à prendre des décisions uniquement en fonction de l'état courant. Dans le cas contraire, elle court le risque de se retrouver « piégée » dans des cycles locaux. Il est donc nécessaire que cette évaluation soit réalisée jusqu'à une certaine profondeur dans le graphe sous-jacent et selon des techniques classiques issues de l'intelligence artificielle : recherche par branchement-élagage, plus court chemin (A^*), minimax, ...

7.2.3 Conformité

Dans le cadre des protocoles modélisés par des systèmes de transitions à entrées-sorties, la relation **ioco** est la plus fréquemment utilisée : sous hypothèse que l'implantation soit toujours réceptive aux entrées, cette relation définit la conformité comme une inclusion des sorties effectives de l'implantation dans les sorties permises par la spécification, et ce pour toute trace valide de cette dernière.

Dans le cas de composants FIDL, l'hypothèse de réceptivité permanente du composant à toutes les entrées possibles ne correspond pas à la réalité des objets que nous cherchons à valider et tester : les facettes et réceptacles modélisent des *échanges* de messages alternant entrées et sorties. Les ports asynchrones possèdent cette propriété de devoir accepter tous les messages mais ici les interactions sont asymétriques et un port ne peut être utilisé que dans un seul mode.

De plus, la notion de *contrat* est essentielle à l'architecture de composants que nous avons définie. Nous utiliserons donc comme relation de conformité la relation contractuelle définie au chapitre 6 en termes de langages consistants (voir définition 6.12).

On a donc une première notion de conformité qui est :

> *Un composant est conforme à sa spécification si son comportement observé sur chacun de ses ports est conforme à la spécification des interfaces typant ces ports.*

Cette première notion de conformité nous permet d'envisager de tester indépendamment chacun des ports du composant pour pouvoir conclure sur sa conformité.

Mais le composant peut posséder lui même une spécification, sous la forme d'un ou de plusieurs automates FIDL synchronisés. Cette spécification a pour effet de lier le comportement des différents ports du composant entre eux et de manière générale de restreindre le comportement observable sur les différents ports et de le rendre plus déterministe que ne l'est la spécification du port concerné : c'est le sens de la relation contractuelle.

On a donc une notion de conformité plus large qui est :

> *Un composant est conforme à sa spécification s'il est conforme à la spécification de chacun de ses ports* restreinte *par la spécification propre du composant.*

Cette deuxième conformité nécessite de prendre en compte le comportement spécifié du composant lors du test de chacun de ses ports.

Test & automates synchronisés

L'algorithme que nous avons présenté dans la section précédente est adapté au cas où l'on teste une IUT par rapport à une spécification représentée par *un* système de transition. Or de toute évidence, dans le cas de composants FIDL on se trouve confronté au problème de construire le produit de synchronisation des différents automates qui les composent. Même en se restreignant à des spécifications sans données ou avec des données de types finis, la construction effective de l'automate dénotant le comportement du composant est problématique. On notera que dans la plupart des travaux concernant le test de composants multiports ou de processus concurrents communiquants, ce problème est supposé être résolu : on teste une implantation, éventuellement susceptible de communiquer au travers de plusieurs canaux, par rapport à une spécification globale.

Pour contourner cette difficulté, il sera donc nécessaire dans le processus de test de réaliser une synchronisation locale en fonction des besoins de progression du test. Cette synchronisation locale a des conséquences importantes sur les conclusions que l'on peut tirer du test en matière de couverture : elle rend caduque toute mesure de la couverture globale du comportement du composant. Ce qui implique que la fonction d'évaluation du choix des transitions et le critère d'atteinte de l'objectif de test ne peuvent non plus être définis globalement.

On va distinguer deux approches de la conformité et deux stratégies de test :

– la première approche correspond à la vérification de la conformité vis-à-vis des spécifications des interfaces du composant ;
– la seconde à la vérification de la conformité par rapport à la spécification du composant (lorsque celle-ci est précisée).

Dans les deux cas, on sera amené à synchroniser un ensemble d'automates FIDL pour déterminer les prochaines actions à effectuer. Mais dans le premier cas, le critère d'évaluation des actions et de décision d'arrêt sera fonction de la spécification d'un port synchrone, tandis que dans le second il sera fonction des automates formant la spécification du composant.

Nous avons émis l'hypothèse qu'il était donc possible de réduire le problème de vérification d'une conformité globale à la vérification de la conformité pour chacun des ports et chacun des comportements atomiques de la spécification. Une propriété de décomposition et une technique de test dite de *push-in* ont été proposé récemment dans Xie et Dang [165]. Ces propriétés sont démontrées dans un cadre formellement proche du nôtre mais pas complétement similaire. En particulier, la notion de valeurs potentiellement infinies et de contraintes ne sont pas prises en compte et la structure des systèmes composés est plate et non arborescente comme dans une architecture de composants FIDL. Il serait intéressant de démontrer la validité de notre stratégie de test à partir de ces premiers résultats.

Algorithme

Il reste toutefois un problème qui est celui de calculer effectivement la fraction du langage global concerné par le port ou le fragment de comportement qui nous intéresse : il est évident que si on effectue d'abord le calcul de l'ensemble de traces du composant pour réaliser ensuite la projection, tout le bénéfice d'une approche compositionnelle est perdue.

Nous proposons donc un algorithme permettant de calculer une *synchronisation locale* : un mot sur un ensemble d'automates synchronisés contenant une lettre de l'alphabet de l'un des automates synchronisés. Pour un ensemble d'*automates synchronisés* $A_1, A_2, \ldots, A_n$, un ensemble d'états courant $q \in Q_1 \times Q_2 \times \cdots \times Q_n$ et une lettre a appartenant à l'union des alphabets de chacun des automates A_i, cet algorithme construit un mot $u = va$ appartenant au langage des automates synchronisés $\bigsqcap_{1 \leq i \leq n} L_{A_i}$, s'il existe.

Cet algorithme est synthétisé dans la figure 7.4 comme une procédure appelée `explore`. Cette procédure manipule des variables globales :

word cette variable contient le mot en cours de construction lors de l'exploration ;

explored contient l'ensemble des n-uplets d'états déjà explorés ;

letter définit la lettre cible à atteindre ;

$\mathcal{A}$ est l'automate synchronisé qui est exploré.

Il est évident que ces variables ne sont globales que pour permettre une présentation plus compacte de l'algorithme. En pratique, on utilise une structure encapsulant l'état courant qui est passée en paramètre à la procédure `explore`.

Cette procédure prend un seul paramètre qui est l'état courant à partir duquel l'exploration est menée. Cet état courant est bien sûr un n-uplet d'état pour chacun des automates synchronisés $A_1, \ldots, A_n$. Le n-uplet nextstate est une variable locale contenant le résultat du déclenchement d'une transition à partir de l'état courant.

Enfin, elle utilise les fonctions auxiliaires suivantes :

- `nonSynch` prend en paramètre l'état courant et une transition et retourne `true` si la lettre cible appartient à l'alphabet de l'automate mais aucune transition marquée par cette lettre ne part de l'état courant ;
- `notAccess` prend en paramètre l'état courant et une transition et vérifie si le franchissement de la transition d mène à un état où plus aucune transition marquée par la lettre cible ne peut plus être atteinte ;
- `resolve` prend un mot *abstrait*, constitué de messages sans variables, et retourne `true` si les contraintes de l'environnement courant peuvent être résolues. Cette fonction a pour effet de bord de mettre l'environnement à jour si une solution au système de contrainte peut être trouvée ;
- `endWith` vérifie simplement si le mot se termine par la lettre cible.

La condition initiale sur le fait que l'état n'ait pas été exploré vient du fait évident qu'il n'est pas nécessaire qu'un mot terminé par la lettre cible contienne des facteurs itérés globaux. Ceci permet de limiter l'espace d'exploration au produit du nombre d'états de chaque automate, déduction faite des états improductifs qui sont les états n'étant pas coaccessibles d'un état source d'une transition étiquetée par la lettre cible.

Les deux fonctions auxiliaires `nonSynch` et `notAccess` servent aussi à élaguer l'espace de recherche. En pratique, elles peuvent être simplement implantées par des tables indexées par les transitions plutôt que par une fonction de parcours des automates.

Cet algorithme termine évidemment toujours si le nombre d'états des automates est fini ce qui est le cas par hypothèse. Dans le pire des cas, l'algorithme explorera l'ensemble des états et des transitions de chacun des automates.

Cet algorithme est bien évidemment incorrect sans restriction sur les contraintes : dans le cas général où les contraintes sont des fonctions arbitraires, la résolution d'un ensemble de contraintes peut dépendre d'un certain nombre de *tour de boucles* dans l'automate, par exemple lorsque la contrainte dépend de calculs fait sur la trace. La séquence de synchronisation calculée par l'algorithme dépend alors de la fonction `resolve` qui peut avoir un comportement arbitrairement complexe.

On fera donc l'hypothèse raisonnable que l'on peut sélectionner un mot candidat à l'aide de l'algorithme donné puis résoudre les contraintes restantes pour instancier les variables.

Ports synchrones & asynchrones

La distinction existant dans les modèles FIDL entre ports synchrones et ports asynchrones induit des effets sur la stratégie de test à mener. Les ports synchrones sont caractérisés par une alternance d'appels/retours de messages qui par définition de ce type de port ne peut introduire dans un état donné de choix entre

```
1  globals :
2           word ← ϵ
3           letter , A ← {A_1, A_2, ..., A_n},
4           explored ← ∅
5
6  procedure explore(state)
7  input : state ∈ Q_{A_1} × Q_{A_2} × ... Q_{A_n}
8   if endsWith(word, letter) ∧ resolve(word)
9        return true
10  else if state ∈ explored
11       return false
12  fi
13
14 /* start of exploration */
15  for each s_i ∈ state
16      foreach d ∈ δ_{A_i}(s_i)
17          if nonSynch(state ,d) ∨ notAccess(state ,d)
18              return false
19          else
20              nextstate = fire(A ,state ,d)
21              word ← word.d
22              if explore(nextstate)
23                 return true
24              word ← word\ d
25          end
26      end
27  end
28  return false
```

FIG. 7.4 – Recherche de lettres synchronisées.

des messages de sorties et des messages d'entrées. Le seul indéterminisme possible concerne la liberté laissée par la spécification d'autoriser plusieurs sorties pour une même entrée.

Les ports asynchrones sont quant à eux caractérisés par un sens unique de communication et par l'absence de protocole de communication précis régissant les connexions entre ports de ce type. Seule la spécification d'un composant peut permettre de contraindre les messages sur un port asynchrone, et encore ne peut-il s'agir uniquement que des messages émis. La vérification de la spécification du comportement d'un port asynchrone n'a donc de sens que dans le cadre de la vérification du comportement global du composant.

7.3 Test de composants FIDL

On va donc s'assurer de la conformité d'un composant FIDL selon un processus en deux étapes. Dans une première étape, on va vérifier la conformité du composant sur chacun de ses ports individuellement, c'est-à-dire la conformité par rapport aux interfaces. Dans une seconde étape, on va vérifier la conformité de l'IUT par rapport à la spécification du comportement du composant lui-même. Ces deux procédures complémentaires nous permettront de nous assurer d'une conformité globale, le terme d'assurance étant bien entendu à prendre avec toutes les restrictions d'usage.

7.3.1 Scénarios de test

Suivant une tradition désormais bien établie (voir van Glabbeek [161]), nous définissons ici le scénario générique du processus de test sous la forme d'*expérimentations* entre un *testeur* et une IUT médiatisée par un *contexte de test*.

La figure 7.5 représente graphiquement les différents testeurs de port disponibles. Le testeur de facette est une boîte noire possédant un ensemble — fini — de *boutons* et un écran sur lequel s'affichent un ensemble — fini — de lettres. Une expérience — un test — se déroule comme une suite d'interactions

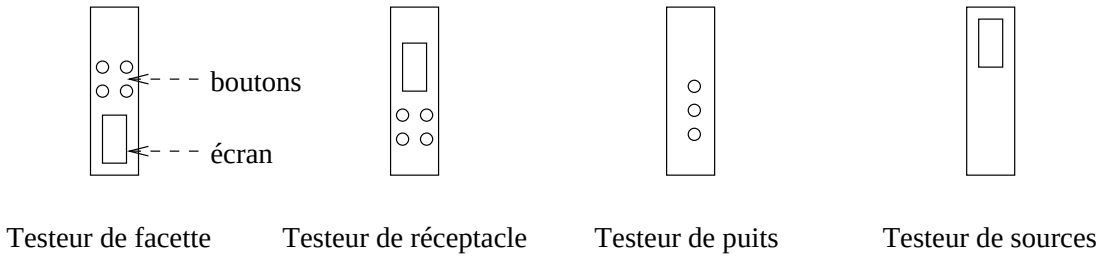


FIG. 7.5 – Testeurs de ports.

entre le testeur et la boîte :

1. le testeur pousse un bouton parmi l'ensemble des boutons disponibles ;
2. l'écran s'efface s'il contenait déjà un symbole et reste noir un certain temps, temps pendant lequel le bouton sélectionné reste enfoncé et tous les autres boutons sont bloqués ;
3. éventuellement, l'écran affiche un symbole en réponse à ce stimulus. Le bouton se relâche et tous les boutons sont à nouveau disponibles pour le testeur.

Le testeur de réceptacle a une structure identique mais le scénario d'expérimentation est légèrement différent : les boutons sont inactifs tant que rien n'est affiché sur l'écran. Les testeurs de puits — respectivement de sources — ne comprennent pas d'écrans — respectivement de boutons — et les boutons ne sont jamais bloqués.

