Méthodes formelles pour les systèmes répartis et coopératifs

Serge HADDAD, Fabrice KORDON et Laure PETRUCCI

22 septembre 2006

Table des matières

Préface

Le présent ouvrage décrit les techniques de Génie Logiciel propres aux systèmes répartis et coopératifs développés pour les domaines dit « critiques ». Les systèmes critiques nécessitent le recours à certaines des méthodes formelles pour les phases amont et aval du Génie Logiciel : spécifications et vérification. Dégager l'essentiel des perspectives offertes par ces méthodes fut le challenge.

La convergence de vues multiples et variées et l'homogénéité de la présentation que recouvre le résultat final n'auraient pu être atteintes sans l'extraordinaire énergie d'un coordonnateur-né, Fabrice Kordon, à laquelle je tiens tout particulièrement à rendre hommage ici. L'architecture de l'ouvrage est en elle-même une preuve de l'excellence de ses auteurs dans ce domaine. Savoir rassembler dans un même ouvrage des problématiques aussi diverses que celles liées à l'expression des propriétés temporelles, la modélisation comportementale des composants et connecteurs dans un langage de description d'architecture (ADL), les différentes approches de *model checking*, relève d'un art de la synthèse qui n'est pas donné à tout le monde. Face à la multiplication actuelle des méthodes formelles, Serge Haddad, Fabrice Kordon et Laure Petrucci ont su inviter leurs collaborateurs à mettre en exergue les processus de modélisation des architectures réparties fiables et à porter l'accent sur ceux qui se révèleront particulièrement efficaces, en fournissant des exemples concrets dans la dernière partie de l'ouvrage.

Ayant invité Fabrice Kordon à la première journée ADL du club technique Génie Logiciel de la SEE [1], je fus très impressionnée par la maîtrise avec laquelle ce dernier captivait littéralement son auditoire, à la manière d'un grand conteur. La difficulté inhérente aux notions les plus complexes à appréhender, comme par exemple

---

Préface rédigée par Isabelle PERSEIL, présidente du club Géniel Logiciel (SEE).

1. Société de l'Électricité, de l'Électronique et des Technologies de l'Information et de la Communication : `http://www.see.asso.fr`

la réduction du nombre d'états dans les réseaux de Petri par analyse des symétries, s'estompait miraculeusement. Nombreux parmi ceux qui étaient présents me firent part de leur intérêt pour les travaux communs menés entre le Lip6 (Fabrice Kordon) et l'ENST (Laurent Pautet, Jerôme Hugues et Thomas Vergnaud) sur PolyORB (premier intergiciel schizophrène), et je les orientais vers les réunions organisées par la communauté MeFoSyLoMa [2]. Aussi, à l'issue de cette mémorable journée, j'émis le vœu pieux que les travaux présentés se concluent par un ouvrage car il me semblait égoïste et éphémère de les réserver aux seuls membres de la SEE.

Naturellement, il fut convenu que le cadre de cet ouvrage dépasserait largement celui de la journée ADL et engloberait d'une manière générale l'ensemble des préoccupations scientifiques du groupe de recherche qui participe aux travaux de la communauté MeFoSyLoMa, dans l'objectif de démystifier pour l'ingénieur architecte et l'étudiant en systèmes répartis, tout un pan de l'application des méthodes formelles.

Saluons enfin son indéfectible qualité : l'ouvrage s'appuie notamment sur une longue expérience d'enseignement jalonnée de nombreuses publications et de conférences internationales : Serge Haddad, Fabrice Kordon et Laure Petrucci enseignent maintenant depuis une vingtaine d'années dans les universités parisiennes les plus réputées (Dauphine, Pierre et Marie Curie, Paris Nord).

Cet ouvrage, premier du genre, exceptionnellement rédigé en français, représentera, j'en suis convaincue, une aide précieuse pour tous les ingénieurs désireux de se perfectionner dans l'utilisation des méthodes formelles dédiées aux systèmes répartis, qui y puiseront une mine d'informations sans avoir à courir d'un séminaire à l'autre. Il simplifiera grandement le travail du jeune doctorant de première année ayant à rédiger un état de l'art dans le domaine en référence.

Souhaitons-lui simplement tout le succès qu'il mérite.

2. Méthodes Formelles pour les Systèmes Logiciels et Matériels

Chapitre 1

Introduction générale

Problématique

Depuis plusieurs années déjà, il est de notoriété publique que la complexité des gros systèmes croît plus vite que notre capacité à les appréhender [LEV 97]. C'est en particulier le cas des systèmes répartis où l'exécution simultanée de plusieurs flots d'exécution rend extrêmement difficile l'élaboration de techniques de test fiables.

La simulation a toujours été un moyen d'éprouver une réalisation. Pour ce faire, on simule le modèle (qui doit s'appuyer sur une notation exécutable) ou on exécute le programme final (parfois une partie, en émulant son environnement d'exécution). Hélas, une telle démarche donne des résultats décevants dans le domaine des systèmes répartis car l'exécution se fait « au petit bonheur la chance ». Ainsi il est parfois impossible de retrouver une séquence d'exécution à l'origine d'une défaillance.

Le problème vient principalement d'une caractéristique simple de la répartition. Contrairement à un système centralisé, on ne peut déduire l'ordre d'exécution des instructions de manière absolue. Or ce dernier est un postulat important dans toute approche de test. Des techniques d'estampillage automatique des communications machine par machine permettent de pallier ce problème en permettant le « rejeu » d'une configuration d'exécution à l'identique [BON 96]. De tels mécanismes s'intègrent aisément dans une infrastructure d'exécution répartie (*middleware*).

Mais si de telles techniques permettent de déboguer des applications réparties, elles ne s'appliquent que lorsque ces dernières ont atteint un stade de développement

---

Introduction rédigée par Serge HADDAD, Fabrice KORDON et Laure PETRUCCI.

avancé. De plus, la multiplicité des exécutions possibles finit forcément par bloquer l'analyse des systèmes par des approches de test traditionnelles. S'il est indéniable que l'on ne pourra jamais éradiquer complètement une phase de test une fois le système réalisé, cette phase doit être associée à d'autres approches dans un processus de développement pour que les systèmes répartis soient raisonnablement fiables.

Parmi ces approches, on considère naturellement un travail en amont de la phase de développement. L'idée est de disposer au plus tôt dans la phase de conception d'un modèle du système à développer. Ce modèle peut s'appuyer sur des notations standardisées comme UML [CHA 05] mais, caractéristique importante pour les systèmes répartis, doit prendre en compte une sémantique comportementale afin de décrire sans ambiguité le comportement des composants d'un système réparti.

Un pas plus loin, on peut utiliser des techniques formelles pour démontrer que des propriétés d'un système réparti sont vérifiées. Les propriétés les plus intéressantes sont d'assurer que le déterminisme de certaines séquences d'évènements est garanti par une relation causale. D'autres propriétés permettant d'estimer le temps d'exécution sont également de plus en plus recherchées.

Cependant, les méthodes formelles ne sont hélas pas la solution miracle. De nombreux auteurs comme [LUQ 97] signalent plusieurs difficultés telles que la difficulté d'apprentissage, l'adaptation à un processus de développement de type industriel ou le passage à l'échelle. Ainsi, pour rendre ces approches utilisables, il faut être pragmatique et concilier des techniques formelles avec une démarche méthodologique permettant leur usage dans un monde industriel grâce à des outils utilisables par les ingénieurs [KOR 03].

Objectif du présent ouvrage

L'objectif de ce livre est justement de faire le point sur les techniques formelles utilisables dans le contexte des systèmes répartis et coopératifs. Ces systèmes sont caractérisés par :

– des composants distincts ayant un ou plusieurs flux d'exécution (*thread*), chacun d'eux s'exécutant potentiellement sur un processeur différent ;

– des communications asynchrones sur lesquelles la seule hypothèse couramment admise est la fiabilité de communication (mais au prix d'un déséquencement des échanges de messages entre processeurs) ; dans certains cas particuliers, d'autres hypothèses (ordre des messages préservé, perte de messages, etc.) peuvent être considérées ;

– l'absence d'un contexte global à l'application répartie ou, en d'autres termes, l'impossibilité pour un composant d'accéder aux données d'un autre composant autrement que via un message asynchrone.

Les caractéristiques énoncées ici sont celles, de plus en plus courantes, des systèmes répartis modernes. On ne sait actuellement que les adresser partiellement. Pire, dans certains projets actuellement envisagés comme l'automatisation de la conduite de véhicules en convois, on reste extrêmement démunis face à l'absence de techniques d'analyse et de développement (cela est l'objet de travaux de plusieurs grands projets européens).

La tendance actuelle pour un processus « idéal » de développement dédié aux systèmes répartis et coopératifs est de placer les phases de conception et de développement dans un continuum. Elle se situe dans la droite ligne d'une démarche *par prototypage* [KOR 03] centrée sur la notion de modèles.

Ce livre est plus particulièrement destiné aux lecteurs qui souhaitent avoir un panorama sur l'usage des méthodes formelles dans le développement de systèmes répartis. Les étudiants en master $2^{\text{ème}}$ année, les jeunes doctorants mais aussi des ingénieurs en R&D y puiseront une compréhension globale des techniques présentées ainsi que des références vers les travaux les plus à jour dans le domaine.

Plan de l'ouvrage

Le présent ouvrage a pour objectif de montrer les différentes phases de ce continuum : la modélisation, la vérification, la réalisation et leurs relations. Nous l'avons structuré en trois parties.

La première partie s'intéresse à la première étape de la phase de conception : la modélisation. Nous y discutons dans le chapitre 3 de la démarche de modélisation permettant de construire une spécification cohérente. Le chapitre 4 est ensuite consacré a une manière efficace de gérer les aspects temporisés[1] dans les systèmes. Enfin, le chapitre 5 s'intéresse à la description de l'architecture logicielle au moyen de langages dédiés : les ADL (architecture description language).

La seconde partie se focalise sur les techniques de vérification formelle les plus appropriées dans le cadre qui nous intéresse. Le chapitre 7 s'intéresse aux approches structurelles, qui permettent de déduire des propriétés à partir de l'analyse de la structure de la spécification. Cette technique propre aux réseaux de Petri est originale par rapport à d'autres langages de modélisation. Le chapitre 8 présente ensuite les techniques de vérification appliquées dans le cas de systèmes finis. L'objectif est de lutter

1. Dans cet ouvrage et conformément aux usages de la discipline, nous utiliserons le terme « temporisé » pour désigner les mécanismes visant à décrire et manipuler le temps dans les systèmes répartis. Le terme « temporel » désigne, quant à lui, les mécanismes permettant de décrire la *causalité* entre évènements dans un système réparti.

efficacement contre l'explosion combinatoire de l'espace d'états en utilisant des représentations abstraites et des structures de données compactes. Enfin, le chapitre 9 évoque les solutions envisageables lorsque l'espace d'états du système devient infini.

La troisième partie présente des applications concrètes des techniques exposées dans les deux premières parties. Le chapitre 11 se situe dans le contexte du développement d'un middleware. Le chapitre 12 expose des applications dans le domaine des services web. Enfin, le chapitre 13 présente une application dans le contexte de systèmes répartis adaptatifs (systèmes multi-agents).

Chaque partie est précédée d'un panorama visant à situer les différentes techniques présentées dans les chapitres suivants les unes par rapport aux autres.

La communauté MeFoSyLoMa

MeFoSyLoMa (Méthodes Formelles pour les Systèmes Logiciels et Matériels) est une association d'équipes issues de laboratoires dans la région parisienne [MEF 06]. Les principaux membres sont issus du LIP6[2] (Université P. & M. Curie - Paris 6), du LAMSADE[3] (Paris Dauphine), du LIPN[4] (Université Paris XIII), du LTCI[5] (ENST) et du CÉDRIC[6] (CNAM). Les membres de cette communauté comportant presque 70 chercheurs et doctorants ont tous un intérêt commun pour la construction de systèmes répartis (logiciels et/ou matériels) au moyen d'un cycle de développement promouvant la modélisation, l'analyse formelle, et l'implémentation centrés sur la notion de modèle. Cette communauté, créée au début de l'année 2005, se fédère autour d'un séminaire régulier (cinq à six par an) et la participation à des projets de recherche communs.

Les auteurs de cet ouvrage sont presque tous issus de cette communauté, certaines contributions provenant de membres du laboratoire IBISC[7] (Université d'Évry-Val-d'Esssone) qui, sans être à l'origine de MeFoSyLoMa, suivent régulièrement les séminaires.

Bibliographie

[BON 96] BONNAIRE X., BAGGIO A., PRUN D., « Intrusion Free Monitoring : An Observation Engine for Message Server Based Applications », *9th International Conference on Parallel And Distributed Computing Systems (ISCA)*, p. 88-93, 1996.

2. Laboratoire d'Informatique de Paris 6.

3. Laboratoire d'Analyse et Modélisation de Systèmes pour l'Aide à la Décision.

4. Laboratoire d'Informatique de Paris Nord.

5. Laboratoire Traitement et Communication de l'Information.

6. Centre d'Étude et de Recherche en Informatique du CNAM.

7. Informatique, Biologie Intégrative et Systèmes Complexes

[CHA 05] CHARROUX B., OSMANI A., THIERRY-MIEG Y., Eds., *UML2*, Pearson Education, 2005.

[KOR 03] KORDON F., HENKEL J., « An overview of Rapid System Prototyping today », *Design Automation for Embedded Systems*, vol. 8, n°4, p. 275–282, Kluwer, december 2003.

[LEV 97] LEVESON N., « Software Engineering : Stretching the Limits of Complexity », *Communications of the ACM*, vol. 40(2), p. 129–131, 1997.

[LUQ 97] LUQI, GOGUEN J., « Formal Methods : Promises and Problems », *IEEE Software*, vol. 14(1), p. 73–85, January / February 1997.

[MEF 06] MEFOSYLOMA, « MeFoSyLoMa, page d'accueil, `www.mefosyloma.cnam.fr` », 2006.

PREMIÈRE PARTIE

Modèles et langages de spécification

Chapitre 2

Panorama des modèles et langages de spécification

2.1. Motivation

Les systèmes développés de nos jours sont de plus en plus complexes et leur fonctionnement peut avoir des conséquences importantes sur leur environnement, voire irréversibles : systèmes avioniques [PET 03], médicaux [JØR 04], etc. [Ind].

Il est par conséquent indispensable d'obtenir des systèmes sûrs, dont le fonctionnement a pu être vérifié avant la mise en œuvre.

La vérification de tels systèmes critiques est traditionnellement effectuée selon des approches dépendant du problème considéré. Par exemple, pour des systèmes matériels, tels que les systèmes avioniques, des bancs de test physiques sont utilisés. Dans un tel cas, une maquette de l'avion, reproduisant l'objet réel est utilisée au sol et des traces de simulations sont analysées. Il est clair qu'une telle approche nécessite une infrastructure matérielle lourde et coûteuse, ainsi qu'un personnel qualifié pour conduire les tests. La mise en place de ce type de banc de test demande également un temps relativement considérable. De plus, les traces obtenues détaillent le fonctionnement du système avec un grain tellement fin qu'il est difficile de les interpréter. On voit alors que pour pouvoir effectuer une vérification efficace, il faut se situer à un *niveau d'abstraction* reflétant le problème rencontré.

Pour atteindre des objectifs de sûreté, il est nécessaire de s'abstraire du système physique en utilisant un *modèle*. Celui-ci présente de nombreux avantages :

Chapitre rédigé par Laure PETRUCCI.

– n'étant pas matériel, son coût de mise en œuvre est relativement faible ;

– il peut être analysé par des outils informatiques, et le cas échéant modifié sans grand surcoût ;

– le modèle du système considéré permet au concepteur d'avoir une vision plus claire et rigoureuse de ce qu'il souhaite mettre en œuvre ;

– une fois les vérifications concluantes sur le modèle, un prototype expérimental peut-être développé, avec un degré de confiance suffisant, une partie des erreurs ayant déjà été éliminées ;

– de plus, une telle spécification aide à la maintenance ultérieure par une personne extérieure au projet initial.

Divers types de spécification peuvent être envisagés, avec des objectifs complémentaires. Nous allons les expliciter dans les sections suivantes.

2.2. Modèles semi-formels

La spécification d'un système peut être plus ou moins formelle selon les techniques choisies. L'utilisation de *modèles semi-formels* tels que UML (*Unified Modeling Language*, [CHA 05]) permet d'atteindre une partie des objectifs visés :

– l'écriture de la spécification permet au concepteur de mieux comprendre le fonctionnement attendu de son système ainsi que les interactions entre les différents composants. L'écriture du modèle conduit à remettre en cause des choix conceptuels d'une part et à une meilleure compréhension du système développé d'autre part.

– un modèle dans une notation assez aisée à comprendre facilite la communication avec les clients.

– le modèle constitue une documentation précise lors de la maintenance.

Une spécification UML se décompose en différentes étapes d'élaboration qui se traduisent par divers diagrammes. Ceux-ci présentent l'avantage d'être relativement simples à comprendre. Nous présentons ici quelques grandes lignes d'UML.

Tout d'abord, un *diagramme de cas d'utilisation* permet de situer le contexte : il décrit les relations entre les cas d'utilisation et les acteurs du système considéré. Les *cas d'utilisation* résument des séquences d'actions effectuées avec le système auquel on s'intéresse. Les *acteurs* sont les intervenants extérieurs (personnes ou autres systèmes) entrant en interaction avec le système modélisé. Les cas d'utilisation peuvent correspondre à plusieurs scénarios d'exécution.

EXEMPLE.– Considérons un problème d'assurance de voitures. Il fait intervenir deux acteurs : le *client* et l'*assureur*. Le client peut *déclarer un accident* à l'assureur. Ceci est une opération composée de plusieurs actions plus élémentaires telles que la rédaction du constat, l'envoi par mail ou par courrier, etc. L'assureur peut, selon le cas,

décider d'*effectuer un remboursement* ou non du client. Il y a donc plusieurs scénarios possibles. Un diagramme de cas d'utilisation correspondant est présenté dans la figure 2.1.

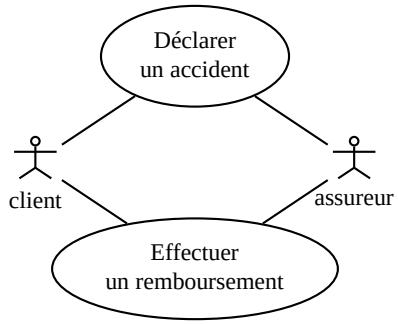


Figure 2.1. *Diagramme de cas d'utilisation*

Ces diagrammes permettent de clarifier le problème à modéliser, de réfléchir à ce qui devrait ou ne devrait pas se produire, et ce sans se noyer dans des détails techniques. Ils permettent également, dans une phase ultérieure, de concevoir des *jeux de tests* visant à valider le fonctionnement du système.

Le système modélisé peut être structuré en classes, représentées dans un *diagramme de classes*. Les différentes classes possèdent des *attributs* et des *opérations* qui leur sont propres, et sont reliées entre elles par des *relations* qui peuvent être de différentes natures.

EXEMPLE.– Dans notre exemple de déclaration d'accident, considérons le diagramme de classes de la figure 2.2. Un véhicule possède plusieurs attributs tels que le numéro d'immatriculation, le type, la marque. De même, un client a un nom, un prénom et un numéro d'assuré. Le client peut effectuer une opération `déclarer()` un accident. Une déclaration concerne exactement un véhicule et un client.

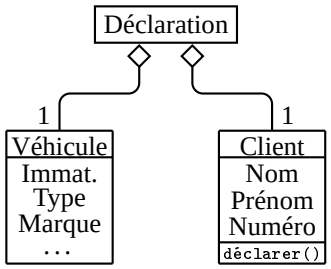


Figure 2.2. *Diagramme de classes*

Chaque objet du système réalise une partie des fonctionalités attendues. Le comportement global est donc obtenu en faisant *coopérer* les objets entre eux. Ces coopérations entre objets s'effectuent par échange de messages, décrits dans un *diagramme de communication*.

EXEMPLE.– Le diagramme de communication de la figure 2.3 montre que le client doit faire une déclaration en envoyant un message `déclarer`. La déclaration est ensuite `envoyée` à l'assureur qui prend une décision pour `répondre` au client.

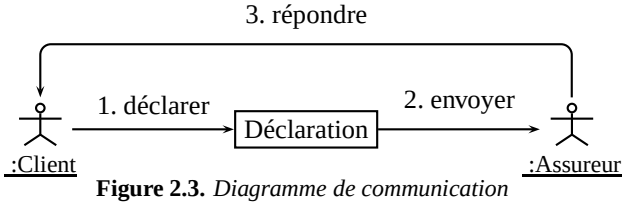


Figure 2.3. *Diagramme de communication*

Ces informations peuvent également être présentées par des *diagrammes de séquences* qui montrent de plus la création d'objets.

EXEMPLE.– Le diagramme de séquences de la figure 2.4 indique les échanges de messages entre les différents objets. On voit également que l'opération du client pour déclarer un accident génère un objet déclaration.

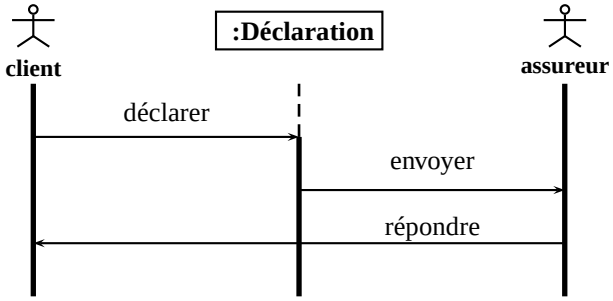


Figure 2.4. *Diagramme de séquences*

Le flot circulant dans le système est représenté sur un *diagramme d'activité*, qui complète les diagrammes de communication et de séquences. Il montre l'évolution du système du point de vue d'un acteur.

EXEMPLE.– Du point de vue de l'assureur, les activités ont lieu comme indiqué dans la figure 2.5 : il reçoit une déclaration, la traite, puis envoie la réponse au client.

Une description comportementale des classes, des cas d'utilisation, des acteurs, etc. peut être représentée au moyen d'un *diagramme d'états*. Au cours de l'évolution

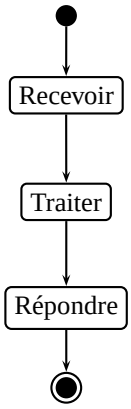


Figure 2.5. *Diagramme d'activité, vu de l'assureur*

du système, les différentes entités en jeu changent d'état. Ceci peut être dû à des évènements externes qui déclenchent une activité particulière.

EXEMPLE.– Pour notre exemple de déclaration d'accident, le diagramme d'états de l'assureur est présenté dans la figure 2.6. Initialement l'assureur attend. Lorsqu'il reçoit une déclaration, il passe dans un nouvel état où il est prêt à la traiter. Il vérifie la déclaration. Deux cas sont alors possibles : soit il n'y a pas de problème (OK), et on pourra alors effectuer le remboursement (envoi d'un réponse positive au client), soit il y a un problème (NOK) et l'assureur envoie une réponse négative au client.

Des outils logiciels permettent de construire une spécification complexe. Ils vérifient la cohérence entre les différents diagrammes. De plus, ils proposent la génération de code et de scénarios de test.

D'autres diagrammes existent [CHA 05], mais leur présentation sort du cadre de ce livre.

Pour conclure, des techniques semi-formelles telles qu'UML permettent de se poser de nombreuses questions sur le système à modéliser qui peuvent être aussi bien de haut niveau qu'adresser des problèmes techniques d'implémentation. La panoplie de diagrammes ainsi conçus donne une bonne vue des différents aspects du problème. Toutefois, même si ces diagrammes sont faciles à comprendre par un non-spécialiste, il peut être difficile de les appréhender comme un tout. De plus, la validation du système ne peut être exhaustive. Les modèles formels visent à pallier à ces problèmes.

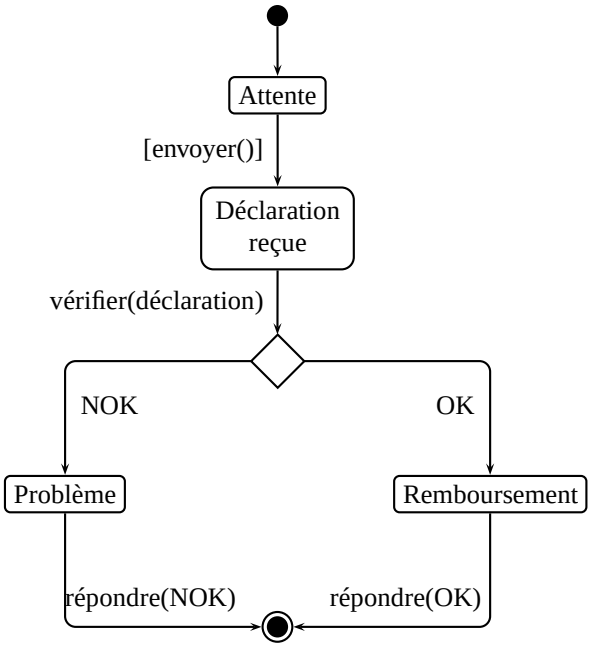


Figure 2.6. *Diagramme d'états de l'assureur*

2.3. Modèles formels

Les *modèles formels* permettent de prouver formellement (i.e. mathématiquement) le bon fonctionnement du système développé. On est alors complètement sûr que quelle que soit l'évolution du système, son comportement sera celui attendu.

Outre les avantages des méthodes précédentes, les méthodes formelles offrent des outils d'analyse du système modélisé :

– la simulation permet de se convaincre du bon fonctionnement du système, ou éventuellement de découvrir quelques erreurs. Leur correction à ce stade est peu coûteuse, comparée à un système déjà mis en œuvre, que ce soit avec des moyens logiciels ou matériels.

– la vérification exhaustive du comportement du système. On applique dans un premier temps des techniques de simulation pour effectuer un débogage grossier, puis on affine le modèle au travers de la vérification des propriétés souhaitées.

Il existe une pléthore de modèles et langages de spécification. Nous nous intéresserons plus particulièrement, dans ce livre, aux *spécifications algébriques*, aux *automates* [BÉR 01a], aux *réseaux de Petri* [GIR 03, DIA 03], et aux *algèbres de processus* [BER 01b]. Ceci est justifié par l'existence de méthodologie de spécification, la possibilité d'utiliser des méthodes structurelles, et d'introduire des contraintes temporelles dans les modèles considérés. Enfin, les *langages de description d'architectures* [MED 00] permettent la composition de modèles.