La figure 7.6 représente un testeur pour un *composant* comprenant un certain nombre de ports répartis entre facettes, réceptacles, sources et puits. Le testeur de composant possède de plus un bouton *reset* qui permet au testeur à tout moment de remettre l'IUT dans son état initial. Les différents testeurs de ports représentés fonctionnent comme indiqué ci-dessus.

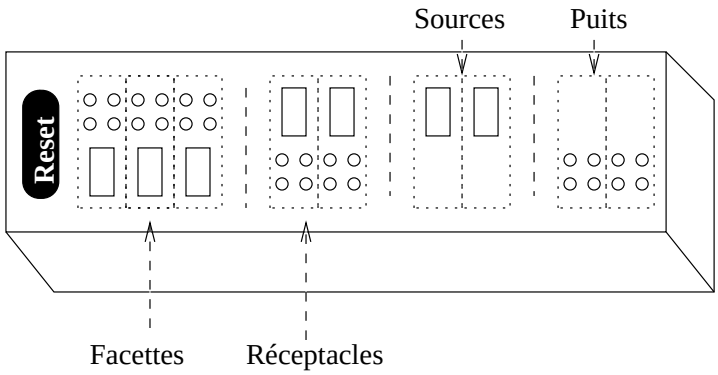


FIG. 7.6 – Testeur de composant.

L'IUT est donc vu comme une boîte noire dont le seul comportement observable est donné par les différents écrans accessibles. Ce scénario de test ne définit pas le comportement du testeur, la manière dont celui-ci agit en poussant les boutons en fonction des informations produites par le scénario de test.

Le comportement du testeur est bien évidemment déterminé par la spécification que l'on souhaite vérifier, autrement dit par un ensemble d'automates FIDL synchronisés. Parmi tous ces automates nous distinguerons les catégories suivantes :

– les automates dits *actifs* sont moteurs dans le processus de test. C'est en fonction d'eux qu'est calculée la fonction d'évaluation ;
– les automates *passifs* ne jouent aucun rôle dans le processus de décision, ils peuvent être amenés à réagir aux sollicitations de l'IUT ou produire des messages permettant au processus de test de progresser en fonction des contraintes de synchronisation.

De plus, nous distinguerons les automates dits *bien formés* dont le comportement est une alternance d'entrées et de sorties des automates quelconques dont les entrées/sorties peuvent être arbitrairement entrelacées. Les premiers correspondent à la spécification du comportement d'une interface et les seconds à une partie du comportement d'un composant.

7.3.2 Algorithme général

Pour prendre en compte l'existence de plusieurs ports de communication et de plusieurs automates synchronisés représentant la spécification, il est nécessaire de modifier substantiellement l'algorithme de base de la figure 7.2. Cet algorithme est fondamentalement séquentiel alors qu'un composant est potentiellement constitué de flots d'exécution parallèles, même si le degré de parallélisme est restreint par les synchronisations entre ports.

```
1  Globals :
2           word ← ϵ
3           letter , A ← {A_1, A_2, ..., A_n} ,
4           explored ← ∅
5
6  Input : S = {A_i = (Q_i, q_0^i, Σ_i, δ_i), 1 ≤ i ≤ n}
7          A ∈ S // un automate actif dans la spécification
8          I une implantation sous test
9  Output : failures
10 T ← ∅, Trace ←ϵ, State ←(q_0^1, ..., q_0^n), Failures ←∅
11 outerloop:
12  while true
13    action ← select(A, State ,T, Trace)
14    switch action
15       case terminate:
16            return Failures
17       case reset:
18            T ← T ∪{Trace}, State ←(q_0^1, ..., q_0^n), Trace ←ϵ
19       case acts ⊂ Σ_A :
20            for each a ∈ acts
21               word ← ϵ, letter ←a
22                explore(State)
23               if word != ϵ then
24                  execute(word,S,I,State ,Trace)
25                  continue outerloop
26               end
27            end
28            // problem
29            return Failures
30     end
31  end
```

FIG. 7.7 – Algorithme de test de composants FIDL.

Le principe de fonctionnement de cet algorithme (voir fig. 7.7) est le suivant :

- parmi l'ensemble des automates synchronisés noté $\mathcal{A}$, un est distingué comme étant l'automate actif A ;
- comme précédemment, une fonction select de sélection de la prochaine action à effectuer est appelée mais cette fois uniquement en fonction de l'automate actif A. Cette fonction retourne soit le symbole terminate, soit le symbole reset, soit un ensemble ordonné acts de lettres de l'alphabet de A, Σ_A ;
- pour chaque lettre $a \in acts$, on essaye de construire une séquence de synchronisation σ dans l'ensemble des automates $\mathcal{A}$. Cette séquence de synchronisation est un mot reconnu par les automates synchronisés $\mathcal{A}$ dans l'état courant et qui se termine par a :
  - si aucune séquence de synchronisation ne peut être construite après épuisement de acts, alors la séquence de test courante indique une *erreur dans la spécification* et le processus de test est arrêté,

– sinon, on réalise les actions de la séquence σ jusqu'à atteindre la lettre a qui en est la fin ou jusqu'à ce qu'un décalage (conforme) soit constaté sur un message de sortie. Ce dernier cas correspond à la situation où une spécification est non déterministe — plusieurs sorties sont possibles pour une même entrée. On recommence ensuite la procédure. La figure 7.9 détaille l'exécution de la séquence de synchronisation.

Cet algorithme utilise la procédure de recherche explore détaillée dans la figure 7.4 ainsi que la fonction auxiliaire `failAndReset` de la figure 7.8.

```
1  function failAndReset(Failures, T, State, Trace)
2     Failures ← Failures ∪ Trace
3     T ← T ∪{Trace}
4     State ← q0
5     Trace ← ϵ
```

FIG. 7.8 – Réinitialisation après échec.

```
1  function execute(word,S,I,State,Trace,Failures)
2       for i ∈ {1,...,#word}
3          if word[i] ∈ Out(Σ_S)
4             out ← output(I)
5             if out != θ then
6                Trace ← Trace.out
7                 //  unexpected output
8                 if out ∉ δ(State) then
9                    failAndReset(Failures,T,State,Trace)
10                else if out ≠ word[i]
11                   State ← fire(out)
12                   continue outerloop   // recommence la
13                                        // procédure de décision
14                end
15            else
16               //  a deadlock
17               Trace ← Trace.θ
18               failAndReset(Failures,T,State,Trace)
19            end
20         else if ! input(I,word[i])
21            //  refused input
22            Trace ← Trace.θ
23            failAndReset(Failures,T,State,Trace)
24         else
25            Trace ← Trace.word[i]
26            State ← fire(S,action)
27         end
28      end
29      // continue
```

FIG. 7.9 – Exécution d'une séquence de synchronisation.

On notera que dans le cas d'automates FIDL, et de manière plus général dans le cas d'IOLTS, la découverte d'un mot permettant de synchroniser différents automates sur un objectif commun n'est aucunement une garantie que l'action correspondant à la lettre recherchée soit effectivement réalisée dans le cas où certains des automates considérés sont non déterministes en sortie pour des états atteints par le mot de synchronisation.

Conformité des ports synchrones

La première phase de la vérification de la conformité d'un composant consiste donc à tester celui-ci par rapport à la spécification de chacun de ses ports. Dans ce cas de figure, l'automate correspondant au port

vérifié est considéré comme actif et tous les autres automates impliqués dans la spécification du composant sont passifs.

Conformité du composant

Dans une deuxième phase, la conformité de l'IUT au comportement spécifié par le composant est vérifiée pour chacun des automates synchronisés de la spécification du composant. Chacun alternativement est donc l'automate actif au cours d'une session de test et tous les autres automates sont passifs.

7.4 Conclusion

L'approche que nous proposons se distingue des travaux existants sur le test d'IOLTS par l'absence de spécification globale et la volonté de ne pas construire celle-ci. Cette approche se situe ainsi à mi-chemin des stratégies de test basées sur la définition d'un objectif de test *ad hoc* par le concepteur et des stratégies de test systématique basées sur la satisfaction d'un critère de couverture. De la première, nous conservons la taille relativement faible des suites de test et des testeurs en ne testant à chaque étape qu'une partie de la spécification. Mais l'utilisation d'une stratégie d'évaluation des cas de test par une fonction arbitraire définissant un certain degré de couverture permet de conserver au test unitaire de composants son caractère systématique.

Bien entendu, en l'absence de couverture complète du comportement du composant, on ne peut tirer du résultat d'une telle procédure de test que des conclusions dont la valeur dépend de la confiance mise dans les critères d'évaluation choisis par le testeur.

Chapitre 8

Mise en œuvre

Nous revenons dans ce chapitre à des aspects plus pratiques de la mise en œuvre des modèles et concepts présentés précédemment dans le cadre d'un processus de développement. L'objectif de ce chapitre est de décrire d'une part comment sont produits et manipulés les modèles FIDL d'architectures de composants et d'autre part comment la génération et l'exécution des tests à partir des modèles s'insèrent dans le processus de développement.

8.1 Vérifications & Validations

Disposant d'un modèle formel d'architecture, représenté dans le langage FIDL, la question se pose donc des validations et vérifications qu'il est possible de réaliser sur ce modèle et sur l'implantation supposée se conformer à ce modèle. Nous pouvons détailler le schéma 1.7 du chapitre 1 en indiquant les différentes activités de V&V qui peuvent intervenir dans le processus de développement (fig. 8.1).

La plupart des procédures de vérification et validation décrites dans cette section ne sont pas implantées. Notre objectif est surtout de montrer comment un modèle FIDL permettrait de réaliser des vérifications automatisés à différentes étapes du processus de développement.

8.1.1 Vérification de l'architecture

La vérification de l'architecture est le processus qui contrôle la correction de l'architecture produite eu égard à un ensemble de règles propres au formalisme FIDL. Ce processus comprend les étapes suivantes :

- vérification syntaxique du modèle FIDL ;
- vérification des règles de typage dans les comportements ;
- vérification de la relation de sous-typage comportemental ;
- vérification de la relation contractuelle pour un composant et ses différents ports ;
- vérification de l'indépendance des facettes d'un composant ;
- vérification des assemblages.

8.1.2 Validation statique

La validation statique du code par rapport à la spécification certifie que celui-ci respecte bien la structure de l'architecture FIDL de manière statique, ce uniquement en analysant le code source et en inférant de celui-ci la manière dont il représente un agencement de composants. Dans le cadre des langages orientés-objets, il existe des propositions pour étendre l'expressivité des règles d'accès entre les entités logicielles du langage[14] au delà des traditionnels accès `private`, `protected` ou `public`. Par ailleurs, il existe des outils permettant de décrire des règles d'accès de manière externe et de les vérifier sur du code classique Java : Macker (`http ://macker.sourceforge.net`) est un exemple d'un tel outil. Enfin, nous avons vu dans le chapitre 2, section 2.2.2 comment Java pouvait servir de base pour un langage de description d'architecture.

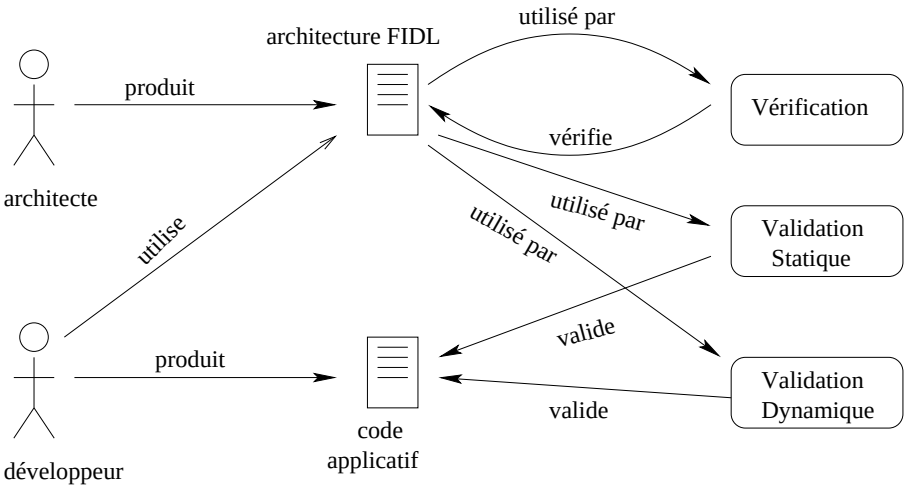


FIG. 8.1 – Validation & Vérification.

À partir du code applicatif produit par le développeur, on peut extraire une architecture par application de deux niveaux de règles :

1. des *règles syntaxiques* qui décrivent comment les éléments structuraux d'une architecture FIDL sont représentés dans le langage cible. Cette transformation, si elle est réversible, permet d'extraire de l'application un modèle FIDL et de le comparer avec la spécification ;
2. des *règles sémantiques* qui analysent le code source, en déduisent un comportement FIDL et un *typage comportemental* pour les différents artefacts architecturaux.

La première étape de validation est relativement aisée et c'est celle qui est réalisée par Macker et ArchJava. L'approche la plus générique consiste à définir des règles de transformations entre méta-modèles selon l'approche MDE. Si le langage dans lequel le code source de l'implantation que l'on souhaite valider dispose d'un méta-modèle UML, on peut alors extraire une instance de ce modèle correspondant au code de l'application, appliquer une transformation vers le modèle FIDL puis vérifier qu'il existe bien un *isomorphisme* entre ce modèle et la spécification FIDL.

La deuxième étape de validation est plus puissante mais bien plus complexe : il s'agit de construire des types comportementaux qui vont être attachés à chacun des éléments du langage du code source susceptibles d'apparaître dans un modèle FIDL, essentiellement les variables représentant des « ports » d'un « composant ».

8.1.3 Validation dynamique

La validation dynamique nécessite l'exécution de l'implantation et la vérification du respect de certaines propriétés à partir de l'observation de son comportement à l'exécution. Le modèle FIDL peut être ici utilisé de deux manières différentes :

1. soit pour générer des *assertions* exécutables ;
2. soit pour générer des *tests*.

Ces deux approches nécessitent au préalable de disposer des règles de transformations de modèles décrites précédemment.

La génération d'assertion introduit un mécanisme de contrôle dynamique similaire à ce qui est réalisé par exemple en Ada ou Eiffel[110] ou encore à l'aide d'un langage de spécification comme JML[47, 86]. Les expressions FIDL sont compilées vers le langage cible et insérées en début et fin de méthodes pour vérifier la correction de l'exécution par rapport aux spécifications. L'état des composants et ports est stocké sous forme d'un invariant qui est contrôlé par le code de vérification.

Le code modifié est ensuite sensibilisé par une suite de tests, les assertions insérées dans le code produisant un échec des tests en cas de violation de l'invariant ou des prédicats sur les paramètres des méthodes.

Cette approche présente l'intérêt d'être techniquement simple à mettre en œuvre : il suffit de modifier le code source avant la compilation ou le code-octet et d'insérer une version exécutable des expressions FIDL dans les différentes entités correspondant aux interfaces et composants de la spécification. L'inconvénient est que la construction des suites de tests reste manuelle et qu'il est nécessaire de briser l'encapsulation des composants. Si l'on souhaite tester un composant « sur l'étagère » disponible uniquement dans un format binaire encapsulé, cette solution devient plus complexe voire impraticable.

La deuxième solution consiste à générer à partir des spécifications FIDL des suites de tests exécutables : l'expression contractuelle du comportement des composants permet à la fois de construire des tests pertinents à partir des automates FIDL et de contrôler le comportement de l'environnement de l'IUT en construisant un automate miroir à partir des spécifications FIDL. Dans ce cas aussi, il est nécessaire de disposer d'une version exécutable ou interprétable des expressions FIDL. La section suivante présente en détail le processus de test de composants FIDL tel qu'il est partiellement implanté dans un prototype en cours de développement.

8.2 Outillage

Nous détaillons dans cette section les principaux composants et traitements d'un prototype d'implantation d'outil manipulant des spécifications FIDL et permettant de tester automatiquement des composants concrets à partir de ces spécifications. Cet outil étant encore en cours de développement, certaines fonctionnalités décrites ne sont encore que partiellement implantées.

L'atelier FIDL est construit à partir d'un certain nombre de composants indépendants écrits en Java, liés dans une interface utilisateur, soit en mode texte, soit en mode graphique : la figure 8.2 est une capture d'écran de cette interface graphique : en haut à gauche se trouve l'éditeur de code FIDL ; en haut à droite l'éditeur d'automates FIDL ; en bas à gauche est représentée une session d'exécution de tests ; enfin la fenêtre centrale représente l'arbre de navigation de l'architecture. Ne sont pas représenté dans la capture d'écran l'outil — sommaire — d'édition de composites et les différents écrans de configuration du testeur.

8.2.1 Analyse syntaxique & compilation

Éléments structuraux

L'architecture FIDL s'appuyant sur une partie du langage IDL3, un premier composant traite des éléments de ce langage. Il comprend un *analyseur syntaxique* et un ensemble de classes et d'interfaces permettant de représenter sous forme d'un graphe d'objets l'ensemble des éléments *structuraux* d'un modèle FIDL.

Le schéma 8.3 est une partie du méta-modèle UML pour les architectures de composants FIDL. Il ne comprend pas les éléments permettant de définir des types structuraux, ni le détail de la description des opérations et attributs.

La proximité entre le méta-modèle FIDL et le méta-modèle CCM ou plus précisément IDL3 permet de disposer des *règles de projection* existantes vers d'autres langages, règles qui sont définies par la spécification CORBA.

Ce méta-modèle peut être utilisé comme base pour des outils de transformation de modèles et permet de disposer automatiquement, au travers de la norme XMI, d'un mode de représentation XML des architectures FIDL. Un des composants de l'atelier réalise cette transformation depuis/vers XMI ce qui permet d'envisager par ailleurs d'interfacer l'outil avec d'autres ateliers de modélisation produisant du XMI.

Éléments comportementaux

Le deuxième niveau d'analyse est constitué de la partie comportementale FIDL, les expressions décrivant les automates FIDL. Rappelons que ces expressions sont incluses en tant que commentaires dans une représentation textuelle d'un modèle FIDL ou dans une représentation XMI ce qui permettra à terme d'attacher ces descriptions comportementales à différentes représentations.

L'analyseur syntaxique FIDL produit à partir des expressions un ou plusieurs automates FIDL : un dans le cas des interfaces, éventuellement plusieurs dans le cas de composants décrits par des **and**-expressions

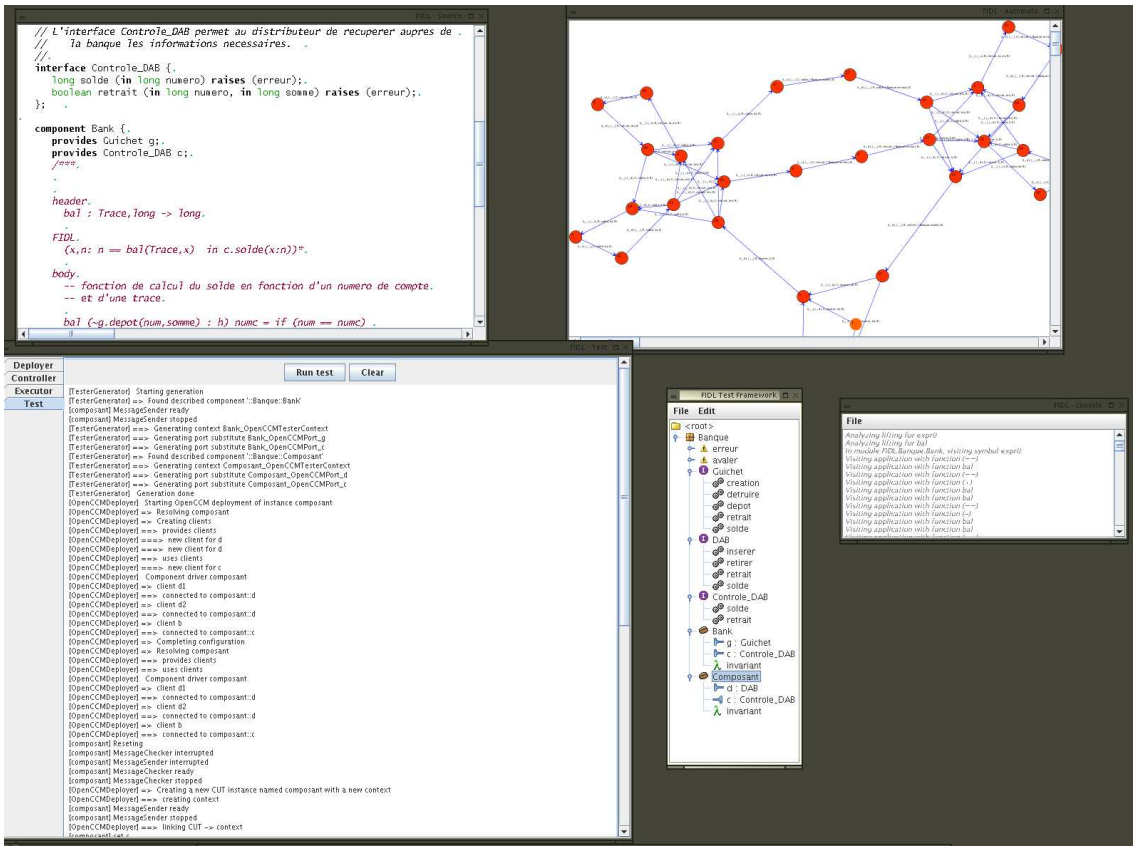


FIG. 8.2 – Interface graphique atelier FIDL

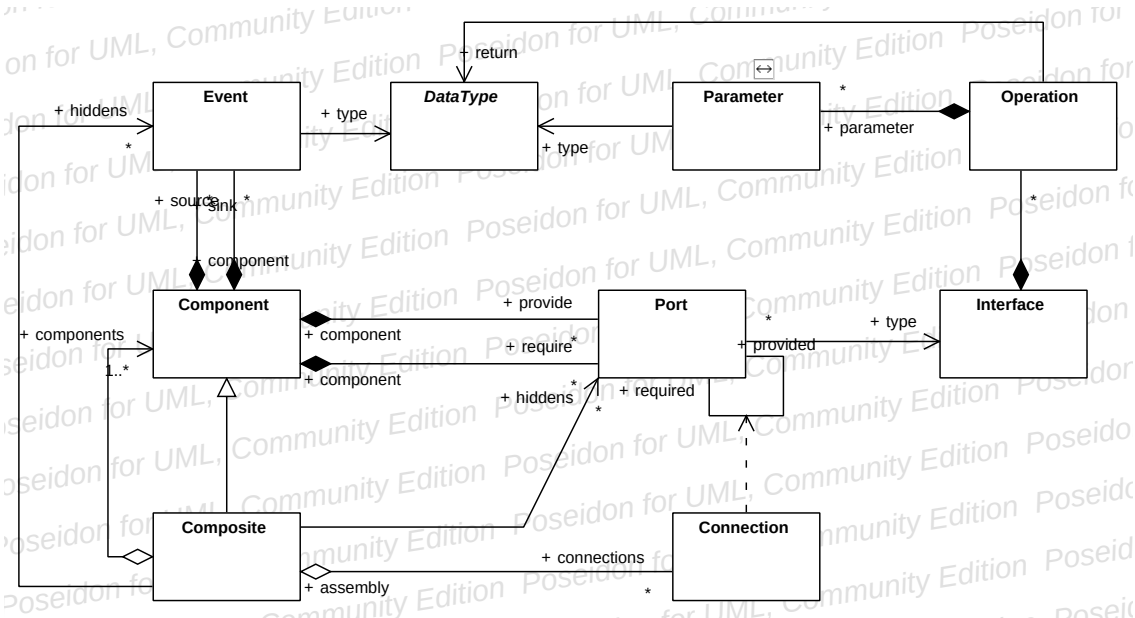


FIG. 8.3 – Méta-modèle de composants FIDL

complexes. Ces automates sont construits à l'aide d'une bibliothèque spécifique de manipulation d'automates intitulée JAuto et qui a été développée partiellement pour les besoins de ce projet. Seules les étiquettes de transition sont propres à l'atelier FIDL et sont constituées de *messages* et d'un *environnement* associé. Ces structures sont la transcription en Java de celles définies dans les chapitres 5 et 6.

Cet analyseur s'appuie sur le modèle structurel pour vérifier les types des variables utilisées dans les messages. Les types IDL3 sont traduits dans le système de type propre du langage FIDL qui est de plus bas niveau et traite de manière équivalente des structures différenciées en IDL3. Par contre, la validité des expressions FIDL par rapport aux règles d'émission-réception des messages n'est pas vérifiée à ce stade. La construction des automates FIDL est aussi totalement indépendante du langage fonctionnel choisi, le traitement de ces expressions et leur transformation en contraintes étant délégués à un composant spécialisé.

Dans l'état actuel du prototype, toutes les vérifications sont effectuées sur une version simplifiée du langage des composants et des interfaces. Ce langage simplifié ignore les contraintes et distingue uniquement les valeurs littérales et les variables, dénotées par leur type. La première étape de vérification consiste en la construction des langages des différents objets, selon les définitions du chapitre 6 : langages d'interfaces, langages de composants, langages d'assemblages.

Les vérifications réalisées sont donc :

- pour les interfaces dans une relation d'héritage, la vérification de l'héritage comportemental au sens de 6.4 ;
- pour les composants et assemblages :
  - la vérification du respect des facettes et réceptacles ;
  - la vérification de l'indépendance des facettes et des puits (sur les alphabets abstraits) ;
  - éventuellement le respect de l'héritage comportemental lorsqu'une relation d'héritage explicite est donnée.

La relation contractuelle peut aussi être vérifiée explicitement entre deux composants ou assemblages.

Éléments fonctionnels

Un composant traite plus particulièrement de l'interfaçage avec le langage Jaskell. Une bibliothèque spécialisée, version dégradée d'un compilateur Haskell, fournie l'analyse syntaxique et la compilation des expressions Jaskell vers du code-octet Java. Le composant FIDL-Jaskell construit une représentation Jaskell des types de données IDL3 et transforme chaque contrainte en une fonction produisant un résultat booléen à partir des valeurs des variables présentes dans l'environnement. Ce composant offre aussi une interface de *gestion des contraintes* utilisée lors de l'exécution des testeurs (voir ci-dessous).

8.2.2 Plate-forme de test

Le composant de test principal permet de configurer et d'exécuter des tests dirigés par un modèle FIDL sur une implantation concrète. Il offre donc une interface de configuration qui permet de définir un certain nombre de paramètres du processus de test, en particulier :

- le modèle FIDL à utiliser, construit à partir des outils d'analyse décrits ci-dessus et qui comprend quatre catégories distinctes d'objets :
  - les types de données primitifs et construits, y compris les types d'événements et les types d'exception et leur traduction dans divers langages,
  - les entités architecturales, interfaces, composants et assemblages,
  - les automates FIDL,
  - les fonctions et prédicats utilisés dans les contraintes sur les automates FIDL ;
- le *déployeur* est le composant qui a la charge d'instancier l'IUT et de construire les différents ports utilisés en fonction des techniques de communication mises en œuvre dans l'IUT ;
- l'*exécuteur* des tests est chargé du routage des messages depuis et vers l'IUT et de la simulation de l'environnement du composant testé ;
- le *contrôleur* de test qui est le moteur de génération et d'analyse des tests.