2.3.1. *Spécifications algébriques*

Les spécifications algébriques ont été développées initialement à partir de l'idée d'abstraction de données (« types abstraits de données »), et ont été à la source des langages de programmation orientés objet (cf. le concept de classe). Elles ont ensuite été développées et étendues pour produire les spécifications formelles de besoins fonctionnels et pour la conception modulaire de logiciel. Après la production de nombreux langages de spécifications algébriques, le langage CASL (*Common Algebraic Specification Language*) a été développé dans le cadre du projet international CoFI, *Common Framework Initiative for algebraic specification and development*. Ce langage est basé sur une sélection critique de constructions qui avaient été déjà explorées, telles que les sous-sortes, les fonctions partielles, la logique du premier ordre, les spécifications structurées et architecturales [AST 02]. Le site du projet CoFI [COF] contient les documents développés dans le cadre de ce projet, le langage est décrit dans le manuel utilisateur [BID 04], et le manuel de référence [CoF 04] fournit une sémantique formelle complète.

Une spécification CASL peut comporter la déclaration de types, d'opérations et de prédicats (munis de leur arité), et des axiomes qui sont des formules du premier ordre. Certaines opérations jouent le rôle de générateurs (ou constructeurs) et figurent dans la déclaration du type (« *datatype declaration* »). Une spécification simple a la forme suivante :

spec NOMSPEC=
  type $nom\_type ::= nom\_gen(args) \mid \ldots$
  op $nom\_op : args\_op \rightarrow res\_op$
  ...
  pred $nom\_pred : arg\_pred$
  ...
  axioms **%%**formules du premier ordre
  end

Le langage comporte des primitives pour construire de manière modulaire des spécifications : l'union **and** et l'extension **then** peuvent être utilisées pour structurer les spécifications.

spec NOMSPEC= SP_1 **and** ... **and** SP_j **then**
type $nom\_type ::= nom\_gen(args) \mid \ldots$

EXEMPLE.– Reprenons l'exemple de la section 2.2. Une spécification des déclarations d'accidents importe les modules des spécifications des véhicules et des clients.

spec DECLARATION = VEHICULE **and** CLIENT **then**
type $Declaration ::= \ldots$
end

En pratique, une spécification de système réaliste comporte des opérations totales et partielles [BID 04], aussi le symbole ? permet-il de distinguer les opérations et générateurs partiels, et les domaines de définition sont donnés dans les axiomes.

spec NOMSPEC=
type $nom\_type ::= nom\_gen(args) \mid \ldots$
op $nom\_op : args\_op \rightarrow ? \; res\_op$ **%%**opération partielle
...
axioms
$def \; nom\_op(args) \Leftrightarrow \ldots$

La construction **free** impose la sémantique initiale et permet d'éviter le recours explicite à la négation. En effet, dans les modèles de spécifications **free**, les valeurs des termes sont distinctes sauf si leur égalité résulte des axiomes de la spécification, toute coincidence non intentionnelle est donc impossible [BID 04].

spec NOMSPEC= SP_1 **and** ... **and** SP_j **then**
free { type $nom\_type ::= nom\_gen(args) \mid \ldots$
op $nom\_op : args\_op \rightarrow ? \; res\_op \ldots$
axioms ...**}**

EXEMPLE.– Un compteur modulo 3, utilisant des opérations d'incrémentation de 1 et d'incrémentation de 2, tel qu'utilisé dans la section 2.3.2, peut être spécifié comme suit :

spec COMPTEUR =
free { type $Compteur ::= 0 \mid suc(Compteur)$;
%% suc ajoute un
axioms $suc(suc(suc(0))) = 0$;
op $add2 : Compteur \rightarrow Compteur$
axioms $add2(0) = suc(suc(0))$;
$add2(suc(0)) = 0$;
$add2(suc(suc(0))) = suc(0)$; **}**
end

Les spécifications génériques sont très utiles pour la réutilisation. La spécification de leur paramètre est très simple, et une instance de la spécification générique est obtenue

en fournissant une spécification argument pour chaque paramètre. La spécification ci-dessous est une extension de l'instance de la spécification générique SET[ELEM] par INT (ces spécifications sont dans la bibliothèque de base [ROG 04]).

spec NOMSPEC = SET [INT] . . . **then** . . .
Remarquons qu'en spécification algébrique les propriétés sont exprimées dans les axiomes, et il est bien sûr possible (et utile) de démontrer des théorèmes qui sont des conséquences de ces propriétés.

Des extensions des langages de spécification algébrique sont proposées afin de pouvoir prendre en compte les systèmes dynamiques et des propriétés exprimées en logique temporelle. Par exemple, CASL-LTL [REG 03] utilise la *Labelled Transition Logic* [AST 01] et permet d'exprimer les états d'un système et les transitions entre états provoquées par des évènements.

2.3.2. *Automates*

Les *automates* présentent de manière explicite les états possibles du système. Il en existe différentes sortes (automates communicants, temporisés, avec variables, etc.) disposant de caractéristiques spécifiques.

Un automate peut être décrit par un graphe où les états sont des nœuds. Le passage d'un état à un autre s'effectue par *franchissement d'une transition*, représentée par un arc reliant les nœuds. Le choix de la transition peut être indéterministe, dans la mesure où plusieurs transitions peuvent être franchies à partir d'un même état. Cela permet de représenter différentes possibilités d'évolution d'un système.

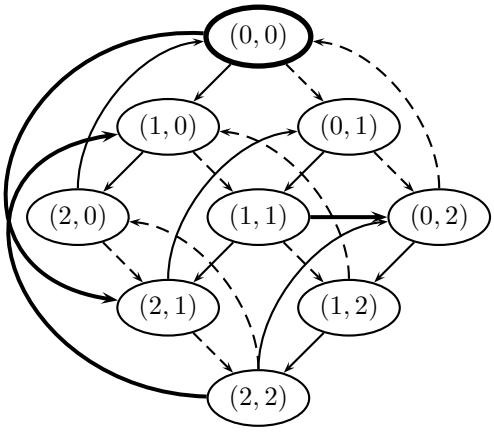


Figure 2.7. *Automate représentant un système à* 2 *compteurs*

EXEMPLE.– Considérons un système comportant 2 compteurs modulo 3, c'est-à-dire prenant successivement les valeurs 0, 1, 2, 0, etc. Les deux compteurs peuvent évoluer indépendamment, par incrément de leur valeur. D'autre part, lorsqu'ils ont la même valeur, on peut décider d'incrémenter le premier compteur de 2 unités, alors que le second n'augmente que d'une unité. L'exemple est modélisé par l'automate de la figure 2.7.

Les états ont un nom composé des valeurs des deux compteurs, i.e. $(0, 2)$ représente l'état dans lequel le premier compteur a la valeur 0, et le second la valeur 2. Les arcs avec un trait plein sont les transitions augmentant le premier compteur seulement, ceux en pointillés correspondent à l'incrémentation du second compteur seulement. Enfin, les arcs en trait gras modélisent les transitions augmentant le premier compteur de 2 unités et le second d'une unité. L'état initial du système est indiqué en gras (les deux compteurs ont la valeur 0).

2.3.3. *Réseaux de Petri*

Les *réseaux de Petri* [GIR 03, DIA 03] sont un autre formalisme permettant d'avoir une vue à la fois des états et des transitions du système. Le graphe comporte deux types de nœuds : les *places* (cercles ou ellipses) et les *transitions* (rectangles). Les places représentent une partie de l'état du système. Elles contiennent des *jetons* indiquant le nombre d'occurrences de ce sous-état. Comme dans les automates, les transitions représentent les évènements pouvant avoir lieu. Les arcs en entrée d'une transition déterminent la *pré-condition* au franchissement de la transition, i.e. les conditions devant être satisfaites pour que l'action puisse avoir lieu. De même, les arcs en sortie de la transition indiquent la *post-condition*, i.e. le résultat du franchissement. La sémantique de *franchissement* d'une transition t consiste donc à supprimer les jetons en entrée de la transition t, comme indiqué par les pré-conditions, et ajouter les jetons dans les places en sortie de la transition t, suivant les post-conditions.

EXEMPLE.– Le réseau de Petri de la figure 2.8 modélise notre exemple de système à compteurs.

Les conventions utilisées pour la signification des traits des arcs est la même que dans l'automate de la figure 2.7. Les places à gauche de la figure ($C1_0$, $C1_1$ et $C1_2$) indiquent les différentes valeurs que peut prendre le premier compteur, tandis que celles de droite ($C2_0$, $C2_1$ et $C2_2$) représentent celles du second compteur. De même, les transitions de gauche ($t1_{01}$, $t1_{12}$ et $t1_{20}$) correspondent à l'incrémentation du premier compteur et celles de droite ($t2_{01}$, $t2_{12}$ et $t2_{20}$) à celle du second compteur. Les transitions du milieu (f_1, f_2 et f_3) représentent les actions incrémentant le premier compteur de deux unités et le second d'une unité. On remarque que ces transitions ne sont franchissables que lorsque toutes leurs pré-conditions sont satisfaites, c'est-à-dire lorsque les compteurs sont égaux. L'état initial du système est représenté par la distribution de jetons dans les places : les deux compteurs ont la valeur 0.

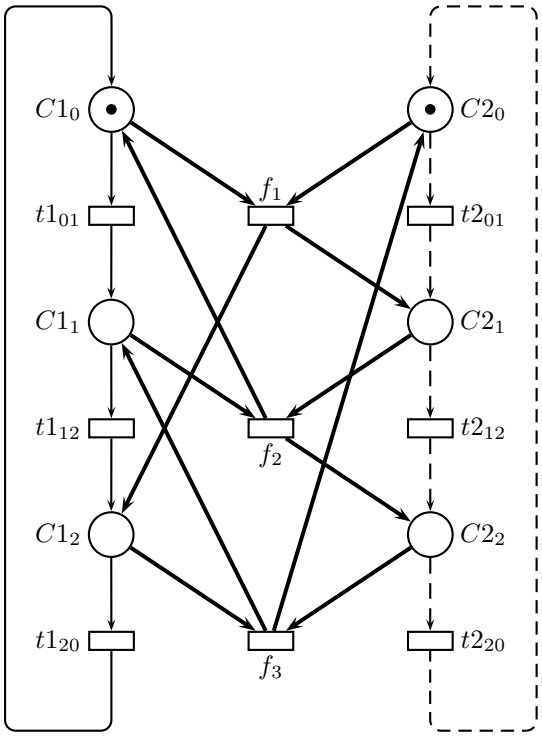


Figure 2.8. *Réseau de Petri représentant un système à* 2 *compteurs*

Le comportement du réseau de Petri est souvent analysé en construisant son *graphe d'états*. Ce graphe montre une représentation exhaustive des différents états atteignables par le système. À partir de l'état initial, les différentes posibilités de franchissements sont examinées et les marquages résultants sont ajoutés comme nœuds du graphe d'états, alors que les transitions étiquettent les arcs montrant ainsi le passage d'un état à un autre. Le graphe d'états du réseau de Petri de la figure 2.8 est similaire à l'automate de la figure 2.7 (seules les étiquettes d'arcs manquent sur la figure de l'automate), où un marquage $(C1_x, C2_y)$ est représenté par le couple (x, y).

Le systèmes à modéliser étant de taille de plus en plus importante, le modèle de réseaux de Petri a évolué vers des *réseaux de haut niveau* [JEN 91], dans lesquels le flux est représenté par une structure de réseau de Petri et les données sont explicitées au moyen d'un langage de spécification de données. Ce langage peut être dédié [AMI], un langage de programmation [CPN], ou une spécification de type abstrait algébrique [REI 91]. Les réseaux de haut niveau sont appréhendés facilement par quelqu'un ayant une expérience de programmation.

Les jetons transportent des données qui sont testées et éventuellement modifiées lors du franchissement de transitions. Les arcs du réseau sont étiquetés par des termes spécifiant les pré-conditions requises et les valeurs associées aux jetons créés.

EXEMPLE.– La figure 2.9 présente une modélisation du système à compteurs à l'aide d'un réseau de Petri de haut niveau.

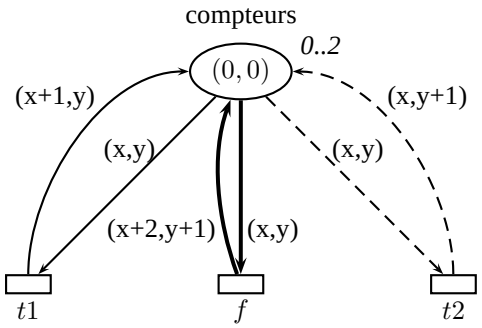


Figure 2.9. *Réseau de haut niveau représentant un système à 2 compteurs*

Le réseau ne comporte qu'une seule place, qui contient toujours un unique jeton dont la valeur est (x, y) où x est la valeur du premier compteur et y celle du second. Initialement, la place *compteurs* contient un jeton de valeur $(0, 0)$. L'inscription $0..2$ à côté de la place indique que les jetons prennent leur valeur dans l'intervalle de 0 à 2. La transition $t1$ incrémente le premier compteur. Elle correspond donc à un repliage des transitions $t1_{01}$, $t1_{12}$ et $t1_{20}$ de la figure 2.8. De même, la transition $t2$ incrémente le second compteur (repliage de $t2_{01}$, $t2_{12}$ et $t2_{20}$) et f incrémente le premier compteur de 2 unités et le second d'une seule (repliage de f_1, f_2 et f_3).

2.3.4. *Algèbres de processus*

Les *algèbres de processus* constituent une famille de formalismes mathématiques pour la description de systèmes concurrents. De nombreuses algèbres ont été ainsi définies, les deux principales familles étant dérivées des algèbres CSP (*communicating sequential processes* [HOA 78]) et CCS (*calculus of communicating systems* [MIL 80]).

Les algèbres de processus les plus expressives comportent le plus souvent des *définitions paramétrées*, des *expressions de processus* et des *préfixes d'actions*. Les données manipulées sont des valeurs simples (CCS par valeur), des évènements structurés (CSP) ou encore des *noms* (pour le π-calcul [MIL 92]).

Une expression de processus est généralement construite à l'aide des opérateurs suivants[1] :

– 0 est le *processus inactif* ;

– $\alpha.P$ est la *composition séquentielle* du préfixe d'action α suivi de l'expression de processus P ;

– $P \| Q$ représente la *composition parallèle* des expressions de processus P et Q. La sémantique utilisée est le plus souvent une *sémantique d'entrelacement* avec une éventuelle synchronisation des processus P et Q. Les processus P et Q coexistent ;

– $P + Q$ est un *choix indéterministe* entre les processus P et Q. Seul l'un des deux s'exécute ;

– enfin, $A(v_1, \ldots, v_n)$ représente un appel à une *définition paramétrée*.

Les *actions* élémentaires des processus se limitent souvent aux suivantes :

– τ représente une *action interne* qui ne peut être observée depuis l'extérieur du processus ;

– $c!v$ indique l'*émission* de la valeur v sur le nom c utilisé en tant que canal de communication. Le préfixe $c!$ indique une émission sans passage de valeur (signal) ;

– $c?(x)$ représente la *réception* d'un nom sur le canal c. La variable x est alors instanciée avec la valeur reçue dans la suite du processus. La réception sans valeur $c?$ est le pendant de l'émission $c!$;

– $[a = b]$ teste l'*égalité* entre les valeurs ou noms a et b ;

– $[a \neq b]$ teste l'*inégalité* entre les valeurs ou noms a et b ;

– (νx) correspond à la *construction du nom privé* x. Ce nom n'est visible que du processus qui l'a créé. On parle également de *restriction*.

EXEMPLE.– Décrivons un processus simplifié de distributeur de boisson. Les canaux de communication explicitent l'interface du distributeur : canal $piece$ pour l'insertion des pièces et canaux $cafe$, the et eau pour les sélecteurs de boisson. La spécification proposée est la suivante :

$$\left[\begin{array}{l} Distrib = piece?(p).([p = 1e]\, CafeThe + [p = 50cts]\, Eau) \\ CafeThe = the?.Distrib + cafe?.Distrib \\ Eau = eau?.Distrib \end{array}\right.$$

Le fonctionnement de la machine est le suivant. Si la pièce insérée a pour montant un euro, le distributeur propose au choix (dit externe, décidé par l'environnement) entre du thé ou du café. Si la pièce est de 50 centimes, seule de l'eau (fraîche cependant) est proposée. On peut également composer cette machine avec, par exemple, un client,

1. Nous utilisons ici une syntaxe de type mixte CCS/CSP mais des équivalents existent dans la plupart des algèbres de processus

pour obtenir un système Sys complet :

$$\left[\begin{array}{l} Client = piece!(1e).the!.piece!(50cts).eau!.0 \\ Sys = Client \parallel Distrib \end{array}\right.$$

Ici, le client est mis en parallèle avec le distributeur. Son comportement est de payer et de sélectionner d'abord un thé, puis de l'eau fraîche.

Les algèbres de processus se déclinent souvent en de nombreuses variantes : algèbres temporisées, probabilistes, réparties, asynchrones, etc.

2.4. Après la spécification, la vérification

Une fois le système modélisé, on peut dans un premier temps se convaincre de sa validité en effectuant des simulations. Pour cela, on part d'un *état initial* du système, et on déroule pas à pas l'exécution du système. Dans le cas des automates ou des réseaux de Petri, les transitions sont franchies à la suite les unes des autres. La simulation peut s'effectuer principalement de deux manières :

La simulation automatique est entièrement effectuée par l'outil. Lorsqu'un choix entre transitions est possible, il est fait de manière indéterministe.

La simulation manuelle ou guidée propose à l'utilisateur de choisir un franchissement soit à chaque étape, soit lorsqu'un choix entre plusieurs franchissements se présente. Cela permet de guider l'outil vers une direction où l'on suppose qu'il pourrait y avoir des erreurs.

Dans tous les cas, on sera amené à étudier une trace de l'exécution, vérifiant ainsi, avec un gros grain, le bon fonctionnement du système pour cette exécution. Cela permet également de corriger des erreurs grossières.

Un des critères principaux à déterminer pour une simulation est la condition d'arrêt. Elle peut être exprimée en termes de :

– *nombre de franchissements* à effectuer pour chaque simulation. La limite fixée doit être alors suffisament grande pour que la simulation conduise à un résultat significatif.

– *propriété ne devant pas être satisfaite*. L'objectif est donc de trouver une trace d'exécution violant une propriété souhaitée. Dès qu'un état ne satisfait pas la propriété, la simulation se termine, et l'examen de la trace permet de comprendre comment le dysfonctionnement peut avoir lieu.

Cette dernière technique se rapproche de la problématique de la vérification, que l'on met en œuvre dans un second temps, pour effectuer une analyse plus fine du système. Il est alors en effet nécessaire d'exprimer les propriétés souhaitées à l'aide d'un formalisme compatible avec le modèle et l'outil choisis.

Pour vérifier des propriétés spécifiques du système, il est nécessaire de les exprimer dans un formalisme idoine. Les réseaux de Petri disposent d'un ensemble de propriétés « classiques », génériques à tout réseau de Petri (bornes, vivacité, états morts, etc.). Toutefois, les propriétés auxquelles l'on s'intéresse sont souvent plus complexes et dépendent généralement du système étudié (par exemple, une exclusion mutuelle entre deux processus). Ces propriétés sont alors formalisées, par exemple en *logique temporelle* [BÉR 01a]. La logique temporelle a pour but d'exprimer des séquencements temporels. Elle se divise principalement en deux logiques :

LTL (*Linear Time Logic*) traduit des propriétés sur les chemins d'exécution ;

CTL (*Computation Tree Logic*) exprime des propriétés sur les états rencontrés lors de l'évolution du système.

La vérification de propriétés suit plusieurs approches, qui seront abordées dans la partie II :

– *l'analyse structurelle* s'intéresse à la validation de propriétés intrinsèques du système, indépendamment de l'état initial.

– *l'analyse comportementale* passe par une représentation totale ou partielle du comportement du modèle.

Chacune de ces techniques présente des avantages et inconvénients. Il est par conséquent indispensable d'effectuer les bons choix dès le début de l'écriture de la spécification.

2.5. Plan de la partie I

La spécification et la validation de systèmes au moyen de modèles formels est importante pour le développement de systèmes sûrs. Pour pouvoir mener à bien une telle démarche, il faut tout d'abord *spécifier* le système que l'on souhaite vérifier. Cette première étape est d'autant plus importante qu'elle conditionne les suivantes. Le spécifieur est amené à se poser un certain nombre de questions sur son système de manière à choisir le modèle ou langage de spécification le plus approprié. Il doit également réfléchir aux propriétés qu'il souhaite vérifier pour se placer à un niveau d'abstraction lui permettant de décrire ces propriétés tout en ne rentrant pas dans des

détails superflus. Une fois ces aspects pris en considération, la *démarche de spécification* nécessite également de savoir comment démarrer l'écriture de la spécification à partir de la connaissance du système. Ceci fait l'objet du chapitre 3.

Certains systèmes requièrent la prise en compte explicite de contraintes temporelles ou dynamiques. On s'orientera alors plutôt vers des *modèles temporisés* ou *hybrides*. Ils permettent au spécifieur de vérifier, outre des propriétés usuelles comme dans des systèmes non temporisés, des propriétés liées à des contraintes temps-réel. On peut alors s'intéresser à la durée de certains évènements, vérifier qu'une certaine action a lieu avant qu'un laps de temps spécifique se soit écoulé, etc. Le temps peut être pris en compte en utilisant une horloge globale, c'est-à-dire commune à tous les composants du système, une horloge locale à chaque composant, des chronomètres, etc. Ces types de modèles font appel à des techniques de vérification adaptées. Ces modèles ainsi que les méthodes de vérification associées sont explicités dans le chapitre 4.

Les modèles de systèmes réels deviennent rapidement trop gros pour assurer un suivi, une compréhension et une analyse corrects. De plus, des parties de tels systèmes peuvent être réutilisées au sein d'un autre système similaire, ou lors d'une opération de maintenance telle que le remplacement d'un sous-système par un autre. Certains modèles, comme les réseaux colorés hiérarchiques [JEN 92], prévoient dès le départ une structure modulaire. Mais ces concepts sont étendus pour permettre de développer des modèles par composants, éventuellements hétérogènes. Il faut alors non seulement modéliser le composant — et vérifier son comportement individuel, mais aussi indiquer comment il s'interface avec d'autres composants du système (entrées/sorties, connecteurs, etc.). Une telle approche est adoptée par les *langages de description d'architectures* (ADL). Le chapitre 5 présente les ADLs formels, visant à la vérification de propriétés, et les ADLs d'implantation, dont l'objectif est de faciliter le passage à un outil réel.

Remerciements. Nous remercions C. CHOPPY pour sa contribution à la rédaction de la section 2.3.1.

2.6. Bibliographie

[AMI] CPN-AMI : home page, `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.

[AST 01] ASTESIANO E., REGGIO G., « Labelled Transition Logic : An Outline », *Acta Inf.*, vol. 37, n°11–12, 2001.

[AST 02] ASTESIANO E., BIDOIT M., KIRCHNER H., KRIEG-BRÜCKNER B., MOSSES P. D., SANNELLA D., TARLECKI A., « CASL : The Common Algebraic Specification Language », *Theoretical Comput. Sci.*, vol. 286, n°2, p. 153–196, 2002.

[BÉR 01a] BÉRARD B., BIDOIT M., FINKEL A., LAROUSSINIE F., PETIT A., PETRUCCI L., SCHNOEBELEN PH., *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer, 2001.

[BER 01b] BERGSTRA J. A., PONSE A., SMOLKA S. A., *Handbook of process algebra*, Elsevier Science, 2001.

[BID 04] BIDOIT M., MOSSES P. D., CASL *User Manual*, LNCS 2900 (IFIP Series), Springer, 2004, With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[CHA 05] CHARROUX B., OSMANI A., THIERRY-MIEG Y., *UML 2*, Pearson Education, 2005.

[COF] CoFI, `http://www.cofi.info`.

[CoF 04] COFI (THE COMMON FRAMEWORK INITIATIVE), CASL *Reference Manual*, LNCS 2960 (IFIP Series), Springer, 2004.

[CPN] DESIGN/CPN online, `http://www.daimi.au.dk/designCPN`.

[DIA 03] DIAZ M., Ed., *Vérification et mise en œuvre des réseaux de Petri*, Hermes science publication - Lavoisier, 2003.

[GIR 03] GIRAULT C., VALK R., *Petri Nets for Systems Engineering : A Guide to Modeling, Verification and Applications*, Springer, 2003.

[HOA 78] HOARE C. A. R., « Communicating Sequential Processes », *Communications of the ACM*, vol. 21, n°8, p. 666–677, 1978.

[Ind] Examples of Industrial Use of CP-nets, `http://www.daimi.au.dk/CPnets/intro/example_indu.html`.

[JEN 91] JENSEN K., ROZENBERG G., *High-Level Petri Nets*, Springer, 1991.

[JEN 92] JENSEN K., *Coloured Petri Nets : Basic concepts, analysis methods and practical use. Volume 1 : basic concepts*, Monographs in Theoretical Computer Science, Springer, 1992.

[JØR 04] JØRGENSEN J., BOSSEN C., « Executable Use Cases : Requirements for a Pervasive Health Care System », *IEEE Software*, vol. 21, n°2, p. 34–41, mars 2004.

[MED 00] MEDVIDOVIC N., TAYLOR R. N., « A Classification and Comparison framework for Software Architecture Languages », *IEEE Transactions on Software Engineering*, vol. 147, n°6, p. 225–236, décembre 2000.

[MIL 80] MILNER R., « A Calculus of Communicating Systems », vol. 92 de *LNCS* , Springer, page165, 1980.

[MIL 92] MILNER R., PARROW J., WALKER D., « A Calculus for Mobile Processes », *Information and Computation*, vol. 100, n°1, p. 1–40, 1992.

[PET 03] PETRUCCI L., KRISTENSEN L. M., BILLINGTON J., QURESHI Z. H., « Developing a Formal Specification for the Mission System of a Maritime Surveillance Aircraft », *Proc. 3rd Int. Conf. on Application of Concurrency to System Design (ACSD'03), Guimarães, Portugal, June 2003*, IEEE Comp. Soc. Press, p. 92–101, 2003.