Déployeur, exécuteur et contrôleur sont des interfaces requises de la plateforme de test et donc totalement génériques. Dans l'implantation actuelle de l'outil, il existe trois types de déployeurs, un exécuteur et deux contrôleurs différents.

8.2.3 Déployeur

Le déployeur gère les aspects techniques d'interaction entre le testeur et le composant réel. Cette gestion prend deux formes :

- d'une part, il implante une transformation du modèle FIDL vers le modèle correspondant à la plate-forme d'exécution qu'il gère ;
- d'autre part il produit des instances de *ports* qui assurent la transcription des messages depuis/vers l'IUT en fonction des modalités de la communication.

Déploiement Java

Le tableau 8.4 donne informellement la traduction des éléments FIDL dans un support d'exécution Java pur, dit POJO, traduction qui est relativement simple. Pour les ports asynchrones, on utilise le patron de conception *Observateur* (voir Gamma *et al.* [62]) pour créer une interface `PuitsE`, avec pour seule méthode `void message(E e)` pour recevoir les messages de type `E`. Cette interface doit être fournie par le composant possédant un port puits. Les types et exceptions sont transcris selon la norme de projection IDL2 vers Java décrite dans la spécification CORBA.

Élément FIDL	⟷	Élément Java
interface	⟷	interface
méthodes d'interface	⟷	méthodes d'interfaces
composant	⟷	interface
facette `I p`	⟷	méthode `I getP()`
receptacle `I p`	⟷	méthode `void setP(I)`
puits `E p`	⟷	méthode `PuitsE getP()`
source `E p`	⟷	méthode `void setP(PuitsE e)`

FIG. 8.4 – Traduction FIDL-POJO.

Le déployeur Java utilise un *chemin des classes* pour construire un *chargeur de classes* propre au composant concret testé et instancie l'IUT en fonction du paramétrage du nom de la classe d'implantation choisi par le testeur.

Les instances de ports créés transforment les objets messages transmis par l'exécuteur et produits par le contrôleur (*cf. infra*) correspondant à des émissions vers l'IUT soit en appels de méthodes, soit en retour d'appels de méthodes. Les ports requis sont simulés par des instances de java.lang.reflect.Proxy.

Déploiement OpenCCM

Le déploiement pour un composant CCM est conceptuellement plus simple puisqu'on est ici très proche du modèle FIDL mais techniquement plus complexe. La partie transformation de modèle est immédiate et fournie par OpenCCM à partir des fichiers FIDL qui sont vus comme de simples descripteurs IDL3. La partie la plus technique consiste à générer les simulateurs de conteneurs qui vont encapsuler l'IUT. Le marshalling des messages est assuré en générant une classe spécifique de port CCM (voir Fontaine [58] pour plus de détails).

Communication

En fonction de la configuration choisie, le déployeur met en place un canal de communication par port du composant à tester, c'est-à-dire par port identifié dans le modèle FIDL. Il n'est pas nécessaire que tous les ports communiquent selon un même protocole. On peut ainsi envisager de *simuler* dans le processus de test différents environnements de communication :

- l'appel de méthode local (dans une même JVM) ou distant (RMI, XML-RPC ou SOAP) ;
- l'échanges de données dans des formulaires HTTP, soit directement au niveau HTTP, soit dans un contexte de *Servlets* ;
- les flux de données type Unix.

8.2.4 Exécuteur

L'exécuteur assure l'interface entre l'IUT instanciée par le déployeur et le contrôleur, il effectue le multiplexage/démultiplexage des communications. Techniquement, il s'agit simplement d'un ensemble de *flots d'exécution* : un par facette de l'IUT plus un pour tous les réceptacles et les messages asynchrones ; et de deux files de messages : une file d'entrée vers l'IUT et une file de sortie. L'exécuteur simule ainsi les différents clients de l'IUT dont les messages sont sensés pouvoir s'entrelacer arbitrairement.

8.2.5 Contrôleur

Le contrôleur est la partie la plus importante du testeur puisque c'est lui qui va orchestrer la génération des séquences de test et fournir un verdict de test. Le contrôleur le plus simple est construit à partir d'un fichier décrivant une séquence de messages, selon la syntaxe FIDL. Un compilateur permet de générer automatiquement un fichier source Java à partir d'une séquence de messages.

Contrôleur FIDL

Le contrôleur FIDL proprement dit est plus complexe et implante les techniques de parcours d'automates, de résolution de contraintes et de génération de cas de test présentées dans les chapitres précédents. Dans l'état actuel du prototype, l'algorithme de test est légèrement différent de celui présenté dans le chapitre 7 : les messages en sortie de l'IUT sont traités en priorité avant de construire des messages en entrée, ce qui implique la possibilité d'un blocage du testeur par *famine* si l'IUT génère un flux continu de messages de sorties.

Le contrôleur est paramétré par :

- le *sélecteur de messages* qui choisit un message parmi l'ensemble des messages d'entrées possibles. Par défaut il implante une stratégie d'évaluation des messages par exploration de l'automate *actif* et calcul d'une fonction d'évaluation qui peut être paramétrée selon plusieurs objectifs — couverture de transitions par exemple ;
- les *critères d'arrêts* qui sont définis dans le contrôleur et vérifiés pour chaque message effectivement exécuté ;
- le *gestionnaire de contraintes* dont le fonctionnement dépend du langage d'implantation de la partie fonctionnelle du langage FIDL.

Pour chaque automate de la spécification, on construit un *état* formé de l'état de l'automate et de son environnement (voir section 5.1.5) qui est mis à jour par le franchissement de transitions. Bien entendu, la *trace* courante est aussi maintenue et mise à jour.

Environnement & Contraintes

Le composant Jaskell maintient l'environnement sous la forme d'un arbre reliant les variables entre elles selon leur relation de dépendances. Les contraintes sont résolues simplement par un parcours de l'ensemble des solutions et un *backtracking* jusqu'à atteindre une valuation satisfaisant les contraintes. Ces contraintes sont des fonctions Jaskell compilées qui sont évaluées lors du positionnement de chaque variable. Cette stratégie est assez primitive (voir section 5.3.1) et devrait être améliorée pour un passage en production.

Le composant utilise par ailleurs un mécanisme inspiré de l'outil QuickCheck[49] pour générer des valeurs de variables : les générateurs sont paramétrables et même programmables de sorte que le testeur peut définir une stratégie pour la couverture des types de données présents dans les messages. Par défaut, une stratégie de construction de données aléatoires est implantée avec une borne maximale sur le nombre d'éléments que peut produire un générateur.

Résultat

Un simple patron de conception *Observateur* permet de recevoir la notification des événements produits par le contrôleur, essentiellement l'envoi/réception de messages et l'atteinte d'un ou plusieurs objectifs

de test. Dans l'interface graphique de l'outil, ce mécanisme permet de visualiser sur une représentation graphique des automates FIDL les transitions et états déclenchés au cours du processus de test.

On peut aussi, dans un mode de fonctionnement en tâche de fond, utiliser un observateur interfacé avec JUnit ce qui permet d'intégrer les résultats produits par le contrôleur dans un rapport de tests plus général (voir ci-dessous).

8.3 Modèles et processus de développement

Nous avons dans le premier chapitre esquissé une chaîne de production d'applications basée sur une architecture de composants, chaîne qui se trouve concrètement représentée par différentes *vues* et implantée dans différents processus de manipulation des artefacts produits par les équipes de développement. La figure 1.5 résume les principales étapes de cette chaîne de production et nous proposons dans cette section quelques solutions pour intégrer la plate-forme FIDL décrite succinctement à la section précédente à cette chaîne.

Dans la gestion des différentes phases et outils du développement de logiciels, on peut grossièrement distinguer deux approches :

– l'approche *monolithique* basée sur l'utilisation par l'ensemble des acteurs du développement d'une plate-forme unique sensée intégrer l'ensemble des outils et procédures — « bonnes pratiques » — nécessaire à la production de logiciels ;
– l'approche *répartie* dans laquelle chaque acteur utilise des outils et techniques spécifiquement adaptées à sa ou ses tâches, l'ensemble étant synchronisé par une gestion des processus et des artefacts.

La première approche s'incarne dans des plate-formes de développement intégrées ou IDE telles que *Visual Studio* ou *Eclipse*, la seconde dans des outils variés comme make, Maven ou Ant qui permettent de contrôler les étapes de la construction d'application par *scriptage*, ou encore Continuum ou CruiseControl qui assurent l'intégration continue des développements et le déploiement automatisé des différentes versions.

Nous avons choisi de privilégier cette seconde voie en construisant un ensemble de composants ouverts de manipulation des modèles FIDL : analyse syntaxique, transformation de modèles, vérification de modèles, validations statiques et dynamiques. La figure 8.5 synthétise les différentes utilisations principales d'un modèle architectural de composants FIDL au cours du processus de développement.

En poursuivant l'analogie avec la chaîne de production, il est nécessaire de disposer, entre chaque étape du processus, de phases de contrôle de la qualité des éléments produits. L'utilisation de modèles formels a pour but de faciliter l'automatisation de ces contrôles aux différentes étapes et de renforcer la cohérence des résultats obtenus.

Nous supposons que l'on dispose au démarrage du processus d'un modèle architectural de composants FIDL suffisamment détaillé. La construction de ce modèle ne fait pas partie du processus de construction du logiciel, même s'il est évident que le modèle peut être affiné en cours de développement, voire modifié si le besoin s'en fait sentir : le développement n'est pas un processus en cascade dans lequel l'information circule à sens unique, mais un processus dialectique supposé mener à un point fixe qui est la mise en production de l'application. Nous pouvons toutefois remarquer que le fait de disposer d'un modèle *exécutable* permet de valider la construction d'un tel modèle par *simulation* : il suffit de définir un *déployeur* spécifique qui va simuler un composant réel en utilisant les même outils que le contrôleur et l'exécuteur.

8.3.1 Conception

L'étape de conception commence quand le modèle FIDL ou un fragment opérationnel de celui-ci a été vérifié syntaxiquement et sémantiquement (voir section 8.1.1). Une partie du modèle de conception peut être produite par transformation de modèle à partir du schéma d'architecture ou d'une partie de celui-ci. Ce modèle de conception est évidemment enrichi et détaillé.

Les mêmes règles de transformations entre le langage FIDL et le langage de conception doivent permettre de réaliser une validation statique telle que définie à la section 8.1.2 : soit par simple vérification de la structure de conception, soit de manière plus approfondie par vérification du typage comportemental.

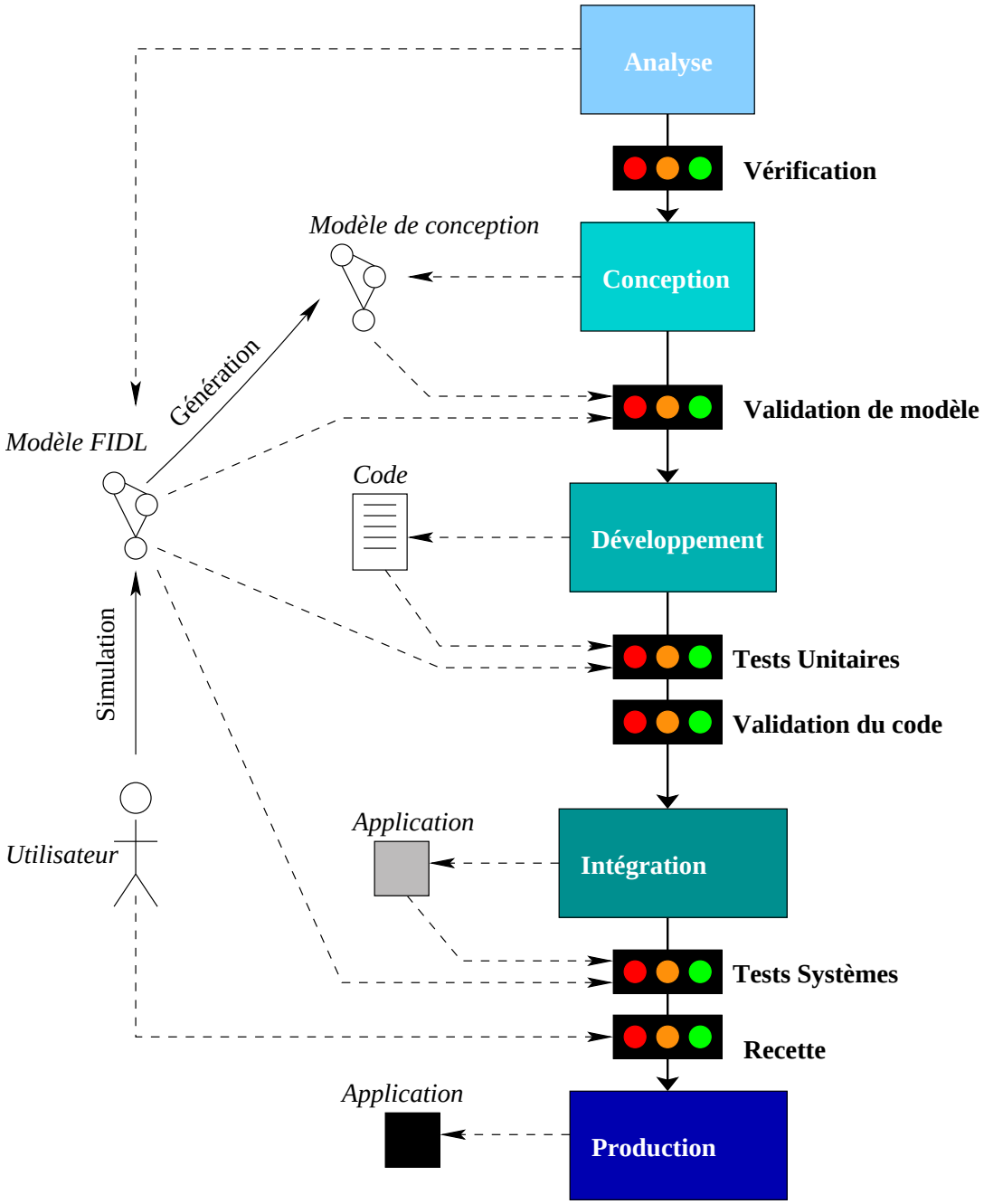


FIG. 8.5 – Processus de développement & Contrôles qualités.