[REG 03] REGGIO G., ASTESIANO E., CHOPPY C., CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0 – Summary, Rapport n˚DISI-TR-03-36, Univ. of Genova, 2003.

[REI 91] REISIG W., « Petri Nets and Algebraic Specifications », *Theoretical Computer Science*, vol. 80, p. 1–34, 1991.

[ROG 04] ROGGENBACH M., MOSSAKOWSKI T., SCHRÖDER L., « CASL Libraries », [CoF 04], part V.

Chapitre 3

Démarches de spécification

Ce chapitre a pour but de donner des indications sur la manière de construire une spécification cohérente. Tout d'abord, dans la section 3.1, nous présentons des critères à prendre en compte pour que la spécification réponde aux attentes. L'objectif est alors, avant de commencer à construire la spécification, de se poser les questions pertinentes quant au type de modèles à utiliser, au niveau d'abstraction choisi, aux propriétés souhaitées. Ensuite, dans la section 3.2, nous introduisons une méthodologie de spécification.

3.1. Critères orientant le développement des spécifications

Avant de commencer à développer une spécification, il est nécessaire de s'interroger sur les points importants du système étudié. Cela permettra d'une part de choisir un formalisme de spécification adapté au problème, d'autre part de structurer le modèle en vue de l'enrichir par la suite en prenant en compte des détails supplémentaires. Enfin, le formalisme utilisé doit permettre la vérification des propriétés que le système devrait satisfaire.

Dans cette section, nous nous intéressons aux différents aspects à prendre en compte avant de commencer le développement d'une spécification. Nous présentons ensuite une étude de cas relativement simple illustrant cette approche.

Chapitre rédigé par Christine CHOPPY et Laure PETRUCCI.

3.1.1. *Concepts pertinents*

La première étape consiste à identifier les *concepts pertinents* dans le système à étudier. Les éléments clés constituant le système doivent être dégagés, permettant ainsi d'orienter le choix du formalisme à utiliser.

Types de données du système. Le problème étudié peut mettre en jeu des *données* plus ou moins complexes. Il s'agit alors de mesurer si le détail de ces données est nécessaire ou non. En effet, en général, plus le niveau de description est élevé, moins les techniques d'analyse associées au formalisme sont performantes.

EXEMPLE.– Soit l'étude d'un protocole de communication. On s'intéresse au bon fonctionnement des retransmissions en utilisant ce protocole. Le détail des messages échangés n'est pas nécessaire, alors que la nature des messages peut l'être. On pourra alors choisir de modéliser les messages avec uniquement leur type (request, indication, response, confirm) et leur numéro de séquence.

Lorsque des données complexes sont manipulées et doivent être modélisées, on s'orientera vers un formalisme tel que les spécifications algébriques ou des modèles de haut niveau comme les réseaux de Petri colorés. Au contraire, si les données sont peu importantes, mais si leur présence est nécessaire au bon fonctionnement du système, des modèles de plus bas niveau tels que les automates ou les réseaux de Petri sont à privilégier. Dans un cas intermédiaire, où les données sont peu typées et peuvent se ramener à des entiers ou des énumérations, les automates à compteurs ou les réseaux de Petri symétriques[1] sont plus adaptés.

Le temps. Parmi les systèmes critiques, les systèmes temps-réel incluent des contraintes *temporelles*. Celles-ci peuvent correspondre :

– à un *temps d'attente*, tel que l'expiration d'un délai de garde ;

– au *temps d'exécution* d'une action, comme le déplacement d'un robot ;

– etc.

Plusieurs extensions des automates et des réseaux de Petri permettent de prendre en compte le temps. Celles-ci seront détaillées dans le chapitre 4. Certaines considèrent qu'il existe une *horloge globale*, qui sert de référence à toutes les opérations possédant des contraines de temps. D'autres supposent qu'il n'y a pas d'horloge globale (les opérations peuvent alors avoir lieu pendant un intervalle de temps à partir du moment

1. Les réseaux de Petri symétriques étaient auparavant appelés *réseaux bien formés*. Le nom de ce type de réseaux a changé récemment, à l'occasion du processus de normalisation des réseaux de Petri (norme ISO/IEC 15909).

où toutes leurs préconditions sont satisfaites), ou que les différents composants du système ont leur propre *horloge locale*.

On distingue deux types de modélisation du temps :

– le *temps discret* correspond à l'observation d'un système à des tops d'horloge réguliers ou un échantillonage.

– le *temps continu* montre l'évolution du système sans considérer des tranches de temps. Ceci est particulièrement important lorsque l'on étudie un *système hybride* dans lequel les variables peuvent évoluer en fonction du temps (par exemple une accélération, une évolution de température, etc.).

La prise en compte du temps dans la spécification n'est pas toujours nécessaire. On peut en effet s'intéresser à des *propriétés qualitatives* pour lesquelles le temps n'intervient pas.

Il s'agit alors de déterminer si le temps doit être pris en compte dans la spécification, s'il doit être discret ou continu, si le modèle doit comporter une horloge globale, des horloges locales ou des intervalles de temps.

Il faudra également choisir entre les paradigmes de *modèle synchrone* et *modèle asynchrone* [LYN 96]. Les sous-systèmes d'un système synchrone évoluent simultanément, à la même cadence. Par contre, dans un modèle asynchrone, un sous-système peut évoluer pendant qu'un autre attend qu'un évènement particulier se produise.

Communication. Les systèmes complexes sont généralement composés de sous-systèmes communiquant entre eux. Les communications peuvent s'effectuer de manières :

– la communication par *rendez-vous* oblige les différentes parties à *se synchroniser*. L'un des sous-systèmes envoie un message qui est reçu simultanément par un ou plusieurs autres sous-systèmes. Si l'un d'eux n'est pas prêt à effectuer cette synchronisation, les autres l'attendent. Ce type de communication est modélisé explicitement dans les automates communicants et dans les réseaux de Petri.

– l'utilisation d'un *canal de communication* peut suivre plusieurs politiques : le canal peut être représenté par une *file de messages* suivant un ordre FIFO (*First In First Out*), ou un autre (auquel cas le déséquencement des messages est possible), avec ou sans *perte de messages*. Les automates à files ou les réseaux de Petri de haut niveau permettent de telles modélisations.

3.1.2. *Niveau d'abstraction*

Le développement d'une spécification ne s'effectue pas d'une seule traite. Une notion importante est le *niveau d'abstraction*. On commence la modélisation à un niveau

très abstrait, puis on procède par *construction incrémentale* en appliquant plusieurs étapes de *raffinement*. À chaque étape, le modèle est validé avant d'être raffiné. Le raffinement consiste à complexifier les données manipulées pour prendre en compte plus de détails, à augmenter la structure de l'automate ou du réseau de Petri, pour détailler une action complexe en la décomposant en actions plus élémentaires.

Une telle démarche de spécification part donc d'un modèle assez abstrait, où seulement les points clés du système sont modélisés, et, au fur et à mesure de la construction du modèle, de plus en plus de détails sont ajoutés.

EXEMPLE.– Reprenons l'exemple des compteurs modulo 3 introduit dans la section 2.3.2, page 18. Cet exemple, bien qu'extrêmement simple, pourrait être considéré comme un système à deux compteurs avec deux types d'opérations, l'une incrémentant un compteur d'une unité et l'autre opérant sur les deux, incrémentant le premier de 2 et le second de 1. Ceci conduit au modèle de la figure 3.1. Ce modèle peut ensuite être raffiné en celui de la figure 2.8.

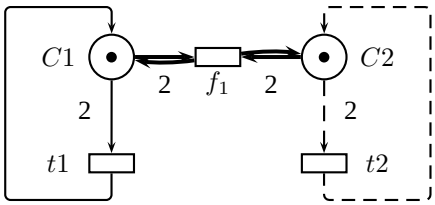


Figure 3.1. *Réseau de Petri abstrait représentant un système à 2 compteurs*

Considérer le « bon » niveau d'abstraction est d'autant plus important que la prise en compte d'éléments superflus complexifie la phase d'analyse du système.

De plus, comme lors du développement d'un programme, la construction pas à pas de la spécification permet de valider le modèle petit à petit. On s'assure donc que les parties étudiées ont un comportement correct avant de continuer la modélisation.

3.1.3. *Structuration de la spécification*

L'écriture d'une spécification pour un système complexe ne s'effectue pas d'un bloc. Par conséquent, il est nécessaire de disposer de mécanismes de structuration de modèles. Le système est généralement composé de plusieurs sous-systèmes qui interagissent. Une approche intéressante consiste à spécifier chacun de ces sous-systèmes de manière relativement indépendante, puis à spécifier les interactions entre eux. Cette approche présente plusieurs avantages :

– la conception de chaque sous-système se concentre sur le comportement de celui-ci. Le modélisateur ne s'encombre donc pas de considérations concernant les autres sous-systèmes.

– un sous-système peut avoir plusieurs variantes. Le système complet peut donc être analysé avec ces différentes variantes, en remplaçant seulement le modèle d'un sous-système par un autre.

– de manière similaire, un composant (sous-système) développé pour un modèle particulier peut être *réutilisé* au sein d'un autre système complexe. Il n'est alors pas nécessaire de spécifier à nouveau ce composant.

– plusieurs instances d'un même composant peuvent être utilisées, en utilisant un modèle paramétré, de manière à pouvoir repérer l'instance concernée, lors de l'analyse. Il n'y a donc pas à dupliquer le modèle du sous-système.

– cette approche permet d'utiliser des composants ou sous-systèmes génériques, issus de bibliothèques de composants fréquemment utilisés, avec un minimum d'adaptation.

EXEMPLE.– Supposons que l'on souhaite modéliser un protocole de communication. Un tel système comprend généralement un *émetteur*, un *récepteur* et un *medium de communication*. Ces trois composants peuvent être spécifiés indépendamment les uns des autres. On peut modéliser par exemple deux types d'émetteurs et vérifier le protocole avec chacun d'eux.

Certains formalismes de spécification disposent de concepts structurants intrinsèques ou intégrés. C'est par exemple le cas des réseaux de Petri colorés hiérarchiques [JEN 92] qui permettent de concevoir le modèle en utilisant plusieurs sous-modèles structurés selon un schéma arborescent. Au niveau le plus élevé (racine de l'arbre), se trouve une description abstraite du système, alors que les feuilles décrivent les divers éléments à un niveau détaillé.

3.1.4. *Propriétés*

L'écriture de propriétés est complémentaire au développement de la spécification. En effet, il faut d'une part décrire les propriétés dans un formalisme compatible avec avec la spécification du système, et d'autre part s'assurer que les éléments constituants de la propriété sont effectivement représentés dans le modèle. Par exemple, pour prouver la satisfaction de contraintes temps-réel, ou la réalisabilité d'un ordonnancement, il est nécessaire d'utiliser un modèle temporisé.

3.1.5. *Évolution future du modèle et des propriétés*

La spécification et les propriétés sont amenées à évoluer au cours de la conception et la vérification du système étudié. Ce paramètre doit également être pris en compte

pour garantir que l'on puisse atteindre le niveau d'abstraction souhaité pour le système.

EXEMPLE.– Les réseaux de Petri des figures 3.2, 3.3, 3.4 et 3.5 modélisent tous le problème de l'exclusion mutuelle entre 2 processus.

Tous ces modèles comportent une boucle représentant le fonctionnement d'un processus. Un processus n'est initialement pas en section critique (place P), peut demander à y entrer (transition *Entrée*), puis se trouve en section critique (place SC), dont il peut sortir (transition *Fin*) pour retourner dans son état initial. Un seul processus peut être en section critique, à tout moment. Par conséquent, l'entrée en section critique d'un processus est conditionnée par l'état de l'autre processus qui ne doit pas déjà se trouver en section critique. Ceci est modélisé par les arcs doubles dans la figure 3.2. Une façon classique de gérer l'accès à la section critique consiste à utiliser un « droit d'accès », modélisé par le jeton dans la place *EXMU* des autres versions de réseau de Petri. Dans la figure 3.3, la boucle effectuée par chaque processus est explicitée. Par contre, dans la figure 3.4, les deux cycles sont confondus, et les deux processus représentés par les deux jetons dans la place P. La figure 3.5 est similaire, mais utilise un réseau coloré, avec les processus nommés $p1$ et $p2$.

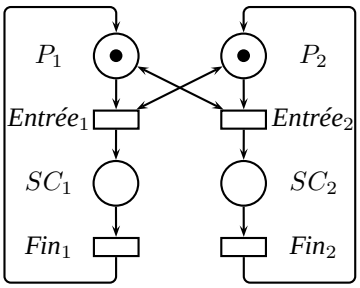


Figure 3.2. *Exclusion mutuelle entre 2 processus (Version 1)*

Considérons maintenant une première optique pour le développement de cette spécification : on souhaite étendre le modèle pour prendre en compte l'exclusion mutuelle entre 3 processus. Le réseau de la figure 3.2 doit alors comprendre un nouveau cycle pour le nouveau processus. La transition *Entrée*$_3$ doit être reliée par un arc double aux places P_1 et P_2, de même que les transitions *Entrée*$_1$ et *Entrée*$_2$ à la place P_3. Le modèle devient alors peu compréhensible. De même, le réseau de Petri de la figure 3.3 doit comprendre une nouvelle boucle pour le troisième processus, connectée à la place *EXMU*. Ceci devient également peu lisible. L'extension des autres modèles se fait en rajoutant simplement un jeton correspondant au troisième processus.

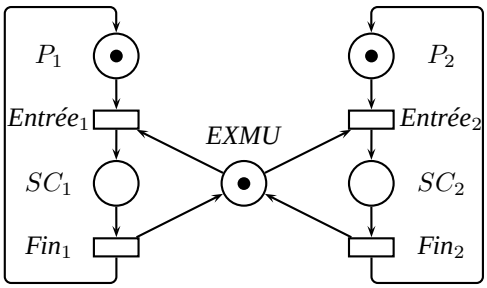


Figure 3.3. *Exclusion mutuelle entre* 2 *processus (Version 2)*

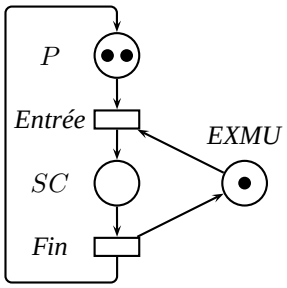


Figure 3.4. *Exclusion mutuelle entre* 2 *processus (Version 3)*

Supposons que l'on souhaite maintenant détailler le comportement des processus en section critique. Seul le réseau de Petri de la figure 3.5 permet de distinguer les différents processus en jeu, tout en gardant un modèle compact (le réseau de la figure 3.3 permet également de distinguer les identités des processus).

De même si l'on souhaite vérifier qu'un processus peut entrer en section critique, et ce quelle que soit son identité, les réseaux des figures 3.3 et 3.5 sont les plus appropriés. On constate donc, que même pour un exemple extrêmement simple, il faut, lors de la conception du modèle, anticiper sur ses évolutions futures ou la vérification de propriétés.

3.1.6. *Utilisation d'outils de spécification et vérification*

De nombreux outils (par exemple [AMI, CPN, HEN 95, LAR 97]) permettent de construire des modèles formels et fournissent des techniques d'analyse. Toutefois, le type de modèle considéré et de techniques d'analyse disponibles diffèrent d'un outil à

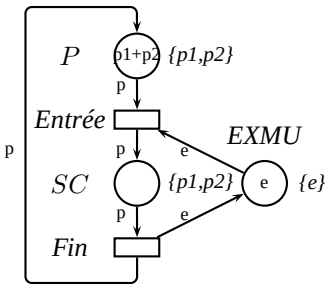


Figure 3.5. *Exclusion mutuelle entre* 2 *processus (Version 4)*

l'autre. Le choix d'utilisation d'un outil doit donc considérer les aspects à la fois spécification et vérification. Une démarche de normalisation est en cours [KIN 05] dans le cadre des réseaux de Petri. Elle vise notamment à développer un langage normalisé de description des réseaux de Petri. Ceci permettra alors d'échanger des fichiers entre outils et d'utiliser une plus large palette de techniques d'analyse.

3.1.7. *Étude de cas : modélisation d'un train électrique*

Dans cette section, nous considérons un exemple plus complexe de modélisation d'un train électrique. Cet exemple est issu d'un projet destiné à des étudiants [BER 01]. L'objectif est de spécifier et analyser formellement un système de train électrique avant de passer à une implémentation pilotant un dispositif matériel. Il est donc impératif de s'assurer de l'absence de collisions avant de mettre en œuvre la solution finale.

Le modèle physique du train électrique est représenté dans la figure 3.6. Il comprend environ $15m$ de voies divisées en 16 sections (blocs B1 à B16), plus 2 voies de garage (ST1 et ST2), connectées par 4 aiguillages et 1 croisement. Le dispositif physique permet de faire circuler plusieurs trains. Il est relié à un ordinateur pouvant lire de l'information sur les voies et transmettre des commandes aux trains (stop, avancer ou reculer) et aiguillages. Une section de voie est équipée à chaque extrémité d'un capteur permettant de détecter l'entrée ou la sortie du train.

Le modèle, pour pouvoir représenter la circulation de plusieurs trains, peut devenir rapidement complexe. C'est pourquoi, le choix du modèle s'oriente vers un langage de spécification de haut niveau. De plus, on peut s'intéresser, dans un premier temps, à une modélisation assez abstraite du système, proche de la réalité. Dans un second temps, les détails du passage d'une section de voie à sa voisine seront pris en compte. Par conséquent, le choix de modèle se porte sur les réseaux colorés hiérarchiques. La figure 3.7 montre une modélisation abstraite du système, « collant » à la description physique des voies. Il est alors facile de faire le lien entre le réseau de Petri et le plan.

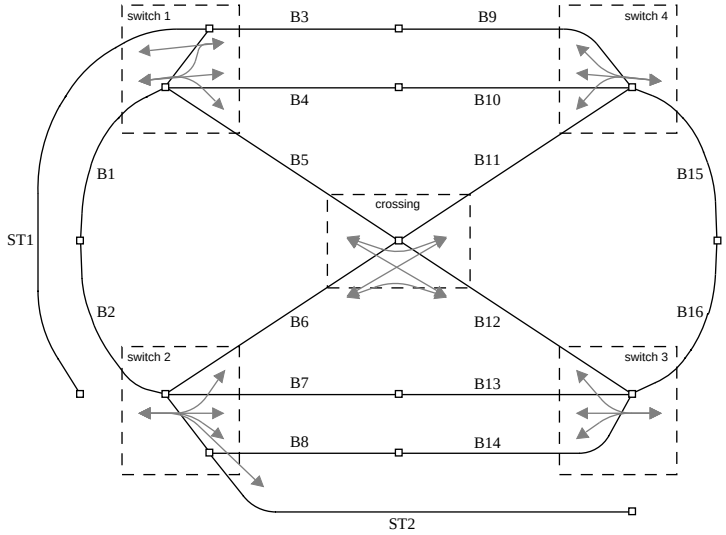


Figure 3.6. *Le plan des voies d'un train électrique*

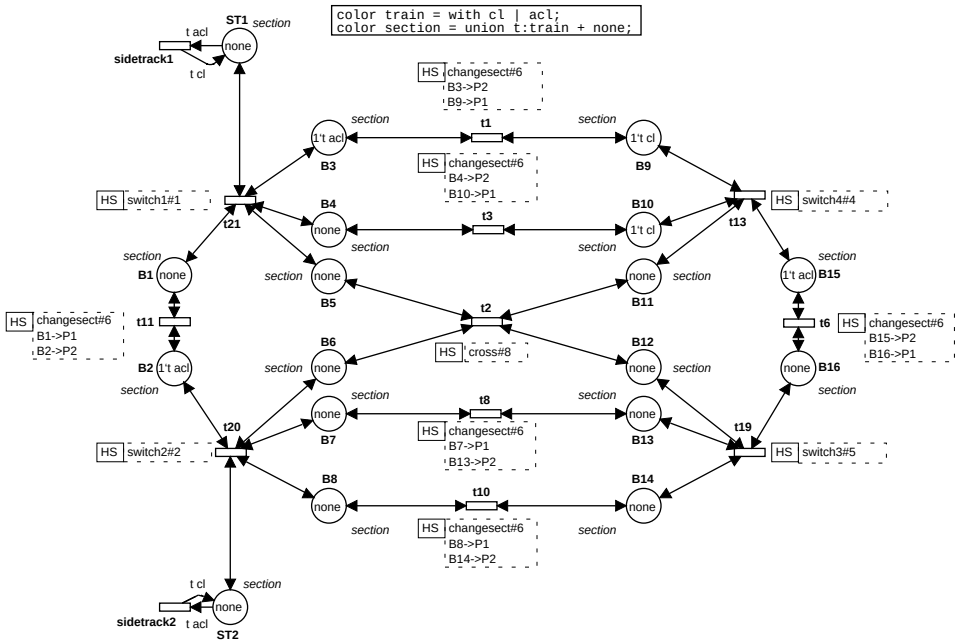


Figure 3.7. *Réseau de Petri abstrait modélisant le train électrique*

La politique de passage d'un tronçon de voie au suivant peut ensuite être détaillée : chaque transition du réseau de la figure 3.7 est alors une transition de substitution, c'est-à-dire que son fonctionnement est décrit par un sous-réseau. La figure 3.8 représente une politique de passage d'un tronçon au tronçon voisin, sans aiguillage. Ce sous-réseau est utilisé pour expliciter le fonctionnement des transitions $t1$, $t3$, $t6$, $t8$, $t10$ et $t11$. Les sous-réseaux correspondant aux aiguillages et au croisement sont détaillés dans [BER 01].

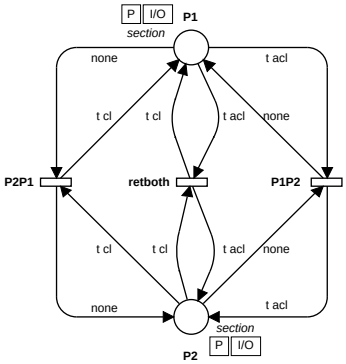


Figure 3.8. *Réseau de Petri modélisant le passage d'un tronçon au suivant*

Examinons maintenant les données mises en œuvre dans cette modélisation. Le réseau ferré est représenté par des places du réseau de Petri. Chaque tronçon peut contenir aucun train, ou un train (et un seul). Le type des places reflète donc ces possibilités. Il est nécessaire, pour chaque train, de savoir dans quel sens il se dirige, d'où une direction (*clockwise* ou *anticlockwise*, pour le sens horaire et le sens antihoraire, respectivement). Il est de plus tentant de nommer chaque train, pour savoir à un moment donné, où se trouve un train particulier. Toutefois, cette information est superflue. En effet, pour le bon fonctionnement du système, il suffit de savoir s'il y a un train sur un tronçon, et sa direction. Le nommage de trains est possible, mais induit des problèmes : lorsque l'on veut passer à la vérification, on se heurte au problème de l'explosion combinatoire de l'espace d'états (à moins d'utiliser des techniques symboliques, voir chapitre 8). Une approche sans nommage des trains permet d'éviter cet écueil.

3.2. Méthodes de développement des spécifications

La question que nous nous posons ici est « comment démarrer et développer une spécification ? ». Il est clair que, lorsqu'il s'agit d'un problème complexe, on ne peut pas démarrer la spécification avant d'avoir eu les moyens de le structurer, et de le décomposer, en utilisant dans un premier temps des structures assez larges qui seront

affinées par la suite. Un facteur commun des méthodes de développement considérées est donc qu'elles permettent d'effectuer, d'une manière ou d'une autre, une décomposition du système étudié. Souvent une première décomposition très grossière peut être obtenue à partir de cas d'utilisation UML. Il s'agit ensuite de poursuivre cette structuration, par exemple à l'aide de *patterns*, et de disposer de guides pour aller jusqu'à une spécification formelle précise.

Les *patterns* (traduits par *schémas* ou *patrons*) proposent des familles de structures fréquemment rencontrées que l'utilisateur est invité à « essayer » (quitte à les adapter) sur le problème à traiter, pour ainsi bénéficier de concepts structurants en « prêt-à-porter ». Les *patterns* peuvent donc être vus comme un moyen élaboré de réutiliser des connaissances acquises par l'expérience. Les *problem frames* [JAC 01] (que nous traduirons par *schémas de problèmes*) sont proposés par M. Jackson pour structurer les problèmes de manière globale. Les *styles d'architecture* [GAR 93, BAS 98] offrent des structures de granularité plus fine qui sont souvent utilisables au niveau de la conception. L'approche d'*architecture par composants* [CHE 01] permet de préciser comment les composants, que la structuration propose de développer indépendamment, seront finalement intégrés. Il peut être souhaitable de combiner ces approches, pour bénéficier de leurs avantages conjugués [CHO 04c].

Une fois qu'un niveau de décomposition suffisant a été atteint, la spécification précise du système peut être abordée, et des guides sont utiles pour le faire de manière méthodique et couvrant autant que possible les différents points pertinents. Une approche présentée dans la section 3.2.1 s'appuie sur une distinction entre structures de données, système dynamique simple, et système dynamique structuré (en sous-systèmes), et propose pour chacun les éléments caractéristiques et les propriétés à rechercher. Les méthodes développées ensuite s'appuient sur les schémas de problème, les styles d'architecture, ainsi que le développement de spécification à base de composants et la composition des éléments solution.

3.2.1. *Lignes guide pour le développement de spécifications*

Dans cette approche [CHO 06], un élément logiciel peut être :

– un système dynamique simple (par exemple un processus séquentiel) ;

– un système dynamique structuré (comportant des sous-systèmes en interaction, ces éléments étant eux-mêmes simples ou structurés) ;

– une structure de données.

Les éléments logiciels sont caractérisés par leurs *parties* et leurs *caractéristiques constitutives*. Par exemple, les parties d'un système simple sont des structures de données, et ses caractéristiques constitutives sont ses états et les interactions élémentaires. Un système dynamique (simple ou complexe) est considéré comme un système de transition étiqueté (« *labelled transition system* » lts), où lts est un triplet

$(\mathit{State}, \mathit{Label}, \rightarrow)$, qui comporte l'état *State* du système, l'étiquette *Label* (composée d'évènements) d'une transition, et la relation de transition $\rightarrow$, avec $\rightarrow \subseteq \mathit{State} \times \mathit{Label} \times \mathit{State}$.

Des guides précis indiquent également quelles propriétés doivent être exprimées et de quelle manière.