8.3.2 Développement

Le code produit à cette étape va être validé par le processus de test décrit dans les sections précédentes et qui constitue le cœur de l'atelier FIDL.

Les tests à partir de modèles FIDL peuvent être réalisés selon deux modalités :

– *traditionnelle* : un certain nombre de cas de test sont générés, puis ensuite exécutés en tant que cas de test JUnit ;
– *dynamique* : un testeur implantant l'interface TestRunner de JUnit exécute des tests de manière adaptative en fonction des réponses reçues depuis l'IUT et selon les stratégies définies dans le chapitre 7.

Dans ce second cas de figure, l'intégration avec JUnit se fait en considérant que toute opération de `reset` signale une fin de test unitaire réussie qui est donc indiquée comme telle au framework JUnit.

Le test à partir d'un modèle FIDL peut être évidemment couplé avec une analyse de la couverture du code du composant réel lorsque celui-ci est disponible localement. Cette information est extrêmement intéressante puisqu'elle permet de savoir quelle fraction du code développé est couverte par la spécification modélisée ou de manière équivalente quelle partie de la spécification est réellement exécutable dans le code.

Comme dans le cas de la conception, si l'on dispose de règles de « traduction » du langage FIDL vers le langage de développement, on peut aussi réaliser une validation statique du code source produit.

8.3.3 Intégration

En phase d'intégration, le code source est assemblé et déployé pour produire un système ou un sous-système complet. Le principal intérêt du modèle d'architectures que nous avons proposé est de permettre de tester à partir du même outil et de façon systématique les fonctionnalités d'un ensemble de composants produisant un système par assemblage. Selon les choix faits dans le processus de développement et le degré de précision de la modélisation, les tests pourront être réalisés à différents niveaux.

La validation du processus d'intégration pourra donc comprendre une suite de tests système générée automatiquement à partir du modèle architectural. Cette phase de test automatisée précède la phase de recette dans laquelle le modèle architectural pourra encore jouer un rôle comme validation dynamique (voir section 8.1.3).

8.3.4 Production

Le passage en production de l'application signe la fin du processus de développement. Ce processus pourra éventuellement être redémarré pour prendre en compte des évolutions et des anomalies. L'analyse précise des anomalies relevées en production et la capacité à *tracer* l'origine de ces anomalies grâce au lien existant entre toutes les étapes de production. Le modèle architectural initial doit faciliter la mise en œuvre de ces corrections et évolutions.

Conclusion

Contributions

La première et principale contribution de ce travail est un langage formel de modélisation d'architectures à base de composants, baptisé FIDL. Ce langage et le modèle sous-jacent ont été décrits dans les chapitres 4, 5 et 6 sur le plan syntaxique comme sur le plan sémantique. Ce langage permet de décrire et de construire un système comme une hiérarchie de *composants* autonomes, reliés au travers d'*interfaces* décrivant des services sous la forme de *messages*. Les interfaces sont l'unité de base du langage fournissant la notion de *contrat comportemental* dont la sémantique est définie comme un ensemble de traces, un langage clos par préfixe.

Les automates FIDL décrits au chapitre 5 sont les machines reconnaissant les langages et leur expressivité dépend *in fine* de la richesse des types de messages disponibles et de la puissance du langage contraignant le contenu de ces messages. Nous avons vu comment, en posant certaines restrictions, on pouvait obtenir un modèle d'automate qui soit effectivement calculable.

Le principal intérêt de la sémantique formelle des contrats d'interfaces et de composants est d'être *compositionnelle* : l'asssemblage de deux ou plusieurs composants contractuellement fiables, sous l'hypothèse qu'ils ne forment pas de cycles de dépendance, est lui-même contractuellement fiable, une propriété démontrée au chapitre 6. Nous avons en passant défini formellement une notion de *relation contractuelle* entre langages qui par ailleurs nous aura permis de préciser ce que l'on entend par *sous-typage comportemental*.

Nous nous sommes ensuite intéressés à la vérification de modèles FIDL vis-à-vis de la fiabilité contractuelle et de l'indépendance des facettes, vérification qui permet de s'assurer de la cohérence d'un modèle et de valider une décomposition. L'intérêt de la compositionnalité du modèle FIDL est pleinement apparue lorsque nous nous sommes intéressé à l'utilisation d'un tel modèle comme outil de validation de composants réels par la technique du test. À partir des travaux antérieurs sur le test d'automates d'entrée-sortie, nous avons défini un processus de test unitaire permettant de construire de manière automatisée des testeurs à partir de spécifications FIDL. Nous avons montré que le processus de test pouvait être décomposé et se ramener au test de chacun des automates FIDL composant la spécification du composant et de ses interfaces, ce qui permet de simplifier notablement ce processus et évite en particulier le calcul du produit de synchronisation de l'ensemble des automates.

Nous avons enfin dans le dernier chapitre, décrit différentes étapes de mise en œuvre de ces outils théoriques. D'une part, en détaillant la structure et le fonctionnement d'un prototype réel de test de composants écrits dans le langage Java ; d'autre part en montrant comment un modèle architectural FIDL pouvait fournir des outils de contrôle de la qualité des développements dans les différentes phases de la chaîne de production. Ces contributions ont donné lieu à diverses publications [15, 16, 138, 147] et à la réalisation d'un prototype (`http ://www.achilleus.net/fidl/`).

Perspectives

Quand j'ai démarré ce travail, les composants étaient à l'avant-garde de la recherche dans le *génie logiciel*. Comme toutes les avant-gardes, elle s'est trouvée rapidement dépassée et aujourd'hui l'*ingénierie des modèles* est la nouvelle avant-garde du génie logiciel, les composants se trouvant relégués dans la soute comme un détail technique et une plateforme d'implantation.

De mon point de vue, toutefois, les composants ne se réduisent pas à un choix technologique mais constituent le cœur d'une démarche d'analyse, de conception et de production d'applications centrée sur la notion d'architecture de composants. Les plateformes à composants, de par leur complexité, constitueraient plutôt un frein à l'utilisation de ces concepts comme outil de modélisation. Une première perspective de travail relativement proche s'offre donc à nous pour intégrer cette vision architecturale dans le flot de l'évolution vers une ingénierie dirigée par les modèles.

Modèle

Le modèle de composant que nous avons développé est limité et un certain nombre de situations concrètes ne sont pas prises en compte. La première extension significative devrait concerner l'utilisation d'un même port dans plusieurs connexions, ce que nous avons nommé des ports *multiple* ou *single*. La notion de *receptacle multiple* est prévue dans le CCM mais sa sémantique est floue. La notion de *facette multiple* permettrait d'intégrer dans le modèle le concept de session d'interaction entre composants. Nous avons déjà noté que l'introduction de ports multiples ne poserait pas de gros problèmes : *grosso modo*, il faudrait étendre la définition de l'ensemble de traces pour qu'elle prît en compte plusieurs instances des traces contenant une facette ou réceptacle multiple. Les ports *single* sont bien évidemment plus problématiques.

La deuxième modification d'importance qui devrait être faite sur le modèle serait l'introduction de connexions dynamiques : la possibilité que les messages transportent des *références* de ports. Ce qui impliquerait donc l'utilisation de variables typées par des interfaces. Une possibilité intéressante serait de typer les variables de port dans une expression FIDL par typage comportemental, ce qui permettrait de vérifier statiquement la validité des « connexions dynamiques » réalisées par liaison d'une variable représentant un port. Cette approche est similaire aux notions de types dans les calculs de processus.

En ce qui concerne les assemblages, nous avons d'ores et déjà esquissé quelques pistes qui permettraient de relâcher les contraintes fortes, notamment l'acyclicité des connexions entre composants, existant dans le modèle. Suivant les travaux de Charpentier [46], il serait aussi intéressant de caractériser des classes de propriétés préservées par la composition de composants, de manière à offrir à l'utilisateur la possibilité de définir ses propres stratégies de vérification.

Langages & Automates

Sur le plan théorique des langages et automates, des questions intéressantes se posent aussi. Des travaux récents dans le domaine du contrôle de modèles s'intéressent à des problèmes de plus en plus complexes, mettant en jeu des domaines de valeurs infinis mais représentables sous forme de langages rationnels (voir section 5.3.2). Il serait intéressant de pouvoir intégrer ces travaux dans notre modèle pour accroître le domaine couvert par les processus de vérification et de validation. Par ailleurs, le fait que le produit de mixage « préserve » la relation contractuelle offre une technique simple pour réduire la complexité d'analyse d'un automate. Suivant les idées de Berstel *et al.* [29], il serait intéressant d'étudier les propriétés de décomposition de langages de traces, par exemple pour analyser ou transcrire du code légataire en en extrayant son architecture.

Test

Nous avons souligné que l'un des problèmes principaux auxquels était confronté le testeur était celui de la *sélection* des cas de tests ou plus généralement de la définition d'un *objectif de test* et de l'interprétation des résultats. Étant entendu que l'objectif général est d'accroître le nombre de défauts détectés, la question se pose donc du pouvoir de détection des différentes stratégies de test, autrement dit de l'accroissement de *fiabilité* induit par l'exécution d'un ensemble de tests donné. Cette évaluation ne peut être faite en général qu'en termes statistiques et c'est toute la problématique de l'ingénierie de la fiabilité de produire et valider des modèles fiables de l'évolution de la fiabilité des logiciels. Si des travaux existent dans le domaine du test dit statistique, basé sur un échantillonage de l'espace des « entrées » selon certains critères, il nous paraîtrait intéressant d'étudier la fiabilité produite par une stratégie de tests unitaires systématiques.

Application

Sur le plan pratique, une étape indispensable consistera à valider expérimentalement l'applicabilité des idées et outils que nous avons pu proposer dans cette thèse. Si la notion d'architecture et de composants est séduisante « sur le papier », il n'existe pas à notre connaissance d'étude sur les avantages réels de la mise en œuvre de telle ou telle méthodologie, et plus particulièrement l'impact sur la fiabilité des logiciels des développements orientés objets et composants. Empiriquement, on ne peut que constater, comme nous l'avons fait dans le premier chapitre, que les résultats ne sont pas à la hauteur du discours marketing. Il faudra donc nous atteler à finaliser le prototype puis à construire un plan d'expérience susceptible de valider scientifiquement les apports en termes de fiabilité d'une architecture à base de composants, étude qui ne pourra s'inscrire que dans le moyen terme.

Bibliographie

[1] Samsom Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2–3) :225–241, 1987.

[2] Samsom Abramsky, Dan R. Ghica, Andrzej S. Murawski, et C.-H. Luke Ong. Algorithmic Game Semantics and Component-Based Verification. In Mike Barnett, Steve Edwards, Dimitra Giannakopoulou, , et Gary T. Leavens, editors, *SAVCBS 2003 - Specification and Verification of Component-Based Systems*, number 03-11 in TR, pages 66–73, September 2003.

[3] Samson Abramsky. Algorithmic game semantics. Marktoberdorf Summer School, 2001. URL `citeseer.ist.psu.edu/505714.html`.

[4] Franz Achermann et Oscar Nierstrasz. *Software Architectures and Component Technology*, chapter Applications = Components + Scripts - A tour of *Piccola*. Kluwer, 2001.

[5] Alfred V. Aho, Anton T. Dahbura, David Lee, et M. Ümit Uyar. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Transactions on Communications*, 39(11) :1604–1616, November 1991.

[6] A.Hoffmann, A. Rennoch, I. Schubert, et A. Vouffo-Feudjio. CCM testing environment. In *Proceedings of ICSSEA 2002*, CNAM, Paris, December 2002.

[7] Jonathan Aldrich, Craig Chambers, et David Notkin. Archjava : Connecting software architecture to implementation. In *Proceedings of the International Conference on Software Engineering2002*, Orlando, FL, USA, May 2002. Association for Computing Machinery.

[8] Jonathan Aldrich, Craig Chambers, et David Notkin. Architectural reasoning in archjava. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2002)*, June 2002.

[9] Jadranka Alilovic-Curgus. Analytic methods in coverage testing of communications software. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, CASCON'92*, pages 301–312. IBM Press, 1992.

[10] Robert Allen et David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, July 1997.

[11] Deepak Alur, John Crupi, et Dan Malks. *Core J2EE Patterns : Best Practices and Design Strategies*. Pearson Education, 2001. Patterns catalog available at `http ://java.sun.com/blueprints/corej2eepatterns/index.html`.