Il est aussi possible de choisir un style de spécification, par exemple un style *orienté propriétés* (aussi appelé axiomatique), ou un style *orienté modèle* (ou encore constructif). Le style orienté propriétés est plus centré sur la description des propriétés du système à un niveau abstrait (accompagné de la possibilité d'effectuer des preuves des conséquences de ces propriétés), tandis que le style orienté modèle est tourné vers la possibilité d'effectuer des calculs (et de la vérification par modèle ou « *model checking* »).

Quel que soit le style choisi, il est toujours recommandé d'adopter une représentation graphique qui facilite la lecture de la spécification formelle.

Cette méthode a été développée initialement pour les langages cibles CASL [BID 04] et CASL-LTL [REG 03] (cf. section 2.3.1), mais elle est applicable à d'autres langages (par exemple, les réseaux de Petri [CHO 04a] ou UML [CHO 04b]). Nous détaillons ci-après le traitement des systèmes dynamiques simples.

Systèmes simples. Nous esquissons ici une description de la méthode pour une spécification orientée propriétés des systèmes simples, que nous illustrons ensuite sur l'exemple du train décrit à la section 3.1.7.

Un système simple est un système sans composants internes, ce peut être un processus séquentiel/non déterministe ou un système distribué/parallèle. Comme indiqué plus haut, les systèmes simples sont considérés formellement comme des systèmes de transition étiquetés (lts). Les états s du lts représentent les situations intermédiaires au cours de la vie du système, et chaque transition représente la capacité d'évoluer de l'état s à l'état s' du système, et l'étiquette l contient l'information relative aux conditions à satisfaire pour que cette transition soit possible et aux transformations induites par l'exécution de transition.

Les étiquettes ont la forme standard d'un ensemble d'interactions élémentaires. Chaque interaction élémentaire correspond à un échange élémentaire avec l'environnement externe. Les interactions élémentaires ont aussi des types différents, et un type est caractérisé par un nom et des arguments (éléments de structures de données). Elles font donc partie des caractéristiques du système simple.

La forme de l'état est aussi une caractéristique du système simple, et selon le style de spécification adopté, elle sera décrite au moyen d'observateurs de l'état (pour une

spécification orientée propriétés) ou de constucteurs de l'état (pour une spécification orientée modèle).

Enfin les structures de données utilisées par les descripteurs d'état et par les interactions élémentaires sont les parties du système.

Toutes les propriétés du système simple correspondent aux propriétés sur lts c'est-à-dire sur les étiquettes, les états et les transitions. Ces propriétés peuvent exprimer quels sont les ensembles admissibles d'interactions élémentaires pour construire une étiquette de transition et elles relient l'état source, l'étiquette et l'état cible d'une transition. Les propriétés peuvent aussi fournir des informations sur les valeurs observées par les différents observateurs d'état. Plus précisément :

1) Les propriétés d'étiquette expriment que deux interactions élémentaires différentes sont incompatibles sous certaines conditions.

2) Les propriétés d'état décrivent les conditions que les valeurs retournées par les observateurs d'état devraient satisfaire pour n'importe quel état. Les formules d'état peuvent aussi inclure des atomes spéciaux qui expriment les propriétés sur les chemins (suites de transitions) quittant/arrivant de l'état (donc sur le comportement futur/passé du système).

3) Les propriétés de transition sont des conditions sur les observateurs d'état, appliqués aux états source et aux états cible de la transition.

Les propriétés doivent être décrites de manière informelle à l'aide d'un commentaire en langage naturel et de manière formelle. Une liste complète des propriétés est disponible [CHO 06], elles seront illustrées plus loin dans l'exemple de la spécification du train. Ci-dessous des exemples de propriétés sont détaillés : la précondition d'une interaction élémentaire, et ses incompatibilités avec elle-même ou avec d'autres interactions élémentaires.

pre-cond1 (TransitionProp) Si l'étiquette d'une transition contient une instance de ei, alors l'état source de la transition doit satisfaire une condition :

if $cond(args)$ **then** $ei(args)$ **happens**
Certains observateurs d'état source doivent apparaître dans $cond(args)$ et les observateurs d'état cible ne peuvent y figurer.

incompat1 (LabelProp) Sous certaines conditions (portant sur les arguments), une instance de ei est incompatible avec une autre instance de ei (aucune étiquette de transition ne peut les contenir simultanément) :

ei(*arg*$_1$) **incompatible with** ei(*arg*$_2$) **if** $cond$(*arg*$_1$,*arg*$_2$)

Sous certaines conditions (portant sur les arguments), une instance de ei est incompatible avec une instance de ei' (aucune étiquette de transition ne peut les contenir simultanément) :

$ei(arg_1)$ **incompatible with** $ei'(arg_2)$ **if** $cond(arg_1, arg_2)$

Dans la section suivante, ces idées sont utilisées pour guider la spécification d'une partie de l'exemple du train décrit à la section 3.1.7.

Spécification du train. Pour illustrer les idées décrites ci-dessus, nous considérons le train comme un système dynamique simple.[2] Il s'agit donc d'identifier les caractéristiques de ce système, sous forme d'observateurs d'état et d'interactions élémentaires, qui sont représentés dans la figure 3.10.

La description de l'état du système s'appuie tout d'abord sur la disposition du système physique, c'est-à-dire des voies sur lesquelles vont circuler les trains, ainsi que des aiguillages. Les observateurs d'état doivent permettre de savoir quelles voies sont contiguës (« *connected* ») ou reliées par un aiguillage (« *switched* »), et aussi de savoir si un train est présent sur une section de voie (« train_present ») et quelle est sa direction de trajet (ici simplement dans le sens horaire « *clockwise* », ou anti-horaire « *anticlockwise* »).

Les interactions élémentaires (associées à un changement d'état du train) sont donc les changements de section de voie par un train, que ces voies soient contiguës ou reliées par un aiguillage. Les propriétés des observateurs d'état refléteront bien sûr que la position des voies est fixe, ainsi que les connections potentielles d'un aiguillage.

Les interactions élémentaires et observateurs d'état identifiés doivent être nommés de manière pertinente, il faut également déterminer quels sont les types de leurs arguments, et ces types de données doivent être également spécifiés.

Les structures de données qui constituent les parties du système sont identifiées et élaborées pendant le travail sur les interactions élémentaires et observateurs d'état. En ce qui concerne les sections de voie et les aiguillages, leurs noms sont donnés dans le plan des voies de la figure 3.6, et ce sont aussi les valeurs possibles des types associés. Ces valeurs sont en nombre réduit, et les types sont donc définis par l'énumération de ces valeurs (cf. figure 3.9 et la spécification CASL ci-après). La direction de voyage des trains comporte de même deux valeurs possibles. La spécification en langage CASL de ces données est donnée ci-dessous. La construction **free** assure qu'aucune propriété

2. Bien sûr, dans un travail plus complet prenant en compte la politique de passage de tronçon de voie, il pourrait s'agir d'un système structuré.

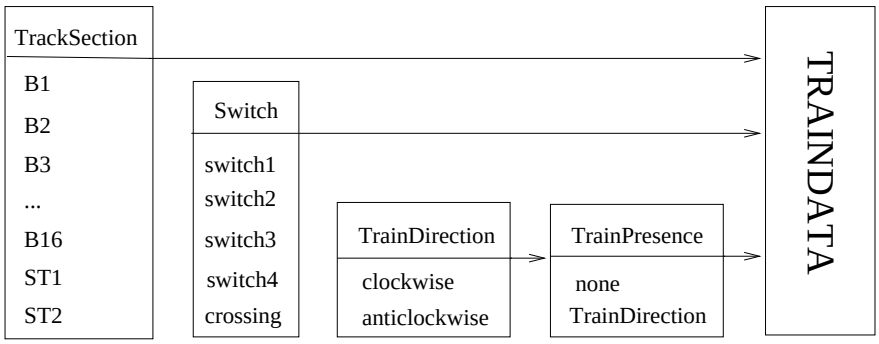


Figure 3.9. *Les structures de données*

ne lie ces valeurs et qu'elles sont donc toutes distinctes. La construction **sort** est utilisée ici pour exprimer que tout élément du type $TrainDirection$ est aussi du type $TrainPresence$.

spec TRAINDATA =
  free type
    $TrackSection ::= B1 \mid B2 \mid B3 \ldots \mid B16 \mid ST1 \mid ST2$
  free type
    $Switch ::= switch1 \mid switch2 \mid switch3 \mid switch4 \mid crossing$
  free type
    $TrainDirection ::= clockwise \mid anticlockwise$
  free type
    $TrainPresence ::= none \mid$ **sort** $(TrainDirection)$
  end

On peut remarquer que ces données sont reflétées dans les noms des états et des transitions du réseau de Petri, ou dans les couleurs des jetons (cf. figure 3.6). La présence d'un train sur une section de voie évolue au cours de l'histoire du système, en fonction d'une part de la disposition à l'état initial du système, et d'autre part des changements de section de voie qui ont eu lieu depuis. Il est donc nécessaire d'introduire une notion d'histoire du système « History » et d'état « State ».

spec STATE =
  sort $State$;
  op $initial : State$;
  end

spec HISTORY = STATE **then**
  type $History ::= initial \mid \_\_.\_\_(History; State)$;
    $\ldots$
  end

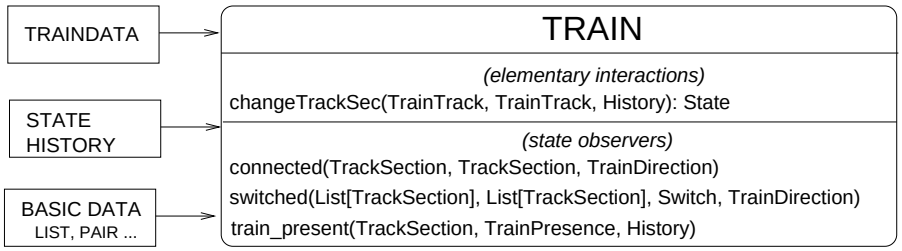


Figure 3.10. *Les interactions élémentaires et observateurs d'état pour le train*

La construction **then** introduit les éléments ajoutés après l'importation du module de spécification STATE.

Une fois que les caractéristiques constituantes sont identifiées et que les parties du système sont spécifiées, les propriétés du système doivent être exprimées. Les propriétés relatives à l'interaction élémentaire $changeTrackSec$ sont une pré-condition, une post-condition et une propriété d'incompatibilité.

pre-cond1 (propriété de transition) Le changement de section de voie est possible lorsque deux sections de voie sont connectées (ou reliées par un aiguillage), quand il y a un train sur la section de départ en direction de cette connexion (ou cet aiguillage), et pas de train sur la section d'arrivée :

if $(connected(TS_i, TS_j, TP_i) \vee$
$\exists sw : Switch\ t.q.\ swichted((\ldots, TS_i, \ldots), (\ldots, TS_j, \ldots), sw, TP_i)) \wedge (TP_j = none)$
then $changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j >)$ **happens**

post-cond1 (propriété de transition) Après qu'un changement de section de voie se soit produit, le train est dans la section de voie d'arrivée :

if $changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j >)$ **happens then** $(TP'_j = TP_i)$

incompat1 (propriété sur l'étiquette) L'idée ici est d'exprimer qu'un train ne peut changer simultanément entre plusieurs sections de voie à un moment donné de l'histoire h du système. Comme la direction de voyage du train est connue, la seule possibilité à écarter est qu'à un aiguillage un train puisse prendre simultanément plusieurs directions différentes :

$changeTrackSec(< TP_i, TS_i >, < TP_j, TS_j, h >)$ **incompatible with**
$changeTrackSec(< TP_i, TS_i >, < TP_k, TS_k, h >)$
if $\exists sw : Switch\ t.q.\ swichted((\ldots, TS_i, \ldots), (\ldots, TS_j, \ldots, TS_k, \ldots), sw, TP_i) \wedge (T_j \neq T_k)$

Les propriétés sur les observateurs d'état $connected$ et $switched$ reflètent le plan de voie (qui ne change jamais) :

value1 (propriété sur l'état)

$connected(B1, B2, anticlockwise)$
$connected(B2, B1, clockwise)$
...
$switched((ST1, B1), B3), switch1, clockwise)$
$switched((B1), (B3, B4, B5), clockwise)$
$switched((B3), (B1, ST1), anticlockwise)$
$switched((B3, B4, B5), (B1), switch1, anticlockwise)$
...

Les propriétés sur l'observateur $train\_present$ portent d'une part sur les valeurs obtenues à l'état initial (par exemple des trains sont ou non positionnés sur certaines voies), et d'autre part sur la manière dont ces valeurs évoluent lorsque des interactions élémentaires (ici $changeTrackSec$) ont lieu.

value1 (propriété sur l'état) $train\_present(B1, none, initial)$ ...

how-change (propriété de transition)

if $train\_present(TS_i, TP_i, h) \wedge train\_present(TS_j, none, h)$
$\wedge\, changeTrackSec(< TS_i, TP_i >, < TS_j, none >, h)$ **happened**
then $(TP_i \neq none) \wedge train\_present(TS'_i, none, h') \wedge train\_present(TS'_j, TP_i, h')$

où h et h' dénotent respectivement l'histoire du système avant et après le changement de section de voie.

On peut bien sûr faire le lien entre cette spécification exprimée en CASL-LTL et une spécification exprimée sous forme de réseau de Petri. Cette étape de spécification peut être suivie d'autres étapes où des détails supplémentaires sont introduits (on parle alors de *raffinement* de la spécification).

Par ailleurs cette spécification peut être utilisée (par exemple par importation) pour construire une spécification de système structuré, ou pour d'autres systèmes. Nous décrivons dans ce qui suit les schémas de problème et nous montrons, en l'illustrant sur l'exemple du train, comment la spécification que nous venons de développer trouve sa place dans un cadre plus large.

3.2.2. *Schémas de problème*

Un *schéma de problème* [JAC 01] (ou *problem frame*) est un schéma qui définit de manière intuitive une classe de problèmes identifiée en termes de son contexte et des caractéristiques de ses domaines, de ses interfaces et des besoins. Le système à développer est représenté par la « machine ».

Pour chaque schéma de problème un diagramme est établi (tel celui de la figure 3.11). Les rectangles simples dénotent les domaines de l'application (qui existent déjà), les rectangles avec une double barre dénotent les domaines « machine » qui sont à réaliser, et les besoins sont notés par un ovale pointillé. Les lignes qui les relient représentent des interfaces, aussi appelées *phénomènes partagés*. Jackson distingue les domaines *causaux* qui obéissent à certaines lois, les domaines *lexicaux* qui sont des représentations physiques des données, et les domaines *« biddable »* qui sont des personnes. Jackson définit cinq schémas de base (*Required Behaviour, Commanded Behaviour, Information Display, Workpieces* et *Transformation*), et nous présentons dans la figure 3.11 le schéma contrôle-commande (*Commanded Behaviour*) qui propose une structure pour les problèmes où un domaine doit être contrôlé selon les commandes d'un opérateur.

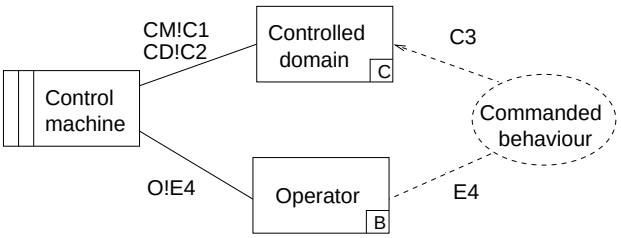


Figure 3.11. *Schéma de problème de contrôle-commande*

Le `C` indique que le domaine contrôlé (« *Controlled domain* ») est causal, et `CM!C1` indique que le phénomène `C1` est contrôlé par la machine (CM « *Control machine* »). Le trait pointillé représente une référence aux besoins, et la flèche indique que c'est une référence contraignante.

L'utilisation d'un schéma de problème consiste à instancier les domaines, les interfaces et les besoins, selon le problème à traiter. On cherchera donc un schéma de problème qui corresponde au problème considéré. Les schémas de base fournis par Jackson sont très simples, aussi en général il faut soit avoir effectué une première décomposition (par exemple à l'aide de cas d'utilisation UML) qui permette de trouver une correspondance, soit adapter des schémas existants pour obtenir cette correspondance, ou encore utiliser des schémas plus élaborés obtenus par combinaison de plusieurs schémas. La nature des domaines (lexicaux, causaux, « biddable ») est une clé importante pour déterminer le schéma de problème approprié.

Une fois qu'une structuration est trouvée, l'écriture de la spécification se fera pour les différents éléments de la structure (domaines, machine, et interfaces).

Pour l'exemple du train électrique décrit dans la section 3.1.7, le schéma de problème contrôle-commande ne peut être utilisé car il n'est pas fait référence à un utilisateur du train. On peut penser que le train a un fonctionnement automatique à partir du moment où il est mis en marche. Le schéma de problème « *Required Behaviour* » (où l'utilisateur est absent) est plus adapté. Ce schéma (cf. figure 3.12) correspond à toutes les applications pour lesquelles un fonctionnement automatique prévaut, et où l'être humain intervient rarement (pour la mise en route, et quelques réglages).

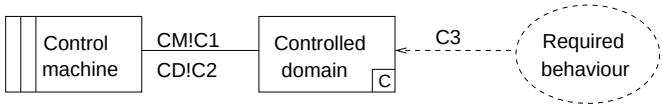


Figure 3.12. *Schéma de problème « Required Behaviour »*

La spécification associée à ce schéma de problème fait intervenir la spécification du domaine contrôlé, puis celle du comportement requis et enfin de la conception du système qui assurera ce comportement. Dans notre exemple du train, le domaine contrôlé est composé des trains qui circulent sur le plan de voie. Le comportement souhaité doit être décrit, par exemple en imposant des règles de circulation sur les voies (un train avance tant que c'est possible sans collision). Pour assurer ce comportement, le système doit avoir le moyen de connaître l'état du domaine contrôlé (dans la pratique, ceci est souvent assuré au moyen de capteurs) et il doit aussi avoir le moyen de modifier cet état (au moyen d'activateurs). Dans l'exemple du train, $train\_present$ peut être une information transmise par un capteur, et $changeTrackSection$ un évènement contrôlé par le système.

Dans ce que nous avons vu jusqu'ici seule intervient la structure induite par la nature et les éléments du problème à résoudre. Lorsqu'une description prenant en compte une structure globale du système à implémenter est souhaitée, on peut se tourner vers les schémas fournis par les styles d'architecture, soit en première intention si la nature du problème est bien connue, soit après avoir travaillé tout d'abord sur la spécification du problème (il peut alors s'agir d'une approche de raffinement).

3.2.3. *Styles d'architecture*

Les styles d'architecture [GAR 93, BAS 98] sont des schémas d'architectures logicielles, il s'agit donc *a priori* de structures moins abstraites que les schémas de problème et qui ne seront donc pas utilisées en première intention, mais pour un niveau de développement (et de spécification) proche de la conception. Les styles d'architecture sont caractérisés par :

– un ensemble de types de composants (par exemple répertoire de données, processus, . . .) qui effectuent certaines fonctions à l'exécution,

– une répartition topologique de ces composants indiquant leurs relations à l'exécution,

– un ensemble de contraintes sémantiques,

– un ensemble de connecteurs pour la communication, la coordination ou la coopération entre composants.

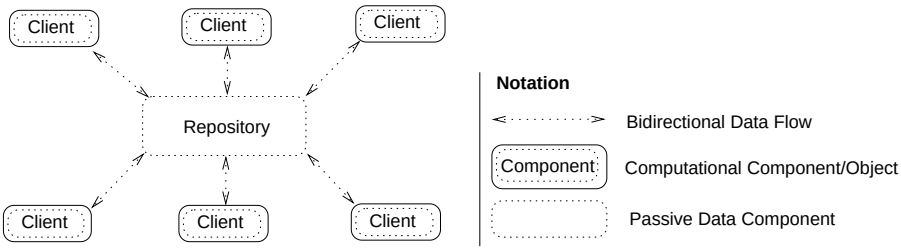


Figure 3.13. *Style d'architecture centré sur les données*

Parmi les principaux styles d'architecture, le style « *repository* » (voir figure 3.13), centré sur les données, où différents clients accèdent à des données partagées, convient bien pour les systèmes d'information.

Comme pour les schémas de problème, il faut fournir une instance d'un style d'architecture. Ce style sera choisi en fonction du type de problème considéré, mais souvent cela laisse plusieurs choix possibles. Les spécifications non fonctionnelles peuvent alors rentrer en ligne de compte. Il est aussi possible de guider ce choix à partir du (des) schéma(s) de problème retenu(s) dans une première étape.

La spécification formelle associée prendra en compte tous les éléments de l'architecture choisie.

3.2.4. *Spécification à base de composants*

En ingénierie du logiciel basée sur les composants [SZY 99] l'idée de base est de construire des logiciels à partir de parties de logiciel pré-fabriquées, qui sont bien encaspulées et relativement indépendantes. Ces parties de logiciel sont aussi nommées « composants », mais, dans ce contexte, les *composants* doivent satisfaire les conditions suivantes :

– Tous les services rendus et tous les services requis par un composant sont accessibles uniquement via des *interfaces* bien définies.

– Un composant adhère à un *modèle de composants*. Ce modèle spécifie entre autres des conventions syntaxiques pour la définition des interfaces et la manière dont les composants communiquent. Il sert à garantir l'interopérabilité de plusieurs composants qui adhèrent au même modèle de composant.

– Les composants sont intégrés sous forme binaire et le code source n'est souvent pas accessible au « client » de composants. Il est donc important que la spécification d'un composant contienne toutes les informations nécessaires à son utilisation.

Dans l'approche de l'ingénierie du logiciel basée sur les composants, les principes architecturaux jouent aussi un rôle important, car la composition d'un système à partir des composants est effectuée selon une architecture choisie. Les deux domaines sont donc assez fortement liés.

L'idée est ici de décomposer un problème en sous-problèmes. Pour chaque sous-problème, un logiciel doit être développé, et le système entier consiste en une combinaison appropriée de ces logiciels qui résolvent les sous-problèmes. Chaque sous-problème est réalisé sous forme de composant qui doit respecter des contraintes relatives aux interfaces. La combinaison des différents composants peut se faire à l'aide d'une *architecture de composants*, comme par exemple celle proposée par Cheesman et Daniels [CHE 01].

3.3. Conclusion

Le développement de la spécification formelle d'un système est une étape cruciale pour la bonne suite du développement du système. Si nous voyons dans ce chapitre que cette spécification pose de nombreuses questions, cela ne doit pas pour autant conduire à penser qu'il s'agit là d'une entreprise insurmontable. En effet, les différents points abordés ici ont pour but d'aider à cerner assez rapidement et de manière précise la nature du problème étudié, et aussi d'éclairer le point de vue avec lequel on souhaite effectuer ce travail (par exemple, le choix d'un niveau d'abstraction, le choix d'éléments à prendre en compte immédiatement ou à laisser en suspens jusquà l'implémentation, déterminer ce qui est du ressort du système et ce qui ne l'est pas).

Nous avons vu également comment une spécification peut évoluer, par exemple dans le cas de l'exclusion mutuelle.

Enfin nous avons montré comment les schémas ou patterns peuvent être utilisés également au niveau de la spécification, et nous avons fourni des guides d'écriture de spécification qui indiquent des éléments à rechercher et des propriétés à exprimer.

3.4. Bibliographie

[AMI] CPN-AMI : home page, `http://www-src.lip6.fr/logiciels/mars/CPNAMI/`.

[BAS 98] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice*, Addison-Wesley, 1998.

[BER 01] BERTHELOT G., PETRUCCI L., « Specification and Validation of a Concurrent System : An Educational Project », *Journal of Software Tools for Technology Transfer*, vol. 3, n°4, p. 372–381, 2001.

[BID 04] BIDOIT M., MOSSES P. D., CASL *User Manual*, Lecture Notes in Computer Science 2900 (IFIP Series), Springer, 2004, With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[CHE 01] CHEESMAN J., DANIELS J., *UML Components – A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

[CHO 04a] CHOPPY C., PETRUCCI L., « Towards a methodology for modelling with Petri nets », *Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark*, p. 39–56, octobre 2004, Proceedings published as Report DAIMI-PB 570, Aarhus, DK.

[CHO 04b] CHOPPY C., REGGIO G., « Improving Use Case Based Requirements Using Formally Grounded Specifications », *Fundamental Approaches to Software Engineering (FASE)*, Lecture Notes in Computer Science 2984, Springer, p. 244-260, 2004.

[CHO 04c] CHOPPY C., HEISEL M., « Une approche à base de "patrons" pour la spécification et le développement de systèmes d'information », *Proceedings Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2004*, p. 61–76, 2004.

[CHO 06] CHOPPY C., REGGIO G., « A Formally Grounded Software Specification Method », *Journal of Logic and Algebraic Programming*, vol. 67, n°1-2, p. 52–86, Elsevier, 2006.

[CPN] CPNTOOLS, `http://wiki.daimi.au.dk/cpntools`.

[GAR 93] GARLAN D., SHAW M., « An Introduction to Software Architecture », AMBRIOLA V., TORTORA G., Eds., *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific Publishing Company, 1993.

[HEN 95] HENZINGER T. A., HO P., WONG-TOI H., « HYTECH : the Next Generation », *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, IEEE Computer Society Press, p. 56–65, 1995.

[JAC 01] JACKSON M., *Problem Frames. Analyzing and structuring software development problems*, Addison-Wesley, 2001.

[JEN 92] JENSEN K., *Coloured Petri Nets : Basic concepts, analysis methods and practical use. Volume 1 : basic concepts*, Monographs in Theoretical Computer Science, Springer, 1992.

[KIN 05] KINDLER E., « Software and Systems Engineering - High-level Petri Nets. Part2 : Transfert Format. Working Draft for the International Standard ISO/IEC 15909 Part 2 - Version 0.9.0 », June 2005.

[LAR 97] LARSEN K. G., PETTERSSON P., YI W., « UPPAAL in a Nutshell », *Journal of Software Tools for Technology Transfer*, vol. 1, n°1–2, p. 134-152, Springer, 1997.

[LYN 96] LYNCH N., *Distributed Algorithms*, Series in Data Management Systems, Morgan Kaufmann, 1996.

[REG 03] REGGIO G., ASTESIANO E., CHOPPY C., CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0 – Summary, Rapport n˚DISI-TR-03-36, Univ. of Genova, 2003.

[SZY 99] SZYPERSKI C., *Component Software - Beyond object oriented programming*, Addison Wesley, 1999.

Chapitre 4

Modèles temporisés

Ce chapitre est consacré aux problèmes de modélisation, en vue de la conception d'applications qui font intervenir des contraintes quantitatives concernant le temps. Après avoir montré brièvement l'intérêt d'utiliser des modèles temporisés (section 4.1), nous décrivons les aspects sémantiques de ces modèles dans la section 4.2. Les sections 4.3 et 4.4 sont consacrées respectivement à la présentation de quelques modèles temporisés largement étudiés au cours des quinze dernières années et de logiques permettant la spécification de propriétés temporisées.

4.1. Introduction

Nous expliquons maintenant de manière informelle l'intérêt de faire apparaître le temps dans la phase de modélisation d'un système. Nous évoquons également les difficultés supplémentaires qui proviennent de cet ajout.

4.1.1. *Pourquoi introduire explicitement le temps dans les modèles ?*

Les modèles temporisés forment une sous-classe des systèmes réactifs, dans laquelle la notion de temps est manipulée de manière explicite, de sorte qu'il soit possible d'exprimer des conditions critiques sous forme de contraintes quantitatives.

Un exemple classique est celui des temps de réponse : on souhaite vérifier que, « chaque fois qu'une situation dangereuse se produit, une alarme se déclenche ».

Chapitre rédigé par Béatrice BÉRARD.

Lorsque le modèle n'est pas temporisé, il faut se contenter de cette spécification telle quelle, sans aucune précision sur le délai séparant l'apparition de la situation dangereuse du déclenchement de l'alarme. Au contraire, dans un modèle où le temps est représenté explicitement, on peut ajouter qu'il doit se passer moins de trente secondes entre les deux évènements.

Bien entendu, un système peut faire intervenir des fonctions du temps qui sont plus compliquées que de simples délais, comme la température d'une pièce chauffée par un appareil à thermostat ou le niveau d'eau d'une baignoire qui se remplit d'un côté et se vide de l'autre. Dans les deux cas, température et niveau d'eau sont vues comme des *variables dynamiques*, dont on peut vouloir représenter explicitement l'évolution en fonction du temps et spécifier qu'elles restent comprises entre des valeurs seuils.

Des exemples similaires sont développés de manière plus formelle dans la suite du chapitre.

4.1.2. *Les difficultés liées à l'ajout du temps*

Ces difficultés sont de natures diverses, nous en abordons seulement quelques unes.

La première concerne l'expression des propriétés. En reprenant l'exemple du temps de réponse, supposons que $p(t)$ et $a(t)$ désignent respectivement l'occurrence d'une situation dangereuse et d'une alarme au temps t. En logique du premier ordre, on pourrait écrire :

$$\forall t\ (p(t) \Rightarrow \exists t'\ (t \leq t' \leq t + 30\ \wedge\ a(t')))$$

ce qui permet d'imaginer la difficulté que l'on aurait à exprimer des propriétés plus complexes. Des logiques mieux adaptées au cadre temporisé ont donc été proposées et sont exposées plus en détail dans la section 4.4.

Le second point est plus général puisqu'il concerne les problèmes de modularité et de compositionnalité. Lors de la conception d'un système de grande taille, il est toujours souhaitable de pouvoir construire des parties du système qu'il faudra ensuite assembler. Les difficultés habituelles de synchronisation entre les différents modules se compliquent par la nécessité d'y ajouter une synchronisation du temps. Il est donc d'usage, dans un cadre simplifié, d'introduire une horloge globale. Malheureusement, une telle hypothèse n'est pas toujours réaliste lorsque les composants sont géographiquement éloignés et l'introduction de plusieurs horloges augmente la complexité des problèmes de vérification. La présence d'horloges non synchrones rend même indécidables la plupart des questions de vérification.

Enfin, dans un souci analogue de conception de grands modèles, se pose la question de la construction hiérarchique et du raffinement. Ce problème a été jusqu'à présent très peu étudié dans le cadre temporisé [BER 06].

4.2. Aspects sémantiques des modèles temporisés

Les systèmes de transitions étiquetés forment un cadre bien adapté à la description des comportements de systèmes dynamiques. Ils peuvent être étendus pour les modèles temporisés, en ajoutant des transitions de temps aux classiques transitions d'actions. Après une description des domaines de temps, nous définissons ces systèmes et les langages associés, et nous relevons les difficultés provenant de l'ajout des transitions temporisées.

4.2.1. *La représentation du temps.*

Les domaines traditionnellement utilisés pour le temps sont les ensembles $\mathbb{N}$ (des entiers naturels), $\mathbb{Q}^+$ (des rationnels positifs ou nuls) et $\mathbb{R}^+$ (des réels positifs ou nuls). Le premier est un cas particulier de domaine *discret*, parfois modélisé aussi par des séquences d'évènements appelés *ticks*, lors desquels il est possible d'observer les états du système. Les deux autres représentent un temps *dense*. De façon plus générale, [NIC 93] propose un modèle unifié du domaine de temps sous la forme d'un monoïde commutatif $(\mathbb{T}, +)$, où l'élément neutre est noté 0, vérifiant les deux propriétés suivantes :

- $\forall t, t' \in \mathbb{T},\ t + t' = t \Leftrightarrow t' = 0$
- la relation $\leq$ définie par $t \leq t'$ si $\exists t'' \in \mathbb{T}$ tel que $t' = t + t''$, est un ordre total sur $\mathbb{T}$.

Cette définition implique que 0 est le plus petit élément de $\mathbb{T}$ et que si $t \leq t'$, alors il existe un unique $t'' \in \mathbb{T}$, noté $t' - t$, tel que $t' = t + t''$.
Ainsi, les éléments de $\mathbb{T}$ représentent des *dates* et la différence entre deux dates correspond naturellement à une *durée*, également dans $\mathbb{T}$. Il est possible dans ce cadre de formaliser les notions de domaine dense ou discret.
Une *suite de dates* est une suite croissante (au sens large) d'éléments de $\mathbb{T}$. Lorsque le domaine est $\mathbb{Q}^+$ ou $\mathbb{R}^+$, une telle suite peut être convergente tout en étant infinie et non stationnaire. Les comportements associés sont alors qualifiés de convergents ou *zénoniens*, en référence au fameux paradoxe de Zénon d'Élée. De tels comportements sont souvent exclus dans la sémantique des modèles, car ils posent des problèmes de réalisation. Ils ont néanmoins été étudiés, par exemple dans [HAN 95, BER 00]. Dans la suite, pour des raisons de simplicité, le domaine de temps $\mathbb{T}$ est plongé dans $\mathbb{R}$.

4.2.2. *Systèmes de transitions temporisés.*

Rappelons qu'un système de transitions étiqueté est habituellement défini par un quadruplet $\mathcal{T} = (L, S, s_0, E)$, où L est l'ensemble des étiquettes, S est l'ensemble des configurations, s_0 est la configuration initiale et E est la relation de transition, donnée

par un sous-ensemble de $S \times L \times S$. On note généralement $s \xrightarrow{\ell} s'$ une transition (s, ℓ, s') de E et $\xrightarrow{\ell}$ l'ensemble des transitions d'étiquette ℓ.

Un *système de transitions temporisé* (STT) est un système de transitions $\mathcal{T}$ étiqueté par $L = \Sigma \cup \mathbb{T}$, où Σ est un alphabet fini et $\mathbb{T}$ un domaine de temps. Les transitions $\xrightarrow{a}$, pour a dans Σ, correspondent à des actions au sens usuel et sont considérées comme instantanées. Dans certaines variantes, on utilise $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ au lieu de Σ, le mot vide ε représentant une action non observable. Dans d'autres variantes, comme les structures de Kripke avec durées (cf. définition 4.5, page 67), l'alphabet Σ est vide. Les transitions $\xrightarrow{d}$, pour $d \in \mathbb{T}$, expriment l'écoulement d'une durée et doivent de ce fait vérifier des conditions particulières, qui traduisent la compatibilité du système par rapport aux opérations sur le temps :

- *délai nul :* $s \xrightarrow{0} s'$ si et seulement si $s' = s$,
- *additivité :* si $s \xrightarrow{d} s'$ et $s' \xrightarrow{d'} s''$, alors $s \xrightarrow{d+d'} s''$,

De plus, le système $\mathcal{T}$ est dit *déterministe par rapport au temps* si $s \xrightarrow{d} s_1$ et $s \xrightarrow{d} s_2$ implique $s_1 = s_2$. Il est dit *continu* si $s \xrightarrow{d} s'$ implique que, pour tous d' et d'' tels que $d = d' + d''$, il existe s'' tel que $s \xrightarrow{d'} s''$ et $s'' \xrightarrow{d''} s'$.

Une autre sémantique possible consiste à considérer un seul type de transitions, avec pour étiquettes des couples $(a, d) \in \Sigma \times \mathbb{T}$. Cette approche permet, par exemple, d'interpréter d comme la durée de a. La relation de transition correspondante peut être définie à partir de $\mathcal{T}$ en combinant une transition de délai $\xrightarrow{d}$ et une transition d'action $\xrightarrow{a}$:

$$s \xRightarrow{a,d} s' \text{ s'il existe } s'' \in S \text{ tel que } s \xrightarrow{d} s'' \text{ et } s'' \xrightarrow{a} s'$$

L'ordre des deux transitions $\xrightarrow{d}$ et $\xrightarrow{a}$ est parfois interverti. Le système de transitions étiqueté $\mathcal{T}_m = (\Sigma \times \mathbb{T}, S, \Rightarrow)$ est appelé *système de transitions mixte* associé à $\mathcal{T}$, comme dans [LAB 98]. C'est cette sémantique qui est utilisée dans [ALU 90, ALU 94], où elle est définie directement à partir d'un modèle de machine.

Dans tous les cas, on définit les exécutions d'un système $\mathcal{T}$ ou $\mathcal{T}_m$ comme des chemins du graphe partant de la configuration initiale, avec parfois des restrictions conduisant à un sous-ensemble d'exécutions « acceptables ». Notons qu'avec les propriétés de délai nul et d'additivité énoncées ci-dessus, on peut toujours se ramener, pour $\mathcal{T}$, à des exécutions où actions et durées alternent strictement.

4.2.3. *Langages temporisés.*

Un *mot temporisé* est un couple $w = (\sigma, \theta)$, où σ est un mot $a_1 a_2 \ldots$ sur l'alphabet Σ et $\theta = t_1 t_2 \ldots$ est une suite de dates dans $\mathbb{T}$ de même longueur que σ. On note

aussi $w = (a_1, t_1)(a_2, t_2) \ldots$. Dans ce cadre, un couple (a_i, t_i) est appelée une *action temporisée*, puisqu'elle peut représenter une observation du système à la date t_i, date à laquelle se produit instantanément l'action a_i.
A partir de la suite d'étiquettes $(a_i, d_i)_{i \geq 1}$ d'un chemin dans un système de transitions mixte $\mathcal{T}_m$, on obtient un mot temporisé $w = (a_i, t_i)_{i \geq 1}$ en cumulant les délais : $t_i = \sum_{j \leq i} d_j$, pour $i \geq 1$. Réciproquement, à partir d'un mot temporisé, on obtient une séquence de couples $(a_i, d_i) \in \Sigma \times \mathbb{T}$, où $d_i = t_i - t_{i-1}$ (en posant $t_0 = 0$) peut aussi être considéré comme la durée maximale de a_i.

Lorsque les étiquettes sont dans $\Sigma \cup \mathbb{T}$, il faut se ramener à des chemins où les transitions d'action alternent avec les transitions de temps. De plus, si les étiquettes d'actions sont dans $\Sigma \cup \{\varepsilon\}$, il faut supprimer les actions (a_i, t_i) non observables, c'est-à-dire celles pour lesquelles $a_i = \varepsilon$. Notons que, dans ce dernier cas, l'opération de calcul des dates à partir des délais doit nécessairement précéder l'effacement des actions non observables, pour que les délais associés à ces dernières soient pris en compte.

Un *langage temporisé* est un ensemble de mots temporisés. On peut se ramener au cadre classique des mots non temporisés par la projection $Untime(L) = \{\sigma \mid \exists \theta \ (\sigma, \theta) \in L\}$ d'un langage temporisé L. Les langages temporisés ont été peu étudiés indépendamment des systèmes de transitions temporisés. Quelques travaux [ASA 97, HEN 98, BOU 99, BOU 02, ASA 02] ont néanmoins abordé la question des expressions régulières dans le cadre temporisé, et des approches algébriques ont été proposées dans [GRO 95, DIM 00, BOU 01].

4.2.4. *Problèmes sémantiques*

Un problème typique est celui de l'urgence [BOR 98b]. Une transition d'action d'étiquette $a \in \Sigma$ est dite *urgente* dans une configuration s du système de transitions si elle ne peut pas être retardée dans cet état. Cela signifie donc qu'aucune transition de temps n'est possible dans cette configuration, ou encore que la transition $s \xrightarrow{a}$ doit se produire dans un délai nul. Cette notion d'urgence est très utile en pratique, par exemple lorsque l'on souhaite modéliser l'enchaînement instantané de plusieurs transitions d'actions. C'est le cas dans certains modèles où cette caractéristique sert à simuler une synchronisation de plusieurs composants. Elle donne toutefois lieu à des ambiguïtés dans la définition de la sémantique de certains modèles, et peut aussi occasionner des blocages lors de la composition de modèles [BOR 98a]. Ces deux points sont illustrés dans la section suivante.

4.3. Modèles temporisés classiques

Cette section montre comment le temps a été ajouté à des modèles classiques comme les automates finis ou les réseaux de Petri, afin d'obtenir des comportements temporisés.

4.3.1. *Comment ajouter le temps ?*

Le mécanisme utilisé pour introduire un temps explicite dans des modèles déjà existants consiste en l'introduction d'une horloge globale, éventuellement complétée par la *temporisation des transitions* : à partir d'un modèle non temporisé avec un ensemble Q d'états, parfois appelés *modes de contrôle*, on munit d'une contrainte de temps chacune des transitions ou *changements de mode* $e : q \xrightarrow{a} q'$, pour $q, q' \in Q$, et on définit ensuite un système de transitions temporisé. Ce mécanisme comporte lui-même deux variantes : l'ajout de variables fonctions du temps et la temporisation par intervalles, qui en constitue un cas particulier.

4.3.2. *Horloge globale discrète*

Nous illustrons d'abord un premier type d'approche, qui a été largement utilisé depuis [EME 92], en supposant que le domaine de temps est $\mathbb{N}$. Il est fondé sur la remarque que les observations d'un système ont toujours lieu selon des séquences discrètes, dûes par exemple à un échantillonage. Avec une granularité assez fine, il suffirait alors de représenter une horloge globale par une séquence de « ticks » et de la synchroniser avec le reste du système. Cette technique correspond à un dépliage du modèle initial, dans lequel toutes les unités de temps sont apparentes.

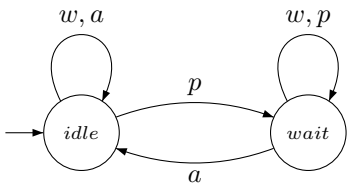


Figure 4.1. *$\mathcal{A}_1$: toute action p sera suivie de a*

EXEMPLE.– Considérons un système où trois actions sont possibles : w (travail), p (occurrence d'un problème) et a (alarme) et devant satisfaire la spécification suivante : « toute occurrence d'un problème est suivie par une alarme ». Dans ce cas très simple, le système peut être modélisé par l'automate fini $\mathcal{A}_1$ de la figure 4.1. Ajoutons maintenant une contrainte de temps, par exemple : « un problème est suivi par une alarme

dans un délai inférieur ou égal à 3 unités de temps ». En introduisant dans le système l'action supplémentaire τ, qui représente l'action vide (ou une action interne), on synchronise $\mathcal{A}_1$ avec l'horloge (réduite) $\mathcal{H}$ de la figure 4.2,

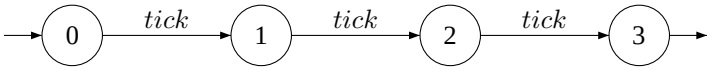


Figure 4.2. *$\mathcal{H}$: trois ticks consécutifs*

ce qui revient à déplier l'automate précédent et fournit l'automate « temporisé » $\mathcal{A}_2$ de la figure 4.3 dans lequel, implicitement, l'intervalle entre deux actions (instantanées) est d'une unité de temps. Dans ce cadre, les suites de dates sont de la forme $(t_i)_{i\geq 0}$, avec $t_i = i$ pour tout i.

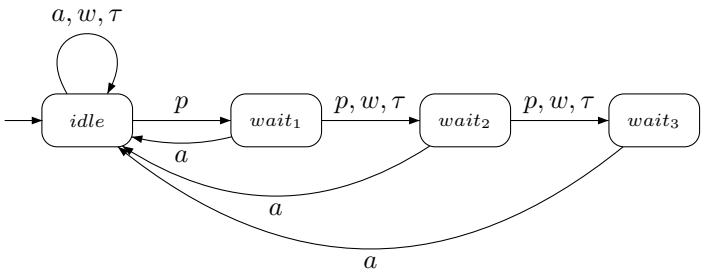


Figure 4.3. *$\mathcal{A}_2$: toute action p est suivie d'un a après au plus 3 unités de temps*

On peut aussi modifier légèrement les modèles de façon à autoriser plusieurs actions (généralement un nombre fini) simultanées. On obtient alors des suites de dates « avec répétitions ».

Les inconvénients de ce type de modèles sont la sensibilité aux constantes, qui peut conduire à une explosion combinatoire du nombre d'états lors du dépliage, et la difficulté d'y associer un traitement symbolique des contraintes.

4.3.3. *Exemples de variables fonctions du temps : les automates temporisés et les systèmes hybrides*

Une méthode plus récente utilise des variables pour décrire certains aspects du système [DIL 89, ALU 97]. Ces variables sont considérées comme des fonctions du temps dans les modes de contrôle, et elles peuvent être modifiées de manière discrète

lors des changements de modes. La classe très générale des systèmes hybrides est obtenue de cette façon, avec des variables à valeurs réelles. On en trouve une présentation synthétique pour les cas où le contrôle est fini dans [HEN 96].

Automates temporisés. Nous commençons par la définition des automates temporisés, qui forment le modèle le plus simple de cette classe, et pour lesquels de nombreux problèmes de vérification sont décidables. Dans ce cadre simplifié, le contrôle consiste en un automate fini au sens usuel et les seules variables utilisées sont des horloges : dans un mode de contrôle donné, toutes ces variables évoluent donc à la même vitesse, de manière synchrone avec le temps. De plus, lors d'une transition discrète, la valeur courante d'une ou plusieurs horloges peut être comparée à une constante et certaines horloges peuvent être remises à zéro.

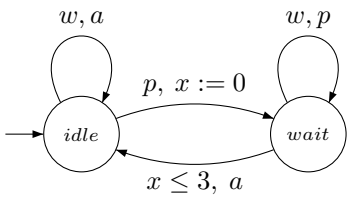


Figure 4.4. *$\mathcal{A}_3$: toute action p est suivie d'un a après au plus 3 unités de temps*

EXEMPLE.– Reprenons dans ce contexte la spécification « toute occurrence de l'action p est suivie d'une occurrence de l'action a dans un délai inférieur ou égal à 3 unités de temps ». Ce délai peut être pris en compte en introduisant une horloge x, remise à zéro lorsque p se produit, et comparée à la constante 3 lorsque a apparaît. Le résultat est l'automate temporisé $\mathcal{A}_3$ de la figure 4.4, qui est directement obtenu en décorant les transitions de $\mathcal{A}_1$ (figure 4.1) par les opérations sur les horloges.

Nous donnons maintenant une définition formelle des automates temporisés. Pour un ensemble X d'horloges, on note $\mathcal{P}(X)$ l'ensemble des parties de X et $\mathcal{C}(X)$ l'ensemble des conjonctions de contraintes atomiques de la forme $x \sim c$ où x est une horloge, c une constante (généralement entière) et $\sim$ un opérateur de comparaison dans $\{<, \leq, =, \geq, >\}$. Un élément de $\mathcal{C}(X)$ est appelé une *garde*.

DÉFINITION 4.1 *Un* automate temporisé *(sur l'alphabet Σ) est un quintuplet $\mathcal{A} = (\Sigma, X, Q, q_0, \Delta)$, où X est un ensemble d'horloges, Q est un ensemble fini d'états (ou modes de contrôle), $q_0 \in Q$ est l'état initial et Δ est un sous-ensemble de $Q \times \mathcal{C}(X) \times \Sigma \times \mathcal{P}(X) \times Q$. Une transition de Δ, notée $q \xrightarrow{g,a,r} q'$, exprime un passage possible de q à q' avec l'action a, si la garde g est vraie. Les horloges de $r \subseteq X$ sont alors remises à zéro, ce qui est souvent noté aussi $r := 0$, comme dans la figure 4.4.*

Sémantique des automates temporisés. La sémantique d'un automate temporisé $\mathcal{A} = (\Sigma, X, Q, q_0, \Delta)$ est donnée par un système de transitions temporisé $\mathcal{T} = (\Sigma \cup \mathbb{T}, S, s_0, E)$, avec un domaine de temps qui peut être dense ou discret. Comme $\mathbb{T}$ est plongé dans $\mathbb{R}$, il peut être défini de la manière suivante. Les configurations sont des couples (q, v) où $q \in Q$ et v est une valuation des horloges, c'est-à-dire une application de X dans $\mathbb{R}$. Autrement dit, $S = Q \times \mathbb{R}^X$. La configuration initiale est $(q_0, \mathbf{0})$, où $\mathbf{0}$ désigne la valuation nulle pour toutes les horloges de X.

Pour décrire les transitions, nous définissons

– le passage du temps depuis une valuation v par : $(v+d)(x) = v(x)+d$ pour toute horloge x, ce qui exprime l'évolution synchrone des valeurs d'horloge lorsqu'une durée d s'écoule,

– la remise à zéro des horloges de $r \subseteq X$ à partir de v par :
$v[r \mapsto 0](x) = 0$ si $x \in r$ et 0 sinon.

Les deux types de transitions de E sont alors définies par :

• la transition de durée d, qui s'écrit $(q, v) \xrightarrow{d} (q, v + d)$,

• la transition discrète d'étiquette a, qui s'écrit $(q, v) \xrightarrow{a} (q', v')$, possible s'il existe une transition $q \xrightarrow{g,a,r} q'$ dans Δ telle que la valuation v satisfasse la garde g, et avec $v' = v[r \mapsto 0]$.

EXEMPLE.– Pour l'automate $\mathcal{A}_3$ de la figure 4.4, un début d'exécution pourrait être :
$(idle, 0) \xrightarrow{1.5} (idle, 1.5) \xrightarrow{p} (wait, 0) \xrightarrow{2.4} (wait, 2.4) \xrightarrow{a} (idle, 2.4) \cdots$

Un tel système de transitions contient une infinité d'états, non dénombrable lorsque le domaine de temps est $\mathbb{R}$. En ajoutant à un automate temporisé des conditions d'acceptation sur les exécutions, comme par exemple des états finals pour les exécutions finies et/ou des conditions de Büchi pour les exécutions infinies [ALU 94], on peut associer à cet automate un langage de mots temporisés. On définit ainsi des classes de langages temporisés *reconnaissables*.

Les systèmes de transitions $\mathcal{T}$ et $\mathcal{T}_m$ associés à $\mathcal{A}_3$ sont représentés de manière très abrégée dans la figure 4.5. Par exemple, toutes les configurations de la forme $(idle, d)$, $d \geq 0$ ont été fusionnées et toutes les transitions de temps sont étiquetées par d, considéré comme un paramètre, sans précision sur ses valeurs possibles. Ces systèmes peuvent être comparés à l'automate $\mathcal{A}_2$ de la figure 4.3 obtenu avec un temps discret. L'analyse des automates temporisés repose sur une abstraction similaire à ces représentations, qui consiste à regrouper des états de la forme (q, v) pour un état q fixé, en regroupant les valuations d'horloge qui ont un comportement « similaire ». Cette technique a permis de montrer la décidabilité du vide pour la classe des langages temporisés acceptés par des automates temporisés [ALU 90, ALU 94].

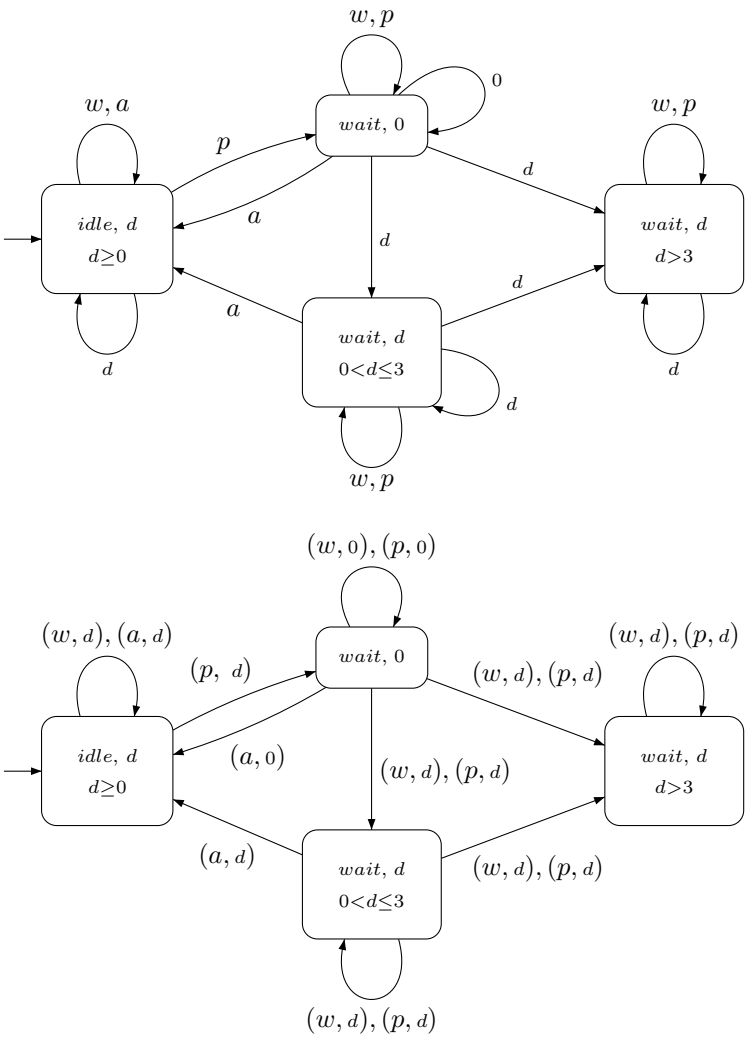


Figure 4.5. *Une représentation simplifiée des systèmes $\mathcal{T}$ et $\mathcal{T}_m$*

Urgence et vivacité. Remarquons que dans ces systèmes, rien n'empêche *a priori* de rester indéfiniment dans une suite d'états $(wait,\ d_i)$, pour une suite quelconque $(d_i)_{i\geq 1}$. Ce problème a conduit à incorporer dans les modèles des conditions pour assurer la vivacité ou l'absence de blocage. Différentes solutions ont été proposées, parmi lesquelles on peut citer l'introduction de conditions de Büchi, présentes dans le modèle original de [ALU 90], mais aussi l'ajout d'*invariants*, qui sont des contraintes d'horloges associées aux modes de contrôle [NIC 92], ou de *deadlines*, qui sont des conditions limites imposées pour le franchissement des transitions [SIF 96].