[12] Farhat Arbab. Abstract behavior types : a foundation model for components and their composition. Technical Report SEN-R0305, Centrum voor Wiskunde en Informatica, Amsterdam, NL, July 2003.

[13] Farhat Arbab et Jan Rutten. A coinductive calculus of component connectors. Technical Report SEN-R0216, Centrum voor Wiskunde en Informatica, Amsterdam, NL, September 2002.

[14] Gilles Ardourel, Pierre Crescenzo, et Philippe Lahire. Lamp : vers un langage de définition de mécanismes de protection pour les langages de programmation à objets. In Briot et Malenfant [39], pages 151–163.

[15] Arnaud Bailly, Isabelle Ryl, et Mireille Clerbout. FIDL - spécifications formelles en IDL3. In Briot et Malenfant [39], pages 213–225.

[16] Arnaud Bailly, Mireille Clerbout, et Isabelle Simplot-Ryl. Component composition preserving behavioural contracts based on communication traces. In *Proceedings of the Conference on Implementation and Application of Automata (CIAA'05)*, Sophia-Antipolis, France, June 2005.

[17] Stéphane Barbey, Didier Buchs, et Cécile Péraire. A Theory of Specification-Based Testing for Object-Oriented Software. In *Proceedings of EDCC2 (European Dependable Computing Conference)*, volume 1150 of *Lecture Notes in Computer Science*, pages 303–320, Taormina, Italy, October 1996.

[18] Stéphane Barbey, Didier Buchs, Marie-Claude Gaudel, Bruno Marre, Cécile Péraire, Pascale Thévenod-Fosse, et Hélène Waeselynck. From requirements to tests via object-oriented design. Technical Report 98476, LAAS, Novembre 1998.

[19] Luís Barbosa. Components as Processes : An Exercise in Coalgebraic Modelling. In S. F. Smith et C. L. Talcott, editors, *Proceedings of the 4^{th} Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems, Stanford, USA*, volume 177, pages 397–417, Boston, 2000. Kluwer Academic Publishers. URL `citeseer.ist.psu.edu/barbosa00components.html`.

[20] Luís Barbosa et Sun Meng. Generic components. In *Proceedings of the First APPSEM-II Workshop*, Nottingham, 2003.

[21] Mike Barnett et Wolfram Schulte. The ABCs of specification : AsmL, behavior, and components. *Informatica*, 25(4) :517–526, Nov. 2001.

[22] Roman Barták. Constraint Programming : In Pursuit of the Holy Grail. In *Proceedings of the Week of Doctoral Students (WDS99), Part IV*, pages 555–564, Prague, June 1999. MatFyzPress.

[23] H. Batteram, W. Hellenthal, W. Romijn, A. Hoffmann, A. Rennoch, et A. Vouffo. Implementation of an Open Source Toolset for CCM Components and Systems Testing. In Roland Groz et Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. ISBN 3-540-21219-1.

[24] Benoît Baudry. *Assemblage testable et validation de composants*. PhD thesis, Université de Rennes 1, June 2003.

[25] Boris Beizer. *Black-Box Testing*. John Wiley & Sons , Ltd., 1995.

[26] Gilles Bernot, Marie-Claude Gaudel, et Bruno Marre. Software testing based on formal specifications : a theory and a tool. *Sofware Engineering Journal*, 6(6) :387–405, November 1991. URL `docs/sej91.ps.gz`.

[27] Gilles Bernot, Laurent Bouaziz, et Pascale LeGall. A theory of probabilistic functional testing. In *Proceedings of the 19^{th} International Conference on Software Engineering*, pages 216–227. ACM Press, may 1997.

[28] Jean Berstel. *Transductions and Context-Free Languages*. Teubner Studienbücher Informatik. B.G. Teubner, Stuttgart, 1979.

[29] Jean Berstel, Luc Boasson, et Michel Latteux. Mixed languages. *Theoretical Computer Science*, 332(1–3) :179–198, 2005.

[30] Robert V. Binder. *Testing Object-Oriented Systems - Models, Patterns and Tools*. Object Technology. Addison-Wesley, 1999.

[31] Andreas Blass, Yuri Gurevich, Lev Nachmanson, et Margus Veanes. Play to Test. Technical Report MSR-TR-2005-04, Microsoft Research, Redmond, WA, January 2005.

[32] B. Boigelot et P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. *Formal Methods in System Design*, 14 :237–255, 1999.

[33] Chourouk Bourhfir, Rachida Dssouli, et El Mostapha Aboulhamid. Automatic test generation for efsm-based systems. URL `citeseer.nj.nec.com/114451.html`.

[34] Robert S. Boyer. SELECT–A Formal System for Testing and Debugging Programs. In IEEE Press, editor, *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

[35] A. Bracciali, A. Brogi, et C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering*, 14(1) :45–54, 2004. (in press). A preliminary version of this paper was published in *Component deployment*, Lecture Notes in Computer Science 2370, pages 185–199. Springer, 2002.

[36] Lionel Briand et Yvan Labiche. A uml-based approach to system testing. *Journal of Software and Systems Modeling*, 1(1) :10–42, 2002.

[37] E. Brinksma, L. Heerink, et J. Tretmans. Factorized Test Generation for Multi Input/Output Transition Systems. In Alexandre Petrenko et Nina Yevtushenko, editors, *IWTCS*, volume 131 of *IFIP Conference Proceedings*. Kluwer, 1998. ISBN 0-412-84430-3.

[38] Ed Brinksma et Jan Tretmans. Testing Transition Systems : An Annotated Bibliography. In *Proceedings of the MOVEP 2000 school*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195, Nantes, France, 2001. Springer-Verlag.

[39] Jean-Pierre Briot et Jacques Malenfant, editors. *Langages et Modèles à Objets*, volume 9 of *L'Objet*, Janvier 2003. Hermes - Lavoisier.

[40] S.D. Brookes, C.A.R. Hoare, et A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3) :560–599, July 1984.

[41] L. Cacciari et O. Rafiq. Controllability and observability in distributed testing. *Information and Software Technology*, 41(11–12) :167–780, September 1999.

[42] Carlos Canal, Ernesto Pimentel, et José M. Troya. Conformance and refinement of behavior in π-calculus, 1999.

[43] Carlos Canal, Ernesto Pimentel, et José M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2) :105–138, October 2001.

[44] Ana R. Cavalli, Bruno Defude, Christian Rinderknecht, et Fatiha Zaïdi. A service-component testing method and a suitable corba architecture. In *ISCC*, pages 655–660, Hammamet, Tunisia, July 2001. IEEE Computer Society. ISBN 0-7695-1177-5.

[45] C.Canal, L. Fuentes, J.M. Troya, et A. Vallecillo. Extending corba interfaces with π-calculus for protocol compatibilty. In *Proceedings of the Conference on Object-Oriented Programming, Systems and Language 2000*, pages 208–225. OOPSLA, 2000.

[46] Michel Charpentier. Composing Invariants. In Keijiro Araki, Stefania Gnesi, et Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 401–421. Springer, 2003. ISBN 3-540-40828-2.

[47] Yoonsik Cheon et Gary T. Leavens. A Simple and Practical Approach to Unit Testing : The JML and JUnit way. Technical Report 01-12, Department of Computer Science, Iowa State University, November 2001.

[48] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *Transactions on Software Engineering*, 4(3) :178–187, May 1978.

[49] Koen Claessen et John Hughes. QuickCheck : a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN ICFP 2000*, 35(9) :268–279, 2000. URL `citeseer.ist.psu.edu/claessen99quickcheck.html`.

[50] D. Clarke, T. Jéron, V. Rusu, et E. Zinovieva. Stg : a symbolic test generation tool. In *(Tool paper) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*. Springer-Verlag, 2002.

[51] Richard A. DeMillo, Richard J. Lipton, et Frederick G. Sayward. Hints on Test Data Selection : Help for the Practicing Programmer. *IEEE Computer*, pages 34–41, April 1978.

[52] François Denis, Aurélien Lemay, et Alain Terlutte. Residual Finite State Automata. In Afonso Ferreira et Horst Reichel, editors, *STACS*, volume 2010 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2001. ISBN 3-540-41695-1.

[53] A. Denise, M.-C. Gaudel, et S.-D. Gouraud. A Generic Method for Statistical Testing. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 25–34, Saint-Malo, France, November 2004. IEEE.

[54] C. Duboc. *Commutations dans les Monoides libres : un Cadre Théorique pour l'Étude du Parallélisme*. PhD thesis, Université de Rouen, France, 1986.

[55] E.Bruneton, T.Coupaye, et J.B. Stefani. The Fractal Component Model. Technical report, ObjectWeb Consortium, February 2004.

[56] Loe M. G. Feijs, Nicolae Goga, Sjouke Mauw, et Jan Tretmans. Test selection, trace distance and heuristics. In Ina Schieferdecker, Hartmut König, et Adam Wolisz, editors, *TestCom*, volume 210 of *IFIP Conference Proceedings*, pages 267–282, Berlin, Germany, March 2002. Kluwer. ISBN 0-7923-7695-1.

[57] Peter H. Feiler, Bruce Lewis, et Steve Vestal. The sae architecture analysis & design language (aadl) standard : A basis for model-based architecture driven embedded systems engineering. In *Proceedings of the RTAS 2003 Worshop on Model-Driven Embedded Systems*, Washington, D.C., May 2003. IEEE.

[58] Arnaud Fontaine. Génération d'un outil de déploiement fidl-openccm. Rapport de stage, 09 2003.

[59] Phyllis G. Frankl et Elaine J. Weyuker. A Formal Analysis of the Fault Detecting Ability of Testing Methods. *Transactions on Software Engineering*, 19(3) :202–213, March 1993.

[60] Roy S. Freedman. Testability of software components. *Transactions on Software Engineering*, 17 (6), June 1991.

[61] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, et Abderrazak Ghedamsi. Test selection based on finite state models. *Transactions on Software Engineering*, 17(6) : 591–603, 1991. ISSN 0098-5589.

[62] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[63] Stephen J. Garland et Nancy A. Lynch. Using I/O automata for developing distributed systems. In Leavens et Sitaraman [85], pages 285–312.

[64] Marie-Claude Gaudel et Perry R. James. Testing Algebraic Data Types and Processes : a Unifying Theory. *Formal Aspects of Computing*, 10(5–6) :436–451, 1999.

[65] Dan R. Ghica. *A Games-based Foundation for Compositional Software Model-Checking*. PhD thesis, Oxford University Computing Laboratory, November 2002.

[66] John B. Goodenough et Susan L. Gerhart. Toward a Theory of Test Data Selection. *Transactions on Software Engineering*, 1(2) :156–184, June 1975.

[67] A. Gotlieb, B. Botella, et M. Rueher. A CLP Framework for Computing Structural Test Data. In *Proceedings of the First International Conference on Computational Logic (CL'2000)*, London, UK, 2000. Imperial College.

[68] Yuri Gurevich, Benjamin Rossman, et Wolfram Schulte. Semantic essence of asml. Technical Report MSR-TR-2004-27, Microsoft Research, 2004.

[69] Jean Hartmann, Claudio Imoberdorf, et Michael Meisinger. UML-Based Integration Testing. In *Proceedings of ISSTA'00*, pages 60–70, Portland, Oregon, 2000.

[70] L. Heerink et J. Tretmans. Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In Togashi *et al.* [154], pages 23–38. ISBN 0-412-82060-9.

[71] Robert M. Hierons et H. Ural. Synchronized checking sequences based on uio sequences. *Information and Software Technology*, 45(12) :793–803, 2003.

[72] Wai-Ming Ho, François Pennaneac'h, et Nöel Plouzeau. UMLAUT : A Framework for Weaving UML-based Aspect-Oriented Design. In *Proceedings of the Technology of object-oriented languages and systems Conference (TOOLS 2000)*, volume 33, pages 324–334. IEEE Computer Society, June 2000.

[73] Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae, et Hasan Ural. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability*, 10 :203–227, 2000.

[74] Atsushi Igarashi, Benjamin Pierce, et Philip Wadler. Featherweight Java - A Minimal Core Calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems and Language 1999*, Denver, November 1999.

[75] J. C. Fernandez, C. Jard, T. Jéron, et G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur et Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 348–359, New Brunswick, NJ, USA, 1996. Springer–Verlag. URL `citeseer.ist.psu.edu/fernandez96using.html`.

[76] Bart Jacobs et Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, (62) : 222–259, 1997.

[77] Claude Jard. Synthesis of distributed testers from true-concurrency models of reactive systems. *Information and Software Technology*, 45(12) :805–814, September 2003.

[78] T. Jeron, J.-M. Jezequel, Y. Le Traon, et P. Morel. Efficient strategies for integration and regression testing of oo systems. In *Proceedings of the* 10^{th} *International Symposium on Software Reliability Engineering (ISSRE 99)*, pages 260–269, November 1999.

[79] Antti Kervinen et Pablo Virolainen. Heuristics for Faster Error Detection With Automated Black Box Testing. In *Proceedings of the Workshop on Model Based Testing (MBT 2004)*, volume 111 of *Electronic Notes in Theoretical Computer Science*, pages 53–71. Elsevier, January 2005.

[80] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, et Mark Utting. Boundary Coverage Criteria for Test Generation from Formal Models. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 139–150, Saint-Malo, France, November 2004. IEEE.

[81] Vipin Kumar. Algorithms for Constraint Satisfaction : A Survey. *AI Magazine*, 13(1) :32–44, 1992.

[82] R. Lai. A survey of communication protocol testing. *J. Syst. Softw.*, 62(1) :21–46, 2002. ISSN 0164-1212. doi : http ://dx.doi.org/10.1016/S0164-1212(01)00132-7.