Par exemple, pour que la propriété considérée soit satisfaite par l'automate $\mathcal{A}_3$, on peut ajouter à l'état $wait$ la contrainte $x \leq 3$, et imposer dans la sémantique de ne conserver que les configurations pour lesquelles cette condition est satisfaite, c'est-à-dire les configurations $(wait, d)$ avec $d \leq 3$. Les relations entre diverses conditions de vivacité sont examinées en détail dans [BOR 97, LAB 98] et d'autres mécanismes permettant de garantir l'urgence d'une transition ont été réalisés dans des outils d'analyse des systèmes temporisés ou hybrides comme HYTECH [HEN 95], KRONOS [DAW 96] ou UPPAAL [LAR 97a].

Systèmes hybrides. Nous commençons par l'exemple du thermostat, extrait de [JAF 91] (et simplifié) :

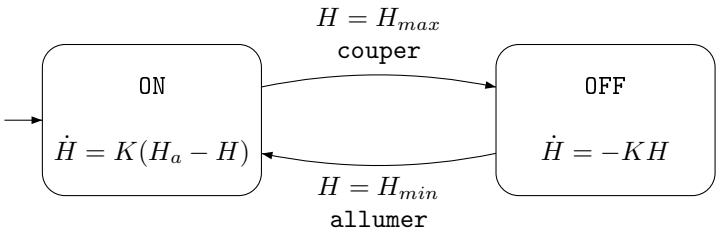


Figure 4.6. *Automate hybride modélisant un thermostat*

Cet automate hybride décrit le fonctionnement d'un thermostat, qui déclenche ou arrête un appareil de chauffage suivant les informations provenant d'un capteur de température. Il comporte deux modes de contrôle : `ON` et `OFF`, qui manipulent une variable H représentant la température. Dans chacun des deux modes, l'évolution de H en fonction du temps est caractérisée par une équation différentielle linéaire. On suppose pour simplifier que le système se trouve initialement dans le mode `ON` avec une température nulle et que les constantes K et H_a dépendent de l'appareil de chauffage. Dans ce mode, l'appareil est en marche et la température augmente en suivant la loi spécifiée, jusqu'à atteindre une valeur maximale, H_{max}. Le système passe alors en mode `OFF`, où l'appareil s'éteint et la température décroît. Lorsque la température vaut H_{min}, l'appareil se remet en marche.

Remarquons qu'un système hybride pour lequel la contrainte sur la dérivée dans les états est de la forme $\alpha \leq \dot{x} \leq \beta$ n'est pas déterministe par rapport au temps, au sens défini dans le paragraphe 4.2.2.

Enfin, comme il a été mentionné plus haut, la plupart des problèmes de vérification sur ces modèles sont indécidables, et ce dès qu'on utilise au moins deux variables de pentes différentes. Cependant, l'outil HYTECH [HEN 95] implante un semi-algorithme de calcul des configurations accessibles pour une classe assez large d'automates hybrides, appelés automates hybrides linéaires.

Nous terminons ce paragraphe par une définition formelle des automates hybrides linéaires, tels qu'ils sont décrits dans le format d'entrée de HYTECH.
Pour un ensemble X de variables à valeurs réelles, on appelle *terme linéaire* sur X une expression de la forme $k_0 + k_1x_1 + \cdots + k_mx_m$, où les x_i sont des variables de X et les k_i des constantes dans (un sous-ensemble de) $\mathbb{R}$. Si ℓ_1 et ℓ_2 sont des termes linéaires, $\ell_1 \leq \ell_2$ est une *inégalité linéaire*. Une *contrainte linéaire* est une conjonction d'inégalités linéaires. L'ensemble des contraintes linéaires sur X est noté $\mathcal{C}_\ell(X)$. Comme pour les automates temporisés, une valuation est une application $v : X \to \mathbb{R}$, qui peut aussi être vue comme un n-uplet $\mathbf{x}$ de valeurs réelles, où $n = |X|$ est le cardinal de X. Une valuation $\mathbf{x}$ satisfait une contrainte $\varphi \in \mathcal{C}_\ell(X)$, noté $\mathbf{x} \models \varphi$, si le prédicat fermé $\varphi[X := \mathbf{x}]$ est vrai.
Pour décrire un automate hybride linéaire, on utilise les notations suivantes : $X' = \{x' \mid x \in X\}$ est une copie de X qui désigne les nouvelles valeurs des variables après une transition discrète, et $\dot{X} = \{\dot{x} \mid x \in X\}$ est une autre copie de X représentant les *pentes* des variables de X, qui caractérisent l'évolution des variables dans les modes de contrôle. Avec ces notations, une remise à zéro de la variable x s'écrit $x' = 0$.

DÉFINITION 4.2 *Un* automate hybride linéaire *(sur l'alphabet Σ) est un n-uplet $\mathcal{A} = (\Sigma, X, Q, q_0, V_0, \Delta, Inv, \Pi)$, où*

– X est un ensemble de variables,

– Q est un ensemble fini d'états (ou modes de contrôle), $q_0 \in Q$ est l'état initial,

– V_0 est un ensemble de valuations initiales,

– Δ est un sous-ensemble de $Q \times \mathcal{C}_\ell(X \cup X') \times \Sigma \times Q$ contenant les transitions,

– $Inv : Q \to \mathcal{C}_\ell(X)$ associe à chaque état un invariant, c'est-à-dire une contrainte que doivent satisfaire les valeurs des variables dans cet état, et

– $\Pi : Q \to \mathcal{C}_\ell(\dot{X})$ associe à chaque état une contrainte sur les pentes des variables.

Nous ne décrivons pas formellement la sémantique, qui est similaire à celle des automates temporisés, en adaptant l'évolution des variables dans les états et les mises à jour lors de transitions discrètes. Un élément de $\mathcal{C}_\ell(X \cup X')$ est appelé un *saut* et décrit une condition de franchissement de la transition. Une transition de Δ, notée $q \xrightarrow{s,a} q'$, exprime un passage possible de q à q' avec l'action a et le saut s. Par exemple, une transition étiquetée par $x \leq 2 \wedge y = 3 \wedge x' = 0 \wedge y' > 2x + 1$ ne peut être franchie que si $x \leq 2$ et $y = 3$, la variable x étant remise à zero et la nouvelle valeur de y étant choisie, de façon non déterministe, strictement plus grande que deux fois la valeur de x plus un.
De même, si $\Pi(q) = 2 \leq \dot{x} \leq 5 \wedge \dot{y} = 0$ pour un état $q \in Q$, alors dans cet état, la pente de x est comprise entre 2 et 5 tandis que y reste constante.

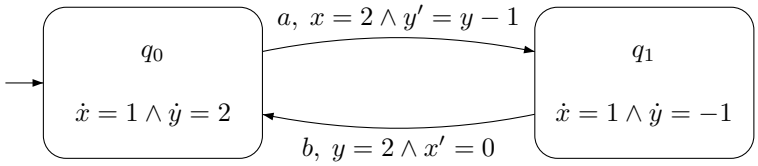


Figure 4.7. *Exemple d'automate hybride*

EXEMPLE.– Considérons l'automate de la figure 4.7. Dans ce cas très simple, la suite des transitions partant de q_0 avec les deux variables x et y à 0 est :

$$\left(q_0, \begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) \xrightarrow{2} \left(q_0, \begin{bmatrix} 2 \\ 4 \end{bmatrix}\right) \xrightarrow{a} \left(q_1, \begin{bmatrix} 2 \\ 3 \end{bmatrix}\right) \xrightarrow{1} \left(q_1, \begin{bmatrix} 3 \\ 2 \end{bmatrix}\right) \xrightarrow{b} \left(q_0, \begin{bmatrix} 0 \\ 2 \end{bmatrix}\right)$$
$$\xrightarrow{2} \left(q_0, \begin{bmatrix} 2 \\ 6 \end{bmatrix}\right) \xrightarrow{a} \left(q_1, \begin{bmatrix} 2 \\ 5 \end{bmatrix}\right) \xrightarrow{3} \left(q_1, \begin{bmatrix} 5 \\ 2 \end{bmatrix}\right) \xrightarrow{b} \left(q_0, \begin{bmatrix} 0 \\ 2 \end{bmatrix}\right) \cdots$$

et, à partir de là, le système reste dans une boucle, ce qui donne, géométriquement, la trajectoire indiquée en figure 4.8.

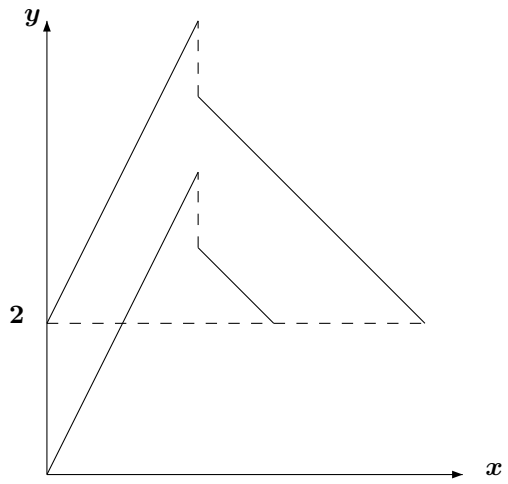


Figure 4.8. *Trajectoire de l'automate de la figure 4.7*

Composition. Comme dans le cas non temporisé, la composition de modèles temporisés est obtenue à partir du produit cartésien des ensembles d'états, avec une fonction de synchronisation sur les étiquettes des transitions. Lorsque plusieurs transitions peuvent se synchroniser, les composants correspondants progressent simultanément. Lorsqu'une action est interne à un composant, celui-ci peut progresser de manière asynchrone. Pour des systèmes temporisés ou hybrides, il faut ajouter les transitions de durées, correspondant à la progression du temps, qui doit être la même pour tous les composants. Le principal problème qui se pose dans ce contexte est le suivant [SIF 96]. Supposons qu'un modèle non temporisé $\mathcal{M}$ soit obtenu par composition parallèle de deux modèles $\mathcal{M}_1$ et $\mathcal{M}_2$. Si ces modèles sont ensuite temporisés, c'est-à-dire étendus en $\mathcal{M}^t$, $\mathcal{M}_1^t$ et $\mathcal{M}_2^t$ par ajout de contraintes temporelles, la composition de systèmes temporisés doit garantir que le produit de $\mathcal{M}_1^t$ et $\mathcal{M}_2^t$ correspond bien à $\mathcal{M}^t$.

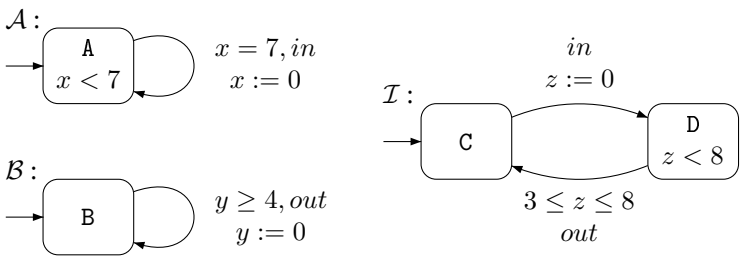


Figure 4.9. *Trois processus communiquant par in et out*

L'exemple proposé dans [SIF 96] illustre cette difficulté par la spécification de trois processus, décrits par des automates temporisés avec invariants (figure 4.9) : le premier ($\mathcal{A}$) produit des messages in toutes les 7 unités de temps, le second ($\mathcal{B}$) émet des sorties out séparées d'au moins 4 unités de temps et le troisième ($\mathcal{I}$) assure la communication en transmettant à $\mathcal{B}$ les messages qu'il reçoit de $\mathcal{A}$ avec un délai compris entre 3 et 8 unités de temps. La figure 4.10 montre comment est réalisée la composition $\mathcal{A} \mid \mathcal{B} \mid \mathcal{I}$, en utilisant une synchronisation sur les actions in et out. Si maintenant l'intervalle de transmission $[3, 8]$ est remplacé par $]7, 8]$, on obtient un blocage *temporel* dans l'état ABD, avec les valeurs d'horloge $x = 7$ et $z = 7$, puisque l'action out ne peut pas avoir lieu et que la progression du temps est limitée par l'invariant $x < 7$.

Les problèmes viennent donc essentiellement de la combinaison entre la synchronisation des actions et la synchronisation temporelle. Plusieurs propositions ont été faites pour résoudre cette question, par exemple en utilisant des priorités sur les actions [BOR 98a, BOR 00].

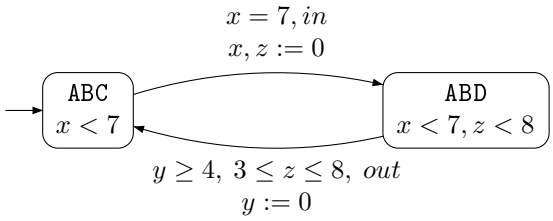


Figure 4.10. *Le produit synchronisé $\mathcal{A} \mid \mathcal{B} \mid \mathcal{I}$*

4.3.4. *Exemples de temporisation par intervalles*

Cette méthode consiste à associer à chaque transition $e : q \xrightarrow{a} q'$ un intervalle (min_e, max_e) du domaine de temps $\mathbb{T}$, avec $min_e \leq max_e$, dont les bornes représentent respectivement le délai minimal et le délai maximal autorisés pour exécuter l'action a dans l'état q. Par analogie avec les automates temporisés, on voit qu'il est possible d'associer une horloge implicite au système, remise à zéro à chaque transition. Sa valeur indique alors le temps écoulé depuis l'arrivée dans un état q et elle peut être contrainte à rester dans l'intervalle associé. C'est ce type d'approche qui a été utilisé pour la temporisation des réseaux de Petri dans [MER 74, JON 77], pour les systèmes temporisés de [MAL 91, HEN 92, HEN 94a], et pour les structures de Kripke avec durées [CAM 95, LAR 02].

Réseaux de Petri temporels. De très nombreuses extensions temporisées des réseaux de Petri ont été proposées : elles peuvent associer des paramètres de temps aux transitions, aux jetons, aux places, ou aux arcs. Nous nous restreignons dans ce paragraphe à la variante introduite dans [MER 74], qui illustre la méthode de temporisation des transitions par intervalle. Dans ce contexte, nous notons $\mathcal{I}(\mathbb{Q})$ l'ensemble des intervalles dont la borne inférieure est dans $\mathbb{Q}_+$ et la borne supérieure dans $\mathbb{Q}_+ \cup \{\infty\}$.

DÉFINITION 4.3 *Un* réseau de Petri temporel *(RPT) $\mathcal{N}$ est un n-uplet $(P, T, \Sigma_\varepsilon, {}^\bullet(.), (.)^\bullet, M_0, \lambda, I)$ où :*

– *P est un ensemble fini de* places *;*

– *T est un ensemble fini de* transitions *avec $P \cap T = \emptyset$;*

– *${}^\bullet(.) \in (\mathbb{N}^P)^T$ représente l'ensemble des* pré-conditions *et $(.)^\bullet \in (\mathbb{N}^P)^T$ l'ensemble des* post-conditions *;*

– *$M_0 \in \mathbb{N}^P$ est le* marquage initial *;*

– *$\lambda : T \to \Sigma_\varepsilon$ est la fonction d'*étiquetage *des transitions ;*

– *$I : T \to \mathcal{I}(\mathbb{Q})$ associe à chaque transition un* intervalle de franchissement.

EXEMPLE.– La figure 4.11 illustre la représentation graphique d'un réseau de Petri temporel. Chaque transition est représentée avec son étiquette et son intervalle

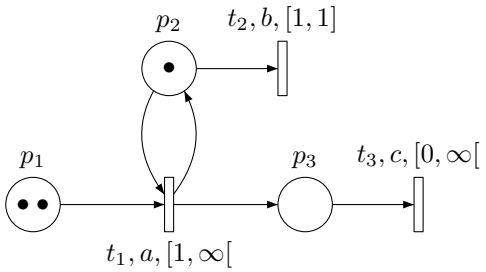


Figure 4.11. *Un exemple de réseau de Petri temporel*

de franchissement. Par exemple, la transition t_1 a pour étiquette a et pour intervalle $I(t_1) = [1, \infty[$. Le marquage initial correspond à deux jetons dans la place p_1 et un jeton dans la place p_2.

Sémantique des réseaux de Petri temporels. Un marquage M d'un RPT est une application de P dans $\mathbb{N}^P$ et $M(p)$ est le nombre de jetons dans la place p. Une transition t est franchissable dans un marquage M si $M \geq {}^\bullet t$. Nous notons $En(M)$ l'ensemble de ces transitions. Comme dans le cas usuel, le franchissement de t produit un nouveau marquage $M' = M - {}^\bullet t + t^\bullet$. Pour savoir si une transition peut être effectivement franchie, il faut que le temps écoulé depuis la dernière arrivée dans M soit dans l'intervalle $I(t)$. De plus, pour toutes les transitions franchissables, le temps ne s'écoule plus dès qu'une des bornes supérieures est atteinte. On note $\nu \in (\mathbb{R}_+)^{En(M)}$ une valuation, qui représente le temps écoulé depuis la dernière arrivée dans M. Une configuration du RPT $\mathcal{N}$ est un couple (M, ν). Elle est admissible si $\forall t \in En(M), \nu(t) \in I(t)^\downarrow$, où $I^\downarrow = \{x \mid \exists y \in I,\ x \leq y\}$ est la fermeture vers le bas de l'intervalle I. Nous notons *ADM*($\mathcal{N}$) l'ensemble des configurations admissibles.

Un point important qu'il faut définir concerne la mise à jour des informations temporelles après le franchissement d'une transition, ce qui revient à préciser quand l'horloge implicite de la transition doit être remise à zéro : nous dirons que t' est nouvellement franchissable après le franchissement de t depuis le marquage M si le prédicat $\uparrow enabled(t', M, t)$ défini par

$$\uparrow enabled(t', M, t) = (t' \in En(M - {}^\bullet t + t^\bullet)) \wedge (t' \notin En(M))$$

est vrai.

Autrement dit, t' est nouvellement franchissable si elle ne l'était pas avant le franchissement de t mais le devient après ce franchissement. Ce choix correspond à une sémantique dite *atomique persistante*, qui n'est pas la plus fréquemment utilisée mais qui est au moins aussi puissante que les variantes usuelles et plus simple d'utilisation [BER 05b].

DÉFINITION 4.4 *La* sémantique d'un RTP $\mathcal{N}$ *est définie formellement par le système de transitions temporisé* $\mathcal{T} = (L, S, s_0, E)$ *où* $L = \Sigma_\varepsilon \cup \mathbb{R}_+$, $S = ADM(\mathcal{N})$, $s_0 = (M_0, \mathbf{0})$, *et* $E \subseteq S \times (\Sigma_\varepsilon \cup \mathbb{R}_+) \times S$ *contient les deux types de transitions suivants à partir d'une configuration admissible* (M, ν) *:*

– Pour toute transition t *franchissable dans* M *telle que* $\nu(t) \in I(t)$, *une transition discrète* $(M, \nu) \xrightarrow{\lambda(t)} (M - {}^\bullet t + t^\bullet, \nu')$ *peut être exécutée, avec* $\forall t \in En(M - {}^\bullet t + t^\bullet)$, $\nu'(t) = \begin{cases} 0 & \text{if } \uparrow enabled(t', M, t), \\ \nu(t) & \text{otherwise.} \end{cases}$

– Pour tout $d \in \mathbb{R}_+$, *tel que* $\forall t \in En(M), \nu(t) + d \in I(t)^\downarrow$, *une transition de délai* $(M, \nu) \xrightarrow{d} (M, \nu + d)$ *peut être exécutée.*

EXEMPLE.– En représentant ν par un vecteur de taille 3, où la valuation d'une transition est notée $\perp$ lorsqu'elle n'est pas définie, une exécution du réseau de Petri de la figure 4.11 est :

$$(M_0, [0, 0, \perp]) \xrightarrow{1} (M_0, [1, 1, \perp]) \xrightarrow{a} (M_1, [1, 1, 0]) \xrightarrow{a} (M_2, [\perp, 1, 0])$$
$$\xrightarrow{b} (M_3, [\perp, \perp, 0]) \xrightarrow{1.5} (M_3, [\perp, \perp, 1.5]) \cdots$$

pour les marquages $M_0 = (2, 1, 0)$, $M_1 = (1, 1, 1)$, $M_2 = (0, 1, 2)$ et $M_3 = (0, 0, 2)$.

Structures de Kripke avec durées. Remarquons tout d'abord que, pour tous les modèles définis jusqu'ici avec domaine de temps $\mathbb{R}$, la sémantique est *continue* au sens du paragraphe 4.2.2. En effet, entre deux configurations s et s' séparées par une durée d, on a une infinité de configurations intermédiaires s_δ, avec $s \xrightarrow{\delta} s_\delta$, pour $0 \leq \delta \leq d$. Nous présentons maintenant un modèle à temps discret, celui des structures de Kripke avec durées (SKD), utilisé par exemple dans [CAM 95, LAR 02], pour lequel la sémantique utilisée est une sémantique dite *de saut*, où ces configurations intermédiaires n'existent pas.

En notant $\mathcal{I}(\mathbb{N})$ l'ensemble des intervalles dont la borne inférieure est dans $\mathbb{N}$ et la borne supérieure dans $\mathbb{N} \cup \{\infty\}$, la définition formelle est la suivante :

DÉFINITION 4.5 *Une* structure de Kripke avec durées *est un triplet* $\mathcal{K} = (Q, q_0, \Delta)$ *où* Q *est un ensemble d'états,* q_0 *est l'état initial et* Δ *est un sous-ensemble de* $Q \times \mathcal{I}(\mathbb{N}) \times Q$.

EXEMPLE.– La figure 4.12, extraite de [LAR 02], décrit l'activité d'un chercheur. On voit sur ce graphe que, comme pour les réseaux de Petri temporels, les transitions sont équipées d'un intervalle, éventuellement réduit à un point. Conformément à l'intuition, on passe d'un état au suivant en choisissant une durée dans l'intervalle associé à la transition.

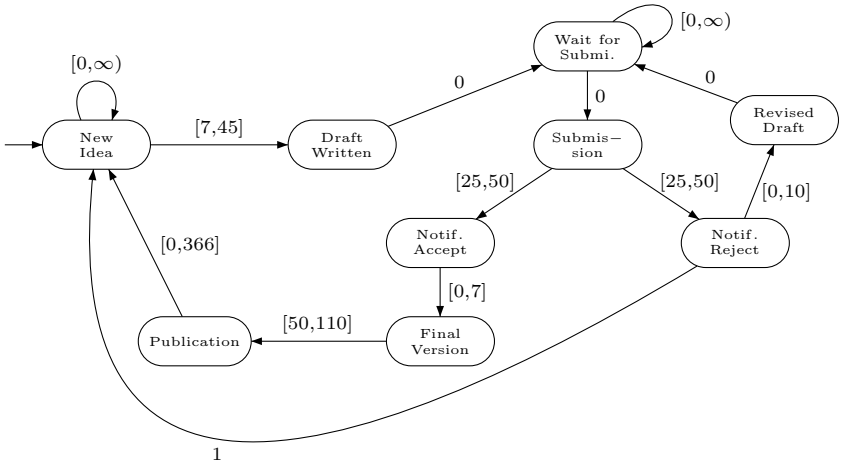


Figure 4.12. *Structure de Kripke avec durées représentant l'activité d'un chercheur*

Pour la vérification de propriétés temporelles, on ajoute généralement un étiquetage des états par des propositions atomiques. Dans l'exemple ci-dessus, on peut considérer que ce rôle est joué par les noms des états. Remarquons aussi qu'il n'y a aucune étiquette d'action dans une telle structure.

Sémantique de saut. À partir d'un tel modèle, il est possible de définir plusieurs sémantiques à base de systèmes de transitions temporisés, selon l'interprétation donnée au franchissement d'une transition de q à q' avec un délai (entier) d. Par exemple, de façon similaire à ce qui a été vu pour les automates temporisés, on pourrait supposer qu'on attend d unités de temps dans q et qu'on passe ensuite de façon instantanée dans q'.

La définition adoptée pour la sémantique de saut consiste plutôt à considérer que, si l'on se trouve à l'instant n dans l'état q, on se retrouve à l'instant $n + d$ en q' et les instants $n + 1, \ldots, n + d - 1$ ne sont pas représentés : ils n'existent pas. Cette sémantique revient à voir les durées plutôt comme des coûts. Le système de transitions temporisé correspondant est $\mathcal{T} = (\mathbb{N}, Q, q_0, E)$, avec un seul type de transitions dans E, défini par : $q \xrightarrow{d} q'$ s'il existe une transition (q, I, q') de Δ telle que d appartient à l'intervalle I. On peut donc voir cette sémantique comme un système de transitions mixte, où une seule action (silencieuse) est associée aux différentes durées.

EXEMPLE.– Pour le cas de la figure 4.12, un début d'exécution pourrait être :
$NewIdea \xrightarrow{15} NewIdea \xrightarrow{10} DraftWritten \xrightarrow{0} WaitforSubmi. \cdots$

Notons pour finir que cette sémantique est peu robuste vis-à-vis des questions de composition, mais que les systèmes correspondants sont plus faciles à analyser que les automates temporisés (voir par exemple les résultats de [LAR 02]).

4.3.5. *Les algèbres de processus temporisées*

Les mécanismes de composition parallèle proposés dans le cadre des algèbres de processus (voir section 2.3.4, page 20) peuvent constituer des réponses aux problèmes mentionnés plus haut.