[83] Dick Lantin. *.NET*. Eyrolles, Paris, France, Octobre 2003.

[84] Pascale Le Gall et Agnès Arnould. Formal specifications and test : correcteness and oracle. In *Proceedings of COMPASS/ADTT 1995*, volume 1130 of *Lecture Notes in Computer Science*, pages 342–358. Springer-Verlag, 1996. URL `docs/formal-specifications-and-test.ps.gz`.

[85] Gary T. Leavens et Murali Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[86] Gary T. Leavens, Albert L. Baker, et Clyde Ruby. Prelimary Design of JML. Technical Report 98-06p, Department of Computer Science, Iowa State University, Ames, Iowa, August 2001.

[87] D. Lee et M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[88] David Lee, Krishan K. Sabnani, David M. Kristol, et Sanjoy Paul. Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach. *IEEE Transactions on Communications*, 44(5) :631–640, May 1996.

[89] Grégory Lestiennes et Marie-Claude Gaudel. Testing Processes from Formal Specifications with Inputs, Outputs and Data Types. In *Proceedings of the 13^{th}Intl. Symp. on Software Reliability Engineering*, pages 3–14, Annapolis, Maryland, November 2002. IEEE Computer Society. URL `docs/testing-process-from-formal.pdf`.

[90] Barbara H. Liskov et Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6) :1811–1841, November 1994. URL `http ://citeseer.nj.nec.com/liskov94behavioral.html`.

[91] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, et Walter Mann. Specification and analysis of system architecture using rapide. In *Transactions on Software Engineering*, volume 21, pages 336–355, April 1995.

[92] Markus Lumpe, Franz Achermann, et Oscar Nierstrasz. A formal language for composition. In Leavens et Sitaraman [85], pages 69–90.

[93] Gang Luo, Rachida Dssouli, Gregor v. Bochmann, Pallapa Venkataram, et Abderrazak Ghedamsi. Test generation with respect to distributed interfaces. *Computer Standards & Interfaces*, 16(2) : 119–132, June 1994.

[94] Gang Luo, G. von Bochmann, et A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *Transactions on Software Engineering*, 20(2) :149–162, 1994. ISSN 0098-5589.

[95] Nancy A. Lynch et Mark R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2 (3) :219–246, September 1989.

[96] M.R. Lyu, editor. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., and McGraw-Hill, New York, 1996.

[97] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer, New-York, NY, second edition, 1997.

[98] J. Magee, N. Dulay, S. Eisenbach, et J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC 95)*, Sitges, Spain, September 1995.

[99] Bruno Marre. *Une méthode et un outil d'assistance à la sélection de jeux de tests à partir de spécifications algébriques*. PhD thesis, Université de Paris-Sud - Orsay, 1991.

[100] Hugues Martin. *Une méthodologie de génération automatique de suites de tests pour applets Java Card*. PhD thesis, Université de Lille I, 2001.

[101] Raphaèl Marvie et Philippe Merle. Vers une modèle de composants pour CESURE. Rapport intermédiaire, projet RNRT CESURE, Novembre 2000.

[102] Thomas J. McCabe et Charles W. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12) :1415–1425, 1989. ISSN 0001-0782. doi : http ://doi.acm.org/10.1145/76380.76382.

[103] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[104] Phil McMinn. Search-based software test data generation : a survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, June 2004.

[105] Nenad Medvidovic et Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[106] Nikunj R. Mehta, Marjan Sirjani, et Farhat Arbab. Effective modelling of software architectural assemblies using constraint automata. Technical Report SEN-R0309, Centrum voor Wiskunde en Informatica, Amsterdam, NL, October 2003.

[107] Sun Meng et Berhard K. Aichernig. Coalg$_{KPF}$: Towards a coalgebraic calculus for component-based systems. Technical Report 271, UNU/IIST, Macau, PRC, January 2003.

[108] Sun Meng, Bernhard K. Aichernig, Luís S. BArbosa, et Zhang Naixiao. A Coalgebraic Semantic Framework for Component-based Development in UML. In L. Birkedal, editor, *Proceedings of the 10th Conference on Category Theory in Computer Science (CTCS 2004)*, volume 122 of *Electronic Notes in Theoretical Computer Science*, pages 229–245. Elsevier, March 2005.

[109] Bertrand Meyer. *Conception et Programmation par Objets*. IIA. InterEditions, Paris, 1990.

[110] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 1997.

[111] Zbigniew Michalewicz et David B. Fogel. *How to Solve It : Modern Heuristics*. Springer-Verlag, Berlin, 2000.

[112] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[113] Robin Milner, Joachim Parrow, et David Walker. A calculus of mobile processes, part i, September 1990.

[114] Robin Milner, Joachim Parrow, et David Walker. A calculus of mobile processes, part ii, September 1990.

[115] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153, Princeton, 1956.

[116] Henry Muccini, Antonia Bertolino, et Paola Inverardi. Using Software Architecture for Code Testing. *Transactions on Software Engineering*, 30(3) :160–171, March 2004.

[117] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons , Ltd., 2nd edition, 2004. Revised and updated by Tom Badgett and Todd Thomas, with Corey Sandler.

[118] Clémentine Nebut, Simon Pickin, Yves Le Traon, et Jean-Marc Jzéquel. Automated Requirements-based Generation of Test Cases for Product Families. In *ASE*, pages 263–266. IEEE Computer Society, 2003. ISBN 0-7695-2035-9.

[119] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the Conference on Object-Oriented Programming, Systems and Language 1993*, pages 1–15. Proceedings of the Conference on Object-Oriented Programming, Systems and Language 1993, 1993.

[120] Oscar Nierstrasz, Jean-Guy Schneider, et Markus Lumpe. Formalizing composable software - a research agenda. In *FMOODS'96*, pages 271–282. Chapmann and Hall, 1996.

[121] Simeon Ntafos. On comparisons of random, partition, and proportional partition testing. *Transactions on Software Engineering*, 27(10) :949–960, October 2001.

[122] Object Management Group. *CORBA 3.0 Components*. Number ptc/2001-11-03. Object Management Group, November 2001.

[123] Object Management Group. The Common Object Request Broker : Architecture and Specification. Technical Report formal/02-06-33, Object Management Group, July 2002.

[124] A. Jefferson Offutt. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Programming Languages and Systems*, 1(1) :3–18, January 1992.

[125] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, et Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1) :25–53, March 2003.

[126] Jefferson Offutt et Aynur Abdurazik. Generating Tests from UML Specifications. unpublished report ?

[127] OMG. *UML 2.0 Superstructure*, chapter Components, pages 133–150. Number ptc/03-08-02. 2004.

[128] Roy P. Pargas, Mary Jean Harrold, et Robert R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability*, 9 :263–282, 1999.

[129] Cécile Péraire, Stéphane Barbey, et Didier Buchs. Test Selection for Object-Oriented Software Based on Formal Specifications. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 385–403, Shelter Island, New York, USA, June 1998. Chapman & Hall.

[130] Alexandre Petrenko, Nina Yevtushenko, et Jia Le Huo. Testing Transition Systems with Input and Output Testers. In *Proceedings of the 15^{th} IFIP Conf. Testing of Communication Systems - TestCom'2003*, Sophia Antipolis, FR, 2003.

[131] Marc Phalippou. *Relations d'implantation et hypothèses de tes sur des automates à entrées et sorties*. PhD thesis, Université de Bordeaux I, 9 1994.

[132] Simon Pickin et Jean-Marc Jézéquel. Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. In Eerke A. Boiten, John Derrick, et Graeme Smith, editors, *Integrated Formal Methods : 4th International Conference, IFM 2004*, volume 2999 of *Lecture Notes in Computer Science*, pages 481–500, Canterbury, UK, April 2004. Springer-Verlag - Heidelberg.

[133] Frantisek Plasil et Stanislav Visnovsky. Behavior Protocols for Software Components. *Transactions on Software Engineering*, 28(11) :1056–1076, November 2002.

[134] Tuomo Pyhälä. Specification-based test selection in formal conformance testing. Master's thesis, Helsinki University of Technology, August 2004.

[135] Tuomo Pyhälä et Keijo Heljanko. Specification coverage aided test selection. In Johan Lilius, Felice Balarin, et Ricardo J. Machado, editors, *Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'2003)*, pages 187–195, Guimaraes, Portugal, June 2003. IEEE Computer Society. URL `citeseer.ist.psu.edu/596175.html`.

[136] Emmanuel Renaux. *Définition d'une démarche de conception de systèmes à base de composants*. PhD thesis, Université de Lille I, Lille, France, décembre 2004.

[137] Emmanuel Renaux, Olivier Caron, et Jean-Marc Geib. Chaîne de production de systèmes à base de composants logiques. In *Langages et Modèles à Objets - LMO'04*, volume 10 of *L'Objet*, pages 147–160. Hermes - Lavoisier, Mars 2004.

[138] Isabelle Ryl, Mireille Clerbout, et A. Bailly. A component oriented notation for behavioural specification and validation. In D. Giannakopoulou, Gary T. Leavens, et M. Sitaraman, editors, *Proceedings of the OOPSLA 2001 Specification and Verification of Component-Based Systems Workshop*, volume Technical Report ISU TR #01-09, Tampa, Florida, 2001. Iowa State University.

[139] Krishan Sabnani et Anton Dahbura. A new technique for generating protocol test. In *Proceedings of the ninth symposium on Data communications, SIGCOMM'85*, pages 36–43, New York, NY, USA, 1985. ACM Press. ISBN 0-89791-164-4. doi : http ://doi.acm.org/10.1145/319056.319003.

[140] Jacques Sakarovitch. *Éléments de théorie des automates*. Vuibert Informatique, Paris, France, septembre 2003.

[141] Behcet Sarikaya et Gregor von Bochmann. Synchronization and specification issues in protocol testing. *Transactions on Communications*, 32(4) :389–395, April 1984.

[142] Alan Schmitt et Jean-Bernard Stefani. The M-Calculus : a higher-order distributed process calculus. Technical Report 4361, INRIA, January 2002.

[143] Alan Schmitt et Jean-Bernard Stefani. The Kell Calculus : A Family of Higher-Order Distributed Process Calculi. In *Proceedings of the Global Computing 2004 workshop*, Lecture Notes in Computer Science, Venice, Italy, January 2004.

[144] Giuseppe Scollo et Silvia Zecchini. Architectural Unit Testing. In Y.Gurevich, A.K. Petrenko, et K.Kossatchev, editors, *Proceedings of the Workshop on Model Based Testing (MBT 2004)*, volume 111 of *Electronic Notes in Theoretical Computer Science*, pages 27–52. Elsevier, January 2005.

[145] Mary Shaw, Robert DeLine, Daniel V/. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Transactions on Software Engineering*, 21(4) :314–335, April 1995.

[146] Deepinder P. Sidhu et Ting-Kau Leung. Formal Methods for Protocol Testing : A Detailed Study. *Transactions on Software Engineering*, 15(4) :413–427, April 1989.

[147] Isabelle Simplot-Ryl, Mireille Clerbout, et Arnaud Bailly. Stac : Communication traces based specifications and tests of software components. In *Proc. of the 15th Nordic Workshop on Programming Theory (NWPT'03)*, Turku, Finland, 2003.

[148] *Enterprise JavaBeans^{TM} Specification 2.1*. Sun Microsystems, November 2003.

[149] *J2EE 1.4 Specification*. Sun Microsystems, November 2003.

[150] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2^{nd} edition, 2002.

[151] Andrem S. Tannenbaum et Maarten van Steen. *Distributed System - Principles and Paradigms*. Prentice-Hall, New Jersey, USA, 2002.

[152] P. Thévenod-Fosse et H. Waeselynck. Statemate applied to statistical software testing. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '93)*, pages 99–109. ACM Press, 1993.

[153] P. Thevenod-Fosse, H. Waeselynck, et Y. Crouzet. An Experimental Study on Software Structural Testing : Deterministic versus Random Input Generation. In *Proceedings of the Twenty-First Annual International Symposium on Fault-Tolerant Computing (FTCS-21)*, page 410, Montreal, Canada, June 1991.

[154] Atsushi Togashi, Tadanori Mizuno, Norio Shiratori, et Teruo Higashino, editors. *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII'97, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII), 18-21 November, 1997, Osaka, Japan*, volume 107 of *IFIP Conference Proceedings*, 1998. Chapman & Hall. ISBN 0-412-82060-9.

[155] J. Tretmans et A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99 :* 7^{th} *European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[156] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. In Tiziana Margaria et Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996. ISBN 3-540-61042-1.

[157] Jan Tretmans. *A formal approach to conformance testing*. PhD thesis, Universiteit Twente, August 1992.

[158] UIT-T. Cadre général des méthodes formelles appliquées aux tests de conformité. Technical Report Z.500, UIT-T, mai 1997.

[159] Andreas Ulrich et Hartmut König. Specification-based testing of concurrent systems. In Togashi *et al.* [154], pages 7–22. ISBN 0-412-82060-9.

[160] Machiel van der Bijl, Arend Rensink, et Jan Tretmans. Compositional testing with ioco. In Alexandre Petrenko et Andreas Ulrich, editors, *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003. ISBN 3-540-20894-1.

[161] R.J. van Glabbeek. The linear time – branching time spectrum I ; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, et S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001. URL `http ://Boole.stanford.edu/pub/DVI/spectrum1.dvi.gz`. Available at `http ://boole.stanford.edu/pub/spectrum1.ps.gz`.

[162] A.M.R. Vincenzi, J.C. Maldonado, W.E. Wong, et M.E. Delamaro. Coverage testing of java programs and components. *Science of Computer Programming*, 56(1-2) :211–230, April 2005.