EXEMPLE.– Commençons par l'exemple tiré de [OUA 03] (dérivé de l'exemple classique) qui définit un distributeur de chocolat et de biscuits par l'équation suivante :

$$M \stackrel{def}{=} in.[(choc.M + bisc.M) \stackrel{60}{\triangleright} (out!.M)]$$

La machine peut recevoir une pièce (in), puis le client choisit entre un chocolat et un biscuit ; cependant, s'il ne choisit pas en moins de 60 secondes, son argent lui est rendu immédiatement (par l'action $out!$) et la machine retourne dans son état initial. On peut associer à ce processus l'automate temporisé de la figure 4.13.

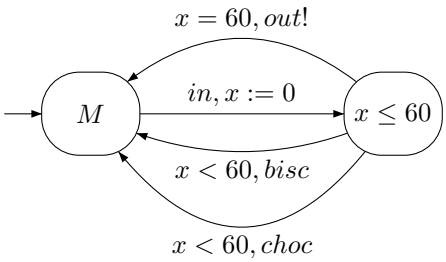


Figure 4.13. *Automate modélisant le distributeur de chocolat et biscuits*

On observe dans cette définition le mécanisme de *timeout*, où un processus $P \stackrel{d}{\triangleright} Q$ se comporte comme P dans un délai inférieur à d unités de temps, puis au bout de ce délai, se comporte comme Q. Cet opérateur est fréquemment utilisé dans les algèbres de processus temporisées. On observe également l'ajout d'actions dites *immédiates*, notées comme ici $out!$, qui ne peuvent pas être retardées. Ceci permet de réaliser l'urgence dont nous avons déjà parlé.

Nous donnons une définition élémentaire, pour fixer les idées. On considère une action interne τ et un ensemble d'actions $\Sigma \cup \{\tau\}$, muni d'une fonction de synchronisation. Pour simplifier, nous présentons seulement la synchronisation entre actions dites

complémentaires de Σ, notées a et $\overline{a}$. L'ensemble $\mathcal{E}$ des *expressions* (ou termes) d'une algèbre de processus temporisée sur cet ensemble d'actions contient des constantes, des variables, et il est défini par la grammaire :

$$E ::= \mathbf{0} \mid a \cdot E \mid (d) \cdot E \mid E_1 + E_2 \mid E_1 \parallel E_2$$

où a est une action dans $\Sigma \cup \{\tau\}$, le délai d est un élément du domaine de temps (discret ou dense), et $\mathbf{0}$ représente le processus constant qui ne fait rien, mais peut laisser passer du temps. Un *processus* est une expression sans variable libre et une constante (autre que $\mathbf{0}$) est un processus défini par une équation utilisant l'opérateur *def* (comme dans l'exemple initial).

On peut bien sûr remplacer *def* par la récursion explicite. Il faut alors procéder en deux temps, pour retrouver des systèmes de transitions « réguliers ». On peut aussi ajouter un processus constant qui ne peut ni faire d'action, ni laisser passer de temps, les opérateurs classiques de renommage et de restriction, ainsi que divers opérateurs temporisés.

Le système de transitions associé à une définition comme ci-dessus est $\mathcal{T} = (\Sigma \cup \{\tau\} \cup \mathbb{T}, \mathcal{E}, \Delta)$, où les états sont les expressions de $\mathcal{E}$, et les transitions de Δ sont données par les règles suivantes, pour $\ell \in \Sigma \cup \{\tau\} \cup \mathbb{T}$, $a \in \Sigma \cup \{\tau\}$ et $d \in \mathbb{T}$.

• Règles de définition et de préfixage par une action

$$\text{D0} : \frac{}{\mathbf{0} \xrightarrow{d} \mathbf{0}} \qquad \text{Def} : \frac{E \xrightarrow{\ell} F}{P \xrightarrow{\ell} F} \text{ si } P \stackrel{def}{=} E$$

$$\text{Act1} : \frac{}{a \cdot P \xrightarrow{a} P} \qquad \text{Act2} : \frac{P \xrightarrow{a} P'}{(\mathbf{0}) \cdot P \xrightarrow{a} P}$$

• Règles de choix : le choix est généralement une opération associative et commutative

$$\text{Choix1} : \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \text{Choix2} : \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

$$\text{Choix3} : \frac{P \xrightarrow{d} P' \qquad Q \xrightarrow{d} Q'}{P + Q \xrightarrow{d} P' + Q'}$$

• Règles de composition parallèle

$$\text{Comp1} : \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \qquad \text{Comp2} : \frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}$$

avec la synchronisation (pour $a \neq \tau$) et le délai synchrone :

$$\text{Comp3} : \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \qquad \text{Comp4} : \frac{P \xrightarrow{d} P' \qquad Q \xrightarrow{d} Q'}{P \parallel Q \xrightarrow{d} P' \parallel Q'}$$

• Délais

$$\text{D1} : \frac{}{a \cdot P \xrightarrow{d} a \cdot P} \qquad \text{D2} : \frac{P \xrightarrow{a} P'}{(0) \cdot P \xrightarrow{a} P}$$

$$\text{Ddec} : \frac{}{(d+d') \cdot P \xrightarrow{d} (d') \cdot P} \qquad \text{Dac} : \frac{P \xrightarrow{d} P'}{(d') \cdot P \xrightarrow{d+d'} P'}$$

On peut définir des règles analogues pour distinguer les actions immédiates des actions qui peuvent être retardées, ainsi que pour l'opérateur de *timeout*. Suivant ce principe, plusieurs algèbres de processus classiques ont été étendues au cadre temporisé. On trouvera un exposé général dans [NIC 91a, BAE 01], des comparaisons et diverses propositions dans [Wan 91, COR 99, COR 03, OUA 03] et un lien entre systèmes hybrides et algèbres de processus dans [NIC 91b, NIC 93, BOR 98a].

4.4. Spécification de propriétés temporisées

Cette section aborde de manière spécifique la problématique de l'expression de propriétés faisant intervenir explicitement le temps. Elle présente d'abord les logiques temporelles temporisées, en s'appuyant sur l'exemple de TCTL, puis la représentation de propriétés par un modèle, qui conduit aux techniques de vérification par des équivalences comme la bisimulation.

4.4.1. *Les logiques temporelles temporisées*

La logique temporelle CTL. Rappelons d'abord brièvement la syntaxe de la logique temporelle CTL, introduite par Pnueli [PNU 77]. Les formules sont définies par la grammaire suivante :

$$\varphi, \psi ::= P_1 \mid P_2 \mid \ldots \mid \neg\varphi \mid \mathsf{EX}\varphi \mid \mathsf{AX}\varphi \mid \varphi \wedge \psi \mid \mathsf{E}\varphi\mathsf{U}\psi \mid \mathsf{A}\varphi\mathsf{U}\psi$$

où les P_i sont des propositions atomiques dans un ensemble AP.

Ces formules s'interprètent sur des structures de Kripke usuelles, c'est-à-dire sans les durées utilisées dans la définition 4.5, mais avec un étiquetage des états par des propositions atomiques de AP. Soit donc $\mathcal{K} = (Q, q_0, \Delta, \ell)$ une telle structure de Kripke, où Δ est simplement un sous-ensemble de $Q \times Q$ et où ℓ est la fonction

d'étiquetage de Q dans $\mathcal{P}(\mathsf{AP})$. Une formule s'applique à un état $q \in Q$, en considérant l'ensemble $Exec(q)$ des exécutions de $\mathcal{K}$ issues de q. Pour une telle exécution $\sigma : q \to q_1 \to q_2 \cdots$ et pour $i \geq 0$, on note $\sigma(i)$ le i-ème état de σ (c'est-à-dire ici q_i) et on définit la sémantique de manière inductive par :

$q \models P$	si P est dans l'ensemble $\ell(q)$ des étiquettes de q
$q \models \neg\varphi$	si q ne satisfait pas φ
$q \models \varphi \wedge \psi$	si q satisfait à la fois φ et ψ
$q \models \mathsf{EX}\varphi$	s'il existe une exécution σ de $Exec(q)$ telle que $\sigma(1) \models \varphi$
$q \models \mathsf{AX}\varphi$	si toute exécution σ de $Exec(q)$ est telle que $\sigma(1) \models \varphi$
$q \models \mathsf{E}\varphi\mathsf{U}\psi$	s'il existe une exécution σ de $Exec(q)$ et un entier $j \geq 0$ tels que $\sigma(j) \models \psi$ et pour tout $k,\ 0 \leq k < j,\ \sigma(k) \models \varphi$
$q \models \mathsf{A}\varphi\mathsf{U}\psi$	si, pour toute exécution σ de $Exec(q)$ il existe un entier $j \geq 0$ tel que $\sigma(j) \models \psi$ et pour tout $k,\ 0 \leq k < j,\ \sigma(k) \models \varphi$

Les modalités X et U sont des opérateurs temporels, à interpréter le long d'une exécution, tandis que E et A sont respectivement les quantificateurs existentiel et universel sur les exécutions. Ainsi, le fait que l'état q satisfasse la formule $\mathsf{A}\varphi\mathsf{U}\psi$ exprime que ψ sera vraie plus loin dans toute exécution partant de q et que φ restera vraie entre temps. La modalité X, appelée également next, s'interprète naturellement de la façon suivante : $\mathsf{X}\varphi$ est vraie dans un état si φ est vraie dans l'état suivant.

Pour cette logique, outre les opérateurs booléens usuels, de nombreuses abréviations sont couramment utilisées, par exemple :

– $\mathsf{EF}\varphi$, définie par $\mathsf{E}(\mathtt{true})\mathsf{U}\varphi$, exprime, pour un état q, que la propriété φ sera vraie « un jour » dans au moins une exécution issue de q (la formule true étant satisfaite dans tout état),

– $\mathsf{AG}\varphi$, définie par dualité comme $\neg\mathsf{EF}(\neg\varphi)$, exprime, pour un état q, que la propriété φ est toujours vraie le long de toute exécution partant de q,

– $\mathsf{AF}\varphi$, définie par $\mathsf{A}(\mathtt{true})\mathsf{U}\varphi$, exprime, pour un état q, que pour toute exécution issue de q, la propriété φ sera vraie « un jour ».

Temporisation de CTL. Ajouter des contraintes quantitatives à ces formules peut se faire de plusieurs manières. Nous illustrons ici la méthode qui a conduit à la définition de la logique TCTL [ALU 91, ALU 93, HEN 94b]. Cette logique est interprétée sur des systèmes de transitions temporisés, dont les configurations sont munies d'ensembles de propositions atomiques. Nous nous plaçons dans un contexte où le temps est dense et où les exécutions sont continues. Dans ce cas, il est clair que la modalité X n'a pas de sens. Pour la partie de formule $\varphi\mathsf{U}\psi$, le principe consiste à assortir la modalité U d'une contrainte de la forme $\sim c$ pour un opérateur de comparaison $\sim$ et une constante c (généralement entière), de façon à contraindre la date d'occurence de la propriété ψ.

La syntaxe de TCTL est donc donnée par la grammaire suivante :

$$\varphi, \psi ::= P_1 \mid P_2 \mid \ldots \mid \neg\varphi \mid \varphi \wedge \psi \mid \mathsf{E}\varphi\mathsf{U}_{\sim c}\psi \mid \mathsf{A}\varphi\mathsf{U}_{\sim c}\psi$$

où les P_i sont des propositions atomiques de AP, l'opérateur $\sim$ appartient à l'ensemble $\{<, >, \leq, \geq, =\}$ et c est une constante dans $\mathbb{N}$.

Pour définir la sémantique de TCTL, nous considérons un système de transitions temporisé $\mathcal{T} = (\Sigma \cup \mathbb{T}, S, s_0, E, \ell)$, où ℓ est la fonction d'étiquetage des configurations de S par des propositions atomiques, et nous notons encore $Exec(s)$ l'ensemble des exécutions issues de la configuration s. Pour une telle exécution $\rho : s \xrightarrow{d_1} s'_1 \xrightarrow{a_1} s_1 \xrightarrow{d_2} s'_2 \xrightarrow{a_2} s_2 \cdots$, prise sous la forme où les durées alternent avec les actions, la continuité permet de considérer toutes les positions (temporelles) intermédiaires. Ces positions sont totalement ordonnées et on note $p' <_\rho p$ la relation associée à ρ. Pour une position p de ρ, on note aussi s_p la configuration associée à p et $\rho^{\leq p}$ le préfixe de ρ antérieur à p. Ce préfixe a une durée, notée $\mathsf{Dur}(\rho^{\leq p})$, qui est la somme des délais apparus le long de $\rho^{\leq p}$.

La sémantique de TCTL se définit alors, sur une configuration s de $\mathcal{T}$, de façon similaire à ce qui a été fait pour CTL. Nous l'écrivons seulement pour les modalités $\mathsf{U}_{\sim c}$:

$s \models \mathsf{E}\varphi\mathsf{U}_{\sim c}\psi$	s'il existe une exécution $\rho \in Exec(s)$ telle que $\rho \models \varphi\mathsf{U}_{\sim c}\psi$
$s \models \mathsf{A}\varphi\mathsf{U}_{\sim c}\psi$	si, pour toute exécution $\rho \in Exec(s),\ \rho \models \varphi\mathsf{U}_{\sim c}\psi$
avec	
$\rho \models \varphi\mathsf{U}_{\sim c}\psi$	s'il existe une position p de ρ telle que $\mathsf{Dur}(\rho^{\leq p}) \sim c$, $s_p \models \psi$ et pour toute position $p' <_\rho p,\ s_{p'} \models \varphi$

Remarquons que la modalité usuelle U s'obtient encore dans cette logique en utilisant $\mathsf{U}_{\geq 0}$.

EXEMPLE.– Revenons maintenant à la propriété considérée au début de ce chapitre : « toute occurrence d'un problème est suivie par une alarme dans un délai inférieur ou égal à 3 unités de temps ». En utilisant des abréviations similaires à celles de CTL, et les propositions atomiques probleme et alarme, cette propriété s'exprime par la formule de TCTL :

$$\Phi :\ \mathsf{AG}(\mathsf{probleme} \Rightarrow \mathsf{AF}_{\leq 3}\ \mathsf{alarme})$$

Pour interpréter des formules de TCTL sur des automates temporisés, il faut ajouter dans leur définition un étiquetage des états par des propositions atomiques. On en déduit alors de manière naturelle un étiquetage du système de transitions temporisé associé : si P appartient à l'ensemble des propositions atomiques d'un état q alors

pour toute configuration du système de transitions de la forme (q, v), la proposition atomique P appartient aussi à $\ell((q, v))$. Dans ce contexte, on dit qu'un automate temporisé $\mathcal{A}$ satisfait une formule Ψ de TCTL, noté $\mathcal{A} \models \Psi$ si la configuration initiale $s_0 = (q_0, \mathbf{0})$ du système de transitions temporisé associé à $\mathcal{A}$ satisfait Ψ.

Il a été prouvé dans [ALU 93] que le *model checking* de TCTL est décidable pour les automates temporisés : étant donnés un automate temporisé $\mathcal{A}$ et une formule Ψ de TCTL, il existe un algorithme (PSPACE-complet) décidant si $\mathcal{A} \models \Psi$. L'outil KRONOS [DAW 96] implante (une version optimisée de) cet algorithme, tandis qu'UPPAAL [LAR 97a] permet de vérifier à la volée des propriétés d'un fragment de TCTL.

Autres logiques temporisées. Le même type de procédé a été utilisé pour temporiser d'autres logiques temporelles. Ainsi, la logique LTL du temps linéaire a été étendue [KOY 90] au cadre temporisé, produisant MTL, dont le *model checking* s'est malheureusement avéré être indécidable. Pour contourner cette difficulté, des fragments suffisament expressifs ont été étudiés par la suite [ALU 96, OUA 05], avec des résultats positifs. De manière similaire, des opérateurs avec contraintes quantitatives ont été ajoutés à un fragment du μ-calcul, donnant la logique L_ν [LAR 95, ACE 02]. De plus, la logique TCTL a elle-même été étendue, soit avec des paramètres [BRU 03], soit avec des modalités permettant la vérification de propriétés « presque partout » [Bel 05].

Dans une direction un peu différente, pour tenter de pallier l'indécidabilité des problèmes de vérification pour les automates hybrides, d'autres extensions des automates temporisés ont été proposées, associant par exemple des coûts aux états ou aux transitions [BOU 04]. La logique WCTL [BRI 04] a alors été définie pour être interprétée sur ces modèles, en ajoutant des contraintes de coût aux modalités de CTL. A nouveau, il a été montré que le *model checking* de WCTL est indécidable, et des fragments moins expressifs ont été recherchés.

4.4.2. *Équivalences de modèles*

Remarquons, à partir de ce qui précède, que la propriété initialement considérée « toute occurrence d'un problème est suivie par une alarme dans un délai inférieur ou égal à 3 unités de temps » s'exprime aisément par la formule Φ (définie plus haut) de la logique TCTL. Elle pourrait aussi s'exprimer par un modèle, par exemple par une version améliorée $\mathcal{B}$ de l'automate temporisé $\mathcal{A}_3$ (paragraphe 4.3.3). Supposons maintenant que nous nous intéressions à un système plus compliqué, modélisé par un automate temporisé $\mathcal{S}$, et faisant également intervenir les actions p et a correspondant aux occurrences respectives d'un problème et d'une alarme. Plutôt que de traiter le problème par *model checking*, on peut envisager de comparer les deux modèles $\mathcal{S}$ et $\mathcal{B}$, afin de décider si tous les comportements de $\mathcal{S}$ satisfont la spécification exprimée

par $\mathcal{B}$. C'est dans ce but qu'ont été introduites les équivalences entre modèles, étendues par la suite au cadre temporisé. Notons cependant qu'il n'est pas toujours possible de représenter une propriété de TCTL par un modèle [ACE 03].

Simulation forte. Considérons deux systèmes de transitions temporisés $\mathcal{T}_1 = (L, S_1, s_0^1, E_1)$ et $\mathcal{T}_2 = (L, S_2, s_0^2, E_2)$, sur le même ensemble d'étiquettes $L = \Sigma_\varepsilon \cup \mathbb{T}$.

DÉFINITION 4.6 *Une relation binaire $\mathcal{R}$ sur $S_1 \times S_2$ est une* t-simulation forte *de $\mathcal{T}_1$ par $\mathcal{T}_2$ si (s_0^1, s_0^2) appartient à $\mathcal{R}$ et :*

1) si (s_1, d, s_1') est dans E_1 pour une durée d et $s_1 \mathcal{R} s_2$ alors il existe s_2' dans S_2 tel que (s_2, d, s_2') est dans E_2 et $s_1' \mathcal{R} s_2'$,

2) si (s_1, a, s_1') est dans E_1 pour une action a et $s_1 \mathcal{R} s_2$ alors il existe s_2' dans S_2 tel que (s_2, a, s_2') est dans E_2 et $s_1' \mathcal{R} s_2'$.

Le STT $\mathcal{T}_2$ t-simule fortement $\mathcal{T}_1$ s'il existe une t-simulation forte de $\mathcal{T}_1$ par $\mathcal{T}_2$. Lorsqu'une relation $\mathcal{R}$ est une t-simulation forte de $\mathcal{T}_1$ par $\mathcal{T}_2$ et que la relation inverse $\mathcal{R}^{-1}$ (pour laquelle $(s_2, s_1) \in \mathcal{R}^{-1}$ ssi $(s_1, s_2) \in \mathcal{R}$) est aussi une t-simulation forte de $\mathcal{T}_2$ par $\mathcal{T}_1$, on dit que la relation $\mathcal{R}$ est une t-bisimulation forte. Les deux systèmes $\mathcal{T}_1$ et $\mathcal{T}_2$ sont alors dits *fortement t-bisimilaires*.

Comme la simulation dans le cadre non temporisé, cette correspondance exige que $\mathcal{T}_2$ puisse « imiter » $\mathcal{T}_1$ au plus près. Les transitions de durée sont considérées comme des transitions ordinaires.

Simulation faible. La t-simulation forte est un peu trop exigeante et peut être assouplie en tenant compte des ε-transitions et des décompositions possibles des durées. À partir d'un STT $\mathcal{T} = (L, S, s_0, E)$, on définit la relation de transition $\Rightarrow \subseteq S \times (\Sigma \cup \mathbb{T}) \times S$ pour $a \in \Sigma$ et $d \in \mathbb{T}$ par :

– $s \stackrel{d}{\Rightarrow} s'$ s'il existe une exécution de $\mathcal{T}$ de s à s' de durée totale d, ne comportant que des ε-transitions,

– $s \stackrel{a}{\Rightarrow} s'$ s'il existe une exécution de $\mathcal{T}$ de s à s' de durée nulle, ne contenant que a comme action visible.

DÉFINITION 4.7 *Une relation binaire $\mathcal{R}$ sur $S_1 \times S_2$ est une* t-simulation faible *de $\mathcal{T}_1$ par $\mathcal{T}_2$ si (s_0^1, s_0^2) appartient à $\mathcal{R}$ et si, pour toute transition $s_1 \stackrel{\ell}{\Rightarrow} s_1'$ de E_1, avec $\ell \in \Sigma \cup \mathbb{T}$, telle que $s_1 \mathcal{R} s_2$, il existe s_2' dans S_2 tel que $s_2 \stackrel{\ell}{\Rightarrow} s_2'$) est dans E_2 et $s_1' \mathcal{R} s_2'$.*

Le STT $\mathcal{T}_2$ t-simule faiblement $\mathcal{T}_1$ s'il existe une t-simulation faible de $\mathcal{T}_1$ par $\mathcal{T}_2$. De même que ci-dessus, lorsque $\mathcal{R}$ est symétrique, on dit que la relation $\mathcal{R}$ est une t-bisimulation faible. Les deux systèmes $\mathcal{T}_1$ et $\mathcal{T}_2$ sont alors dits *faiblement t-bisimilaires*.

Cette notion est bien adaptée aux systèmes temporisés et elle a donné lieu à de nombreux travaux comparant le pouvoir d'expression de divers modèles temporisés [ČER 93, BER 05c, BER 05a].

Autres équivalences. On peut affaiblir encore le type d'équivalence par une relation faisant abstraction du temps, en remplaçant toute transition de délai d par un seul type de transition, représentant un successeur dans le temps. Bien que cette opération fasse perdre les informations quantitatives, elle possède des propriétés intéressantes vis-à-vis de la vérification [LAR 97b]. Enfin, en ajoutant des conditions d'acceptation, on peut aussi considérer l'inclusion ou l'équivalence de langages : deux STT $\mathcal{T}_1$ et $\mathcal{T}_2$ sont tl-équivalents s'ils acceptent le même langage temporisé. Des comparaisons ont aussi été menées pour cette équivalence entre les réseaux de Petri temporels et les automates temporisés [BER 05c].

4.5. Conclusion

Dans ce chapitre, nous avons illustré les mécanismes de « temporisation de modèles » sur de nombreux exemples : le principe consiste à munir un modèle classique comme un automate fini, un réseau de Petri ou une structure de Kripke, de contraintes quantitatives sur les délais de franchissement des transitions.

Nous avons également montré comment ajouter de telles contraintes à des formules de logique temporelle et comment adapter à ce cadre des notions comme la bisimulation. Les travaux des quinze dernières années sur ces approches de temporisation ont été extrêmement fructueux et ont permis de réelles avancées pour la modélisation, la conception et l'analyse de systèmes dans lesquels ces contraintes de temps ont une importance cruciale. En particulier, ils ont présenté de nombreuses études de cas, menées avec les outils HYTECH [HEN 95], KRONOS [DAW 96] ou UPPAAL [LAR 97a], conduisant à la détection d'erreurs liées aux délais et à la synthèse de paramètres corrects.

Au-delà de ces succès, de nombreuses questions restent ouvertes, à la fois sur la théorie des modèles temporisés et sur les aspects pratiques liés à la conception de grands systèmes. Classiquement, la taille des objets étudiés, qui augmente de façon significative avec l'ajout du temps [LAR 00], reste un obstacle important dans les processus de conception et de vérification.

4.6. Bibliographie

[ACE 02] ACETO L., LAROUSSINIE F., « Is Your Model Checker on Time ? On the Complexity of Model Checking for Timed Modal Logics », *Journal of Logic and Algebraic Programming*, vol. 52-53, p. 7-51, Elsevier Science Publishers, 2002.

[ACE 03] ACETO L., BOUYER P., BURGUEÑO A., LARSEN K. G., « The Power of Reachability Testing for Timed Automata », *Theoretical Computer Science*, vol. 300, n°1–3, p. 411–475, 2003.

[ALU 90] ALUR R., DILL D., « Automata for modeling real-time systems », *Proc. of ICALP'90*, vol. 443 de *LNCS*, Springer, p. 322-335, 1990.

[ALU 91] ALUR R., HENZINGER T., « Logics and models of real time : a survey », *Real-time : Theory in practice, Proc. REX workshop*, vol. 600 de *LNCS*, Springer, p. 74-106, 1991.

[ALU 93] ALUR R., COURCOUBETIS C., DILL D., « Model-Checking in Dense Real-Time », *Information and Computation*, vol. 104, n°1, p. 2–34, Academic Press, 1993.

[ALU 94] ALUR R., DILL D., « A theory of timed automata », *Theoretical Computer Science*, vol. 126, p. 183-235, 1994.

[ALU 96] ALUR R., FEDER T., HENZINGER T. A., « The Benefits of Relaxing Punctuality », *Journal of the ACM*, vol. 43, n°1, p. 116–146, ACM, 1996.

[ALU 97] ALUR R., HENZINGER T. A., « Real-time system = discrete system + clock variables », *Software Tools for Technology Transfer*, vol. 1, p. 86–109, 1997.

[ASA 97] ASARIN E., CASPI P., MALER O., « A Kleene theorem for timed automata », *Proc. of LICS'97*, IEEE Comp. Soc. Press, p. 160–171, 1997.

[ASA 02] ASARIN E., CASPI P., MALER O., « Timed Regular Expressions », *Journal of the ACM*, vol. 49, n°2, p. 172–206, 2002.

[BAE 01] BAETEN J. C. M., MIDDELBURG C. A., « Process algebra with timing : real time and discrete time », BERGSTRA J. A., PONSE A., SMOLKA S. A., Eds., *Handbook of Process Algebra*, Chapitre 10, p. 627–684, Elsevier, 2001.