[163] P. Wolper et B. Boigelot. On the Construction of Automata from Linear Arithmetic Constraints. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

[164] Spyros Xanthakis, Pascal Régnier, et Constantin Karapoulios. *Les test des logiciels*. Hermes, 2000.

[165] Gaoyang Xie et Zhe Dang. Testing systems of concurrent black-boxes - an automata-theoretic and decompositional approach. In Wolfgang Grieskamp et Carsten Weise, editors, *FATES*, Lecture Notes in Computer Science. Springer, 2005.

[166] M. Yannakakis. Testing, optimization, and games. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 78–98. IEEE Press, July 2004.

[167] Fatiha Zaïdi. *Contribution à la génération de tests pour les composants de service. Application aux services de Réseau Intelligent.* PhD thesis, Université Évry - Val d'Essonne, novembre 2001.

[168] Hong Zhu, Patrick A.V. Hall, et John H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4) :366–427, December 1997.

Annexe A

Grammaire FIDL

specification-fidl ::= `FIDL` *comportement*
| `header` *function_headers* `FIDL` *comportement* `body` *function_bodies*

comportement ::= *specification-facettes expression-comportement*
| *expression-comportement*

specification-facettes ::= `facets` *liste-facettes* `;`

liste-facettes ::= *liste-facettes* `,` *facette*
| *facette*

facette ::= `multiple`*port_name*
| `single` *port_name*
| `unique` *port_name*

expression-comportement ::= *expression-comportement* `@@` *expression-elementaire*
| *expression-elementaire*

expression-elementaire ::= *contrainte* `in` *expression-base*
| *expression-base*

contrainte ::= *contrainte* `;` *decl-variable*
| *decl-variable*

decl-variable ::= *variable_name* `:` *expr-contrainte*
| *variable_name* `: :` *type_name*

expression-base ::= *expression-base* `||` *expression-par*
| *expression-par*

expression-par ::= *expression-par* `+` *expression-alt*
| *expression-alt*

expression-alt ::= *expression-alt expression-star*
| *expression-star*

expression-star ::= *expression-atom* `*`
| *expression-atom*

expression-atom ::= *message*
| `(` *expression-elementaire* `)`

message ::= *message_complet* | *message_simplifie*

message_complet ::= *arrow method_name* `(` *parametres-message* `)`
| *port_name arrow method_name* `(` *parametres-message* `)`
| *port_name*`[` *parametres-event* `]`
| `<-`*method_name* `<` *exception* `>`
| *port_name* `<-` *method_name* `<` *exception* `>`

message_simplifie ::= *method_name* `(` *parametres-message* `)`
| *port_name* `.` *method_name* `(` *parametres-message* `)`
| *port_name* `.` *method_name* `<` *exception* `>`
| `void`

parametres-event ::= *liste-expr-message*

	|	ε
parametres-message	::=	*liste-expr-message*
	|	: *expr-message*
	|	*liste-expr-message* : *expr-message*
	|	ε
liste-expr-message	::=	*liste-expr-message* , *expr-message*
	|	*expr-message*
expr-message	::=	*literal*
	|	*variable_name*
	|	_
exception	::=	*exception_name* [*liste-expr-message*]
	|	*exception_name*
arrow	::=	->
	|	<-
expr-contrainte	::=	*variable_name operator expr-fonctionnelle*
expr-fonctionnelle	::=	*variable_name*
	|	*literal*
	|	*function_name* (*liste-parametres*)
	|	*function_name* ( )
	|	*expr-fonctionnelle bin-op expr-fonctionnelle*
	|	*un-op expr-fonctionnelle*
	|	(*expr-fonctionnelle*)
liste-parametres	::=	*liste-parametres*, *parametre-fonction*
	|	*parametre-fonction*
parametre-fonction	::=	*expr-fonctionnelle*
	|	Trace
	|	*message_complet*
bin-op	::=	< | == | > | <= | >= | && | || | ^^ | != | + | - | * | / | %
un-op	::=	!
identifiant	::=	[a-z A-Z] [a-z A-Z 0-9 _]*
operator	::=	< | = | > | <= | >= | !=
literal	::=	*integer-literal*
	|	*string-literal*
	|	*character-literal*
	|	*floating-pt-literal*
	|	{ *literal-list* } | {}
literal-list	::=	*literal-list* , *literal*
	|	*literal*
integer-literal	::=	0 [oO] [0-7]+
	|	0 [xX] [0-9 a-f A-F]+
	|	[0-9]+
floating-pt-literal	::=	[0-9]* .[0-9]+ *exponent-part*
exponent-part	::=	[eE] [0-9]
	|	[eE] [+-] [0-9]
	|	ε
scoped_name	::=	:: *colon_name*
	|	*colon_name*
colon_name	::=	*colon_name* :: *identifiant*
	|	*identifiant*
function_headers	::=	*function_headers* ; *function_header*
	|	*function_header*
function_header	::=	*function_name* : *in_types* → *elementary_type*
	|	*function_name* : → *elementary_type*
in_types	::=	*in_types* , *in_type*

	|	*in_type*
elementary_type	::=	`long` | ...
in_type	::=	*elementary_type* | `Trace` | `Message`
port_name	::=	*identifiant*
variable_name	::=	*identifiant*
method_name	::=	*identifiant*
function_name	::=	*identifiant*
exception_name	::=	*scoped_name*
type_name	::=	*scoped_name*
function_bodies	::=	*selon langage choisi*

Annexe B

Glossaire

$\mathcal{D}$ Univers des valeurs

$\mathcal{V}$ Ensemble dénombrable de variables

ϵ Mot vide : $x.\epsilon = \epsilon.x = x$

$\mathrm{alph}(L)$ Alphabet d'un langage

$\sqcap_{X,Y}$ Produit de synchronisation ou *mixage* sur les alphabets X et Y

Π_X Projection sur un alphabet X

$\sqcup\!\sqcup$ Produit de mélange ou *shuffle*

h_x^y Substitution de x par y

$Pref(L)$ Ensemble des facteurs gauches ou *préfixes* de L

X^* Monoïde libre engendré par X

$\mathcal{P}X$ Ensemble des parties de X ou *powerset*

V&V Vérification et Validation

AADL Architecture Analysis & Design Language

ADL Architecture Description Language

ADO Active Data Objects

API Application Programming Interface ou Interface de Programmation d'Applications. Interface d'un système accessible au programmeur

AsmL Abstract State Machine Language. Langage de spécification orienté objet basé sur la théorie des ASM

BO Business Object ou Objet Métier. Représentation dans l'application d'un ou plusieurs éléments d'un processus correspondant au métier de l'entreprise

CCM Corba Component Model

CIDL Component Implementation Definition Language

CIL Common Intermediate Language

CLI Common Language Infrastructure

CLR Common Language Runtime

CORBA Common Object Request Broker Architecture. Norme d'intergiciel proposée par l'OMG

CSP (1) Communicating Sequential Processes. Formalisme pour l'étude des propriétés des programmes concurrents, page 33

CSP (2) Constraint Satisfaction Problem ou *Problème de Satisfaction de Contraintes*

déverminage Dit aussi deboggage ou *debugging*. Recherche et suppression des erreurs dans le code d'une application

DAO Data Access Objects. Objets permettant de simplifier pour le développeur l'accès au système de stockage des données

Design Patterns Voir *Patrons de conception*

DLL Dynamically Linked Library. Bibliothèque de code dynamiquement liée aux applications exécutables, appelées aussi *Shared Object* dans le monde Unix

Eclipse Plateforme libre de développement générique et extensible, essentiellement tournée vers Java (voir `http ://www.eclipse.org`)

EDOC Enterprise Distribued Object Computing

EFSM Extended Finite State Machine

EJB Enterprise JavaBeans. Élément de la spécification *J2EE* définissant la structure et les fonctionnalités de la partie d'une application répartie contenant les traitements métiers. Ne pas confondre avec les *JavaBeans*

ERP Enterprise Resource Planner. Progiciel de gestion intégré

eXtreme Programming Méthodologie rapide de développement de logiciels centrée sur les interactions avec l'utilisateur final et les tests unitaires et systèmes

FIDL Formal Interface Definition Language

FSM Finite State Machine

Grappe d'objets Ensemble des objets liés les uns aux autres et formant un graphe

Haskell Langage fonctionnel pur à évaluation paresseuse (*cf.* `http ://www.haskell.org`)

HTML HyperText Markup Language. Langage dans lequel sont écrits la plupart des documents accessibles par le protocole HTTP

HTTP HyperText Transfer Protocol ou Protocole de Transfert d'HyperTexte. Protocole de base du *World Wide Web* permettant à un navigateur de communiquer avec un serveur

IDE Integrated Development Environment ou ENVIRONNEMENT DE DÉVELOPPEMENT INTÉGRÉ

IDL Inteface Definition Language

IDL2 Version de base d'IDL sur plateforme CORBA

IDL3 Extension de l'IDL2 incluant la notion de composant

IHM Interface Homme-Machine

intergiciel Logiciel facilitant la mise en œuvre d'applications réparties

IOLTS Input/Output Labelled Transition System

IOTS voir *IOLTS*

ITU International Telecommunications Union (voir *UIT*)

IUT Implementation Under Test

J2EE Java 2 Enterprise Edition. Ensemble de spécifications pour la réalisation de systèmes d'informations d'entreprise sur plateforme Java

JAuto Bibliothèque Java de manipulations d'automates, voir `http ://www.achilleus.net/jauto/`

Java Langage de programmation orienté-objet développé par SUN, actuellement dans sa version 5 (1.5)

Java Server Pages Documents interprétés par un serveur d'application et permettant de méler code HTML et code Java

JCA Java Connector Architecture. Élément de la norme J2EE permettant de rendre intéropérable des serveurs d'applications Java et d'autres systèmes

JDK Java Development Kit. Ensemble des outils permettant de construire et exécuter des programmes Java

JRE Java Runtime Environment. Plateforme d'exécution des programmes Java

Just-in-time (JIT) Technique d'exécution des applications à base de *code-octet* dans laquelle le code est compilé vers du code natif au moment du chargement

LDAP Lightweight Directory Access Protocol. Protocole générique d'accès à un annuaire d'entreprise permettant d'uniformiser le traitement des informations sur les utilisateurs d'un système d'information et la structure de l'entreprise

LOTOS Langage de spécification formel normalisé par l'UIT

LTS Labelled Transition System

Marshalling/Demarshalling Transformation des appels/retours de méthode en flux de données transportables sur un réseau

Maven Logiciel et infrastructure permettant de faciliter la production d'une application finie à partir du code source

MDA Model Driven Architecture

middleware voir *intergiciel*

MVC Modèle-Vue-Contrôleur. Patron de conception classique pour les interfaces utilisateurs découpant la gestion des interactions en trois entités : un *modèle* contenant l'état de l'application en fonction des actions de l'utilisateur, une *vue* contenant les informations et commandes accessibles à un instant donné à l'utilisateur et un *contrôleur* régissant les interactions entre vues et modèles

ObjectWeb Consortium international pour le développement de projets libres autour des technologies orienté-objet (voir `http ://www.objectweb.org`)

ODBC Open Database Connectivity. Mécanisme standardisé d'accès aux bases de données relationnelles sur plateforme Windows

OMG Object Management Group. Consortium international produisant des normes sur le développement et la conception de programmes utilisant le paradigme de la programmation orientée objet (voir `http ://www.omg.org`)

ORB Object Request Broker. Partie d'un intergiciel assurant la transmission des appels de méthodes entre objets distants

Patrons de conception Modèles de conception standardisés selon un certain langage et offrant des solutions éprouvées à des problèmes récurrents dans le cadre des langages orienté-objets

PCO Point de Contrôle et d'Observation

PKI Public Key Infrastructure ou Infrastructure à Clé Publique. Système de sécurité utilisant une authentification des utilisateurs par un système de clés cryptographiques asymétriques. Un tel système permet de pallier au problème de la diffusion des éléments nécessaires à l'authentification qui une vulnérabilité des systèmes à clés symétriques ou à mots de passe

POJO Plain Old Java Object. Objets Java classiques, par opposition aux EJB, JSP, objets répartis, ...

Recette Étape finale du processus de développement du logiciel

RPC Remote Procedure Call. Appel de Procédure à Distance, mécanisme logiciel maintenant la sémantique de l'appel de procédure, de fonction ou de méthode entre des entités séparées par un réseau

RUP Rational Unified Process ou Processus Unifié de Rational. Méthodologie de développement configurable centrée sur l'utilisation de modèles UML et la production rapide de multiples versions de l'application

Serveur d'Applications Logiciel de type intergiciel prenant en charge un certain nombre de fonctions techniques et permettant de réaliser plus facilement des applications réparties

Servlet Objet Java s'exécutant dans le contexte d'un serveur d'application et interagissant avec un navigateur Web par l'intermédiaire du protocole HTTP

SGDB Système de Gestion de Base de Données (Relationnelle)

Struts Système de gestion des interactions entre un navigateur et un serveur dans le cadre d'applications J2EE. Voir `http ://jakarta.apache.org/struts`

UIO Unique Input/Output

UIT Union Internationale des Télécommunications. Organisme de normalisation dans le domaine des télécommunications

UML *Unified Modelling Language* ou Langage de modélisation unifié. Un langage de conception normalisé par le consortium OMG à partir de langages et méthodologies préexistantes (Booch, OMT)

XMI XML Metadata Interchange. Dialecte XML définissant la représentation de modèles MOF

XML Extensible Markup Language. Langage permettant de représenter et transporter de manière uniforme tout type d'information structurée