[Bel 05] BELMOKADEM H., BÉRARD B., BOUYER P., LAROUSSINIE F., « A New Modality for Almost Everywhere Propeties in Timed Automata », *Proc. 16th Int. Conf. on Concurrency Theory (CONCUR'05)*, vol. 3653 de *LNCS*, Springer, p. 110–124, 2005.

[BER 00] BERARD B., PICARONNY C., « Accepting Zeno words : a way toward timed refinements », *Acta Informatica*, vol. 37, n°1, p. 45–81, 2000.

[BER 05a] BERARD B., CASSEZ F., HADDAD S., LIME D., ROUX O., « When are timed automata weakly timed bisimilar to time Petri nets ? », *Proc. 25th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'05)* , vol. 3821 de *LNCS*, Springer, 2005.

[BER 05b] BERARD B., CASSEZ F., HADDAD S., LIME D., ROUX O., « Comparison of Different Semantics for Time Petri Nets », *Proc. 3rd Int. Symp. on Automated Technology for Verification and Analysis (ATVA'05)*, vol. 3707 de *LNCS*, p. 293–307, 2005.

[BER 05c] BERARD B., CASSEZ F., HADDAD S., ROUX O., LIME D., « Comparison of the Expressiveness of Timed Automata and Time Petri Nets », *Proc. 3rd Int. Conf. on Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, vol. 3829 de *LNCS*, Springer, p. 211–225, 2005.

[BER 06] BERARD B., GASTIN P., PETIT A., Timed substitutions for regular signal-event languages, Rapport n°06.04, LSV, CNRS & ENS de Cachan, 2006.

[BOR 97] BORNOT S., SIFAKIS J., « Relating Time Progress and Deadlines in Hybrid Systems », *Proc. Int. Workshop Hybrid and Real-Time Systems (HART'97)*, vol. 1201 de *LNCS*, Springer, p. 286–300, 1997.

[BOR 98a] BORNOT S., SIFAKIS J., « On the composition of hybrid systems », *Proc. of Hybrid Systems : Computation and Control*, vol. 1386 de *LNCS*, Springer, p. 49–63, 1998.

[BOR 98b] BORNOT S., SIFAKIS J., TRIPAKIS S., « Modeling Urgency in Timed Systems », *Proc. Int. Symp. Compositionality : The Significant Difference (COMPOS'97)*, vol. 1536 de *LNCS*, Springer, p. 103–129, 1998.

[BOR 00] BORNOT S., GÖSSLER G., SIFAKIS J., « On the construction of live timed systems », *Proc. of TACAS'00*, vol. 1785 de *LNCS*, Springer, p. 109–126, 2000.

[BOU 99] BOUYER P., PETIT A., « Decomposition and composition of timed automata », *Proc. 26th Int. Coll. Automata, Languages, and Programming (ICALP'99)*, vol. 1644 de *LNCS*, Springer, p. 210–219, 1999.

[BOU 01] BOUYER P., PETIT A., THÉRIEN D., « An Algebraic Characterization of Data and Timed Languages », *Proc. 12th Int. Conf. Concurrency Theory (CONCUR'2001)*, vol. 2154 de *LNCS*, Springer, p. 248–261, 2001.

[BOU 02] BOUYER P., PETIT A., « A Kleene/Büchi-like Theorem for Clock Languages », *Journal of Automata, Languages and Combinatorics*, vol. 7, n°2, p. 167–186, 2002.

[BOU 04] BOUYER P., BRINKSMA E., LARSEN K. G., « Staying Alive As Cheaply As Possible », *Proc. 7th Int. Conf. on Hybrid Systems : Computation and Control (HSCC'04)*, vol. 2993 de *LNCS*, Springer, p. 203-218, 2004.

[BRI 04] BRIHAYE T., BRUYÈRE V., RASKIN J., « Model-Checking for Weighted Timed Automata », *Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System (FORMATS+FTRTFT'04)*, vol. 3253 de *LNCS*, Springer, p. 277–292, 2004.

[BRU 03] BRUYÈRE V., DALL'OLIO E., RASKIN J., « Durations, Parametric Model-Checking in Timed Automata with Presburger Arithmetic », *Proc. 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS'03)*, vol. 2607 de *LNCS*, Springer, p. 687–698, 2003.

[CAM 95] CAMPOS S., CLARKE E., « Real-time symbolic model checking for discrete time models », *Theories and Experiences for Real-Time System Development*, vol. 2 de *AMAST Series in Computing*, World Scientific, p. 129–145, 1995.

[ČER 93] ČERĀNS K., « Decidability of bisimulation equivalence for parallel timer processes », *Proc. 4th Int. Workshop on Computer Aided Verification (CAV'92)*, vol. 663 de *LNCS*, Springer, p. 302–315, 1993.

[COR 99] CORRADINI F., D'ORTENZIO D., INVERARDI P., « On the relationships among four Timed Process Algebras », *Fundamenta Informaticae*, vol. 38, n°4, p. 377–395, 1999.

[COR 03] CORRADINI F., DI COLA D., « The expressive power of urgent, lazy and busy-waiting actions in Timed Processes », *Mathematical Structures in Computer Science*, vol. 13, n°4, p. 619–656, 2003.

[DAW 96] DAWS C., OLIVERO A., TRIPAKIS S., YOVINE S., « The Tool KRONOS », *Proc. Hybrid Systems III : Verification and Control (1995)*, vol. 1066 de *LNCS*, Springer, p. 208–219, 1996.

[DIL 89] DILL D. L., « Timing assumptions and verification of finite-state concurrent systems », *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89)*, vol. 407 de *LNCS*, Springer, p. 197–212, 1989.

[DIM 00] DIMA C., « Real-Time Automata and the Kleene Algebra of Sets of Real Numbers », *Proc. 17th Symp. on Theoretical Aspects of Computer Science (STACS'2000)*, vol. 1770 de *LNCS*, Springer, p. 279–289, 2000.

[EME 92] EMERSON E. A., MOK A. K., SISTLA A. P., SRINIVASAN J., « Quantitative Temporal Reasoning », *Real-Time Systems*, vol. 4, n°4, p. 331–352, 1992.

[GRO 95] GROSSMAN R., LARSON R., « An algebraic approach to hybrid systems », *Theoretical Computer Science*, vol. 138, p. 101–112, 1995.

[HAN 95] HANSEN M., PANDYA P., ZHOU C., « Finite divergence », *Theoretical Computer Science*, vol. 138, p. 113–139, 1995.

[HEN 92] HENZINGER T. A., MANNA Z., PNUELI A., « What good are digital clocks ? », *Proc. 19th Int. Coll. Automata, Languages, and Programming (ICALP'92)*, vol. 623 de *LNCS*, Springer, p. 545–558, 1992.

[HEN 94a] HENZINGER T. A., MANNA Z., PNUELI A., « Temporal Proof Methodologies for Timed Transition Systems », *Information and Computation*, vol. 112, n°2, p. 273–337, 1994.

[HEN 94b] HENZINGER T., NICOLLIN X., SIFAKIS J., YOVINE S., « Symbolic model checking for real-time systems », *Information and Computation*, vol. 111(2), p. 193-244, 1994.

[HEN 95] HENZINGER T. A., HO P., WONG-TOI H., « HYTECH : the Next Generation », *Proc. 16th IEEE Real-Time Systems Symposium (RTSS'95)*, IEEE Computer Society Press, p. 56–65, 1995.

[HEN 96] HENZINGER T., « The Theory of Hybrid Automata », *Proc. of LICS'96*, New Brunswick, New Jersey, p. 278 -292, 1996, Invited tutorial.

[HEN 98] HENZINGER T. A., RASKIN J.-F., SCHOBBENS P.-Y., « The regular real-time languages », *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98)*, vol. 1443 de *LNCS*, Springer, p. 580–591, 1998.

[JAF 91] JAFFE M., LEVESON N., HEIMDAHL M., MELHARD B., « Software requirements analysis for real-time process-control systems », *IEEE Trans. Software Eng.*, vol. 17, p. 241–258, 1991.

[JON 77] JONES N., LANDWEBER L., LIEN Y., « Complexity of some problems in Petri nets », *Theoretical Computer Science*, vol. 4, p. 277–299, 1977.

[KOY 90] KOYMANS R., « Specifying Real-Time Properties with Metric Temporal Logic », *Real-Time Systems*, vol. 2, n°4, p. 255–299, 1990.

[LAB 98] LABROUE A., Conditions de vivacité dans les automates temporisés, Rapport n°LSV-98-7, Lab. Specification and Verification, ENS de Cachan, Cachan, France, Sept. 1998, 60 pages.

[LAR 95] LAROUSSINIE F., LARSEN K. G., WEISE C., « From Timed Automata to Logic – and Back », *Proc. 20th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'95)*, vol. 969 de *LNCS*, Springer, p. 529–539, 1995.

[LAR 97a] LARSEN K. G., PETTERSSON P., YI W., « UPPAAL in a Nutshell », *Journal of Software Tools for Technology Transfer (STTT)*, vol. 1, n°1–2, p. 134-152, Springer, 1997.

[LAR 97b] LARSEN K. G., WANG YI, « Time-Abstracted Bisimulation : Implicit Specifications and Decidability », *Information and Computation*, vol. 134, n°2, p. 75–101, mai 1997.

[LAR 00] LAROUSSINIE F., SCHNOEBELEN PH., « The state explosion problem from trace to bisimulation equivalence », *Proc. 3rd Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2000)*, vol. 1784 de *LNCS*, Springer, p. 192–207, 2000.

[LAR 02] LAROUSSINIE F., MARKEY N., SCHNOEBELEN PH., « On Model Checking Durational Kripke Structures (Extended Abstract) », *Proc. 5th Int. Conf. on Foundations of Software Science and Computation Structures (FoSSaCS'02)*, vol. 2303 de *LNCS*, Springer, p. 264–279, 2002.

[MAL 91] MALER O., MANNA Z., PNUELI A., « From timed to hybrid systems », *Real-Time : Theory in Practice, Proc. REX Workshop*, vol. 600 de *LNCS*, Springer, p. 447–484, 1991.

[MER 74] MERLIN P. M., A Study of the Recoverability of Computing Systems, PhD thesis, UCI, Univ. California, Irvine, 1974.

[NIC 91a] NICOLLIN X., SIFAKIS J., « An Overview and Synthesis on Timed Process Algebras », *Real-Time : Theory in Practice, Proc. REX Workshop*, vol. 600 de *LNCS*, Springer, p. 526–548, 1991.

[NIC 91b] NICOLLIN X., SIFAKIS J., YOVINE S., « From ATP to timed graphs and hybrid systems », *Real-time : Theory in practice, Proc. REX workshop*, n° 600LNCS, Springer, p. 549-572, 1991.

[NIC 92] NICOLLIN X., SIFAKIS J., YOVINE S., « Compiling real-time specifications into extended automata », *IEEE Trans. Software Eng.*, vol. 18, n°9, p. 794–804, 1992.

[NIC 93] NICOLLIN X., SIFAKIS J., YOVINE S., « From ATP to Timed Graphs and Hybrid Systems », *Acta Informatica*, vol. 30, n°2, p. 181–202, 1993.

[OUA 03] OUAKNINE J., WORRELL J., « Timed CSP= closed timed epsilon-automata », *Nordic Journal of Computing*, vol. 10, 2003.

[OUA 05] OUAKNINE J., WORRELL J., « On the decidability of Metric Temporal Logic », *Proc. 20th IEEE Symposium on Logic in Computer Science (LICS'05)*, 2005.

[PNU 77] PNUELI A., « The Temporal Logic of Programs », *Proc. 18th IEEE Symp. Foundations of Computer Science (FOCS'77)*, p. 46–57, 1977.

[SIF 96] SIFAKIS J., YOVINE S., « Compositional specification of timed systems », *Proc. 13th Annual Symposium on Theoretical Computer Science (STACS'96)*, vol. 1046 de *LNCS*, Springer, p. 347–359, 1996.

[Wan 91] WANG YI, « CCS + time= an interleaving model for Real-Time Systems », *Proc. 18th Int. Coll. Automata, Languages, and Programming (ICALP'91)*, vol. 510 de *LNCS*, Springer, 1991.

Chapitre 5

Langages de description d'architecture

Ce chapitre présente un panorama des langages de description d'architecture. Nous en étudions tout d'abord les principes généraux (section 5.2). Nous traitons ensuite des languages de description tels que Wright, orientés vers l'analyse des systèmes (section 5.3). Nous finissons par l'étude des languages destinés à l'implantation de systèmes (section 5.4), et plus particulièrement AADL.

5.1. Introduction

L'une des principales difficultés rencontrées lors du développement de systèmes répartis ou distribués réside dans leur complexité. Celle-ci peut être due au nombre d'entités en jeu, à la nécessité de prendre en compte la spécification des interactions (communications) entre parties du système et les problèmes qui y sont relatifs (interblocages, famines, etc.), ou encore plus généralement à la taille de ces systèmes. Pour résoudre cette complexité, des approches basées sur un découpage du système en sous-systèmes peuvent être mises en œuvre.

Derrière ce découpage se cache la notion de *structuration* qui a connu plusieurs instanciations au cours du temps : approches modulaires [PAR 72], approches à objets [MEY 91] et plus récemment approches basées composants [SZY 98, OUS 05] ou aspects [FIL 05]. Les *composants* présentent l'intérêt d'expliciter l'ensemble des services fournis par un élément réutilisable (ce qui était déjà le cas avec les modules et les objets) mais aussi d'expliciter les services requis par ce dernier ; ils sont donc définis

Chapitre rédigé par Pascal POIZAT et Thomas VERGNAUD.

de façon indépendante de leur contexte d'utilisation, ce qui les rend plus réutilisables (en théorie [VAL 00]) que les objets.

Ces approches conduisent cependant toutes à ce que la conception d'un système passe par deux tâches distinctes :

– « *programming in-the-large* », la structuration du système en un ensemble de sous-systèmes reliés par des relations de dépendance ou d'interaction, sorte de plan structurel du système. Ce plan peut, par analogie avec le domaine de la construction où les bâtiments sont construits à partir d'un plan d'assemblage de sous-parties (sol, murs, toît par exemple), être qualifié de *plan architectural* et la personne en charge de cette activité être appelée *architecte logiciel* ;

– « *programming in-the-small* », la conception puis l'implantation, ou la réutilisation, de sous-systèmes répondant à un sous-ensemble des fonctionnalités du système ou bien encore correspondant aux unités d'implantation.

Dans la suite nous adopterons le terme générique de *composants* pour ces sous-systèmes qui peuvent être assemblés pour constituer des systèmes plus grands.

Pour maîtriser la complexité il est par ailleurs nécessaire de pouvoir décrire le plan architectural d'un système à différents niveaux d'abstraction, allant d'un modèle très abstrait et proche du découpage mental de l'architecte logiciel (par exemple un ensemble de sous-systèmes communicants) jusqu'à l'implantation de ce système sur une architecture plus concrète (et donc utilisant un intergiciel). Il est aussi possible de spécifier le système sous différentes vues (toujours par anologie avec l'architecture de bâtiments, ces vues peuvent correspondre au plan structurel — murs, toît — au plan électrique — prises, câblage — au plan de plomberie, etc.). Dans le cadre du logiciel ce principe a par exemple été repris avec succès par UML (vue statique avec les diagrammes de classes et d'objets, vue dynamique avec les diagrammes d'états et d'activités, vue interactions avec les diagrammes de séquence et de collaboration, etc.).

Le développement des *langages de description d'architectures* (*Architectural Description Languages*, ADL) [PER 92, GAR 93, MED 00] correspond à l'application du principe qui consiste à utiliser différents langages pour réaliser différentes activités. Leur objectif est de servir de support à l'architecte logiciel en répondant à un certain nombre de fonctionnalités parmi lesquelles :

– définir l'*architecture* d'un système comme la composition d'éléments de base, de façon abstraite et ce indépendemment de la réalisation concrète (implantation) de ces derniers ; définir de façon explicite les interactions ([PER 92] parle plus généralement de relations) entre ces éléments ;

– supporter une phase d'analyse architecturale qui se place entre l'analyse des besoins (l'architecture sert de support à l'analyse du respect des exigences) et la conception (l'architecture sert de support à la conception en proposant un découpage du système) ; supporter à la fois une approche descendante (servir de support à la décomposition lors de la conception du système) et une approche ascendante (servir de support à la construction d'un système par assemblage d'éléments préalablement définis et réutilisables) ;

– fournir une sémantique bien définie au plan architectural (et ne pas se limiter à une description à base « de boîtes et de lignes ») ;

– permettre l'analyse de l'architecture, soit par l'utilisation de critères liés aux respects de contraintes de style (par exemple le couplage minimal entre composants), soit par l'utilisation de techniques formelles (par exemple la vérification de l'absence d'interblocage dans une architecture) auquel cas l'architecture permet alors l'analyse compositionnelle ;

– aider à l'implantation du système, par exemple en permettant la construction automatique du code des interactions entre composants du système.

Le développement des ADL s'est fait en deux étapes. Dans un premier temps, le monde universitaire s'est intéressé à la définition d'ADL dédiés soit à la vérification des architectures logicielles, soit à l'obtention ou l'aide à l'implantation à partir d'un plan architectural. Cela a conduit à un foisonnement d'ADL au cours des années 1990 : Aesop, C2/SADL, Darwin, OLAN, MetaH, Rapide, UniCon, Wright, pour n'en citer que quelques uns. Ils ne s'intéressent en général pas à l'ensemble des problématiques liées aux architectures mais à certains aspects uniquement : analyse et vérification (Darwin, Rapide, Wright), définition d'architectures liées à des styles architecturaux (Aesop), raffinement d'architectures (C2/SADL), lien avec l'implantation (Darwin, OLAN), définition de code permettant le recollement, les interactions entre composants existants (C2/SADL,UniCon), ...

Ces ADL, dits de première génération, ont actuellement atteint leur maturité. La nécessité de définir des formats d'échange entre ADL, formats définis autour de l'acceptation commune d'un ensemble de concepts partagés par tous les ADL, est rapidement apparue. Le principal résultat est Acme [ABL], langage pivot entre ADL développé à Carnegie Mellon et son atelier AcmeStudio.

Depuis, la recherche autour des ADL s'intéresse entre autres à l'extension des propriétés prises en compte dans la vérification des architectures (comme l'analyse de performance), aux particularités des systèmes spécifiés (comme les systèmes ayant une topologie architecturale dynamique ou les lignes de produits), ou encore aux relations entre ADL et UML 1.x [MED 02] puis UML 2.0 (même si avec l'avènement d'UML 2.0 des mécanismes de description architecturale ont été pris en compte à la base dans UML). Dans le même temps, la communauté industrielle s'est intéressée à l'application des ADL, principalement dans le contexte des systèmes embarqués.

L'aboutissement de cet intérêt est le développement d'AADL, qui définit un standard pour la description d'architectures de systèmes embarqués temps réel.

Dans ce chapitre nous présentons d'abord, dans la section 5.2, les grands concepts et principes communs à l'ensemble des ADL. Les deux sections suivantes sont dédiées à la présentation des ADL sous deux angles complémentaires. La section 5.3 présente les approches plus formelles d'ADL développées principalement dans le cadre universitaire au cours des années 1990. Il s'agit d'ADL souvent plus simples, mettant en jeu moins de concepts, et qui s'intéressent aux architectures sous l'angle de leur conception. Leurs concepts, bien fondés mathématiquement, permettent l'analyse et la vérification des architectures. La section 5.4 présente de son côté une classe d'ADL plus dédiés à l'implantation des systèmes distribués. Dans ce cadre nous nous intéresserons principalement à AADL. En conclusion, nous rappellerons les apports des ADL et montrerons rapidement comment ces deux types d'ADL peuvent se rencontrer avec les développements formels autour d'AADL.

5.2. Concepts

De nombreux ADL ont été proposés pour la modélisation d'architectures logicielles, soit dans un but général, soit dédiées à un domaine particulier d'application. S'il n'existe pas encore vraiment de consensus sur le niveau d'abstraction et de description auquel devrait se situer un ADL, un accord semble toutefois émerger sur un ensemble minimal commun de concepts que l'on peut retrouver dans une grande partie des ADL.

Cette *ontologie* des ADL a conduit à la définition de (méta-) modèles et de formats d'échange comme Acme [ABL] ou bien encore xADL [DAS 01]. Dans [MED 00], Medvidovic et Taylor développent un ensemble de concepts et de critères associés qui permettent de classifier les ADL. Cette étude est reprise comme base, ici comme dans l'ensemble des travaux qui visent à présenter les ADL dits de *première génération*, qui sont ceux que nous présentons dans cette section et dans la section sur les ADL formels. Les ADL de *seconde génération* ont comme représentants AADL ou ArchJava qui sont présentés en section 5.4.

La définition d'un ADL fait intervenir les éléments constitutifs de la description d'une architecture, les caractéristiques applicables à ces éléments et enfin des concepts additionnels.

5.2.1. *Éléments constitutifs de la description d'une architecture*

Les éléments à la base d'une description architecturale sont les *composants*, les *connecteurs* et *les configurations*. Ce sont ces concepts qui guident la construction

d'un système par construction ou réutilisation de composants puis composition de composants liés par des connecteurs au sein de configurations. Un vrai ADL doit donc permettre de modéliser explicitement ces trois concepts.

Composants. Une définition couramment acceptée de composant est celle de Szyperski [SZY 98] : « *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component [...] is subject to composition by third parties.* ».

Un composant est une unité de calcul ou de stockage de données. Leur taille n'est pas imposée et il peut s'agir tant de procédures ou de librairies de fonctions que de systèmes de gestion de base de données ou d'applications entières. Chaque composant peut nécessiter son propre espace d'exécution et de données ou bien les partager avec d'autres composants. La notion de composant se retrouve parfois sous d'autres termes, comme module, tâche ou processus.

L'élément clé est la notion d'*interfaces explicites*. L'interface d'un composant est l'unique moyen d'accès à ses fonctionnalités. Les interfaces sont ce qui permet d'inférer un minimum d'information sur une architecture logicielle. Sans elles, les descriptions architecturales ne seraient qu'un ensemble de boîtes et de lignes, sans aucune sémantique. L'association de propriétés aux composants (et à leurs interfaces) est importante pour la définition de cette sémantique. C'est cependant à double tranchant, l'ajout de propriétés complexes (données, informations temporelles ou stochastiques) rendant plus complexe l'analyse des architectures.

L'*interface d'un composant* est constituée d'un ensemble de *points d'interaction* (parfois appelés *ports*) avec son environnement extérieur, ou contexte d'utilisation. L'interface d'un composant définit explicitement non seulement les services qu'il fournit mais aussi, contrairement à des approches antérieures comme les objets, les services qu'il requiert. Une interface définit donc à la fois les contraintes d'utilisation du composant mais aussi un ensemble d'hypothèses faites sur son environnement. Les interfaces sont définies à l'aide de langages de définition d'interfaces (*Interface Description Languages*, IDL).

Les *services* peuvent correspondre à des opérations (procédures, fonctions), des messages (dans une approche de composants objets) ou encore à des variables partagées. La description d'un service consiste en général en son nom, son type (type des arguments et d'un éventuel retour) et le cas échéant une liste d'exceptions. Il s'agit du niveau minimal de description des interfaces, on parle aussi d'interfaces de niveau signature.

L'expérience a montré [VAL 00] que ce niveau, actuellement celui des modèles et langages de composants industriels (comme CCM, EJB) est insuffisant pour assurer le bon fonctionnement d'architectures de composants. En effet, deux composants

compatibles du point de vue de leurs noms de services peuvent quand même se bloquer si leurs protocoles comportementaux (l'ordre dans lequel les services sont sensés être utilisés) sont incompatibles. De nombreux travaux universitaires ont conduit à prendre en compte des interfaces de plus haut niveau comme les interfaces comportementales décrites à l'aide de langages de description d'interfaces comportementales (*Behavioural Interface Description Language*s, BIDL). Ces travaux définissent ensuite des mécanismes de vérification [BER 03] et de correction [CAN 06] d'architectures logicielles en se basant sur ces descriptions comportementales.

Nous verrons dans la section 5.3 des représentants de ces approches, les ADL formels. Des niveaux de description d'interfaces plus élevés sont aussi possibles, par exemple pour prendre en compte la sémantique (fonctionnalité) des services ou bien encore des informations liées à la qualité de service (temps d'exécution ou ressources nécessaires au fonctionnement d'un service par exemple).

Connecteurs. Le connecteur est l'abstraction utilisée pour modéliser les interactions entre composants. Un connecteur décrit des correspondances entre interfaces de composants ainsi que les règles qui prévalent lors des interactions. La notion de connecteur peut s'instancier de façons très différentes comme par exemple communication par *pipe* dans le style *pipe-and-filter* sous Unix, par variables partagées ou espaces de tuples, appel de procédure distante (*Remote Procedure Call*, RPC) entre un client et un serveur, liaison JDBC entre un programme Java et un SGBD, ou encore abstraction de protocoles ou de conteneurs[1].

Les connecteurs sont équipés de *rôles* qui représentent les rôles que peuvent tenir différents composants dans l'interaction décrite par le connecteur. Ainsi par exemple dans une architecture *pipe-and-filter*, pour un connecteur de type *pipe*, on aura deux rôles : source, qui correspondra au composant — fichier ou processus — qui fournit les données, et puits, qui correspondra au composant — le filtre suivant ou un fichier — destinataire des données.

Du point de vue des connecteurs, il existe trois catégories d'ADL [OUS 05] : ceux qui ont des connections directes entre ports de composants (en général une sémantique très simple et unique est sous-jacente) ou décrivent les connecteurs implicitement (Darwin, Rapide), ceux qui ont des connecteurs définis en termes de combinaisons de connecteurs prédéfinis (Unicon) et enfin, les plus expressifs, les ADL qui ont un langage de définition de l'interaction spécifiée dans les connecteurs (Wright, Section 5.3).

1. Nous nous restreindrons dans la suite à des exemples d'utilisation d'ADL qui sont liés au domaine d'application du livre, c'est pourquoi les connecteurs correspondront en général à une abstraction d'un mécanisme de communication entre systèmes répartis.

La notion de connecteur est en général une notion abstraite qui ne correspond pas, contrairement à celle de composant, à un élément indépendant du système implémenté. On pourra cependant aussi trouver dans certains ADL la notion de connecteurs de première classe ou de composant-connecteur où les connecteurs peuvent être définis de façon plus explicite à l'aide de composants (réutilisables).

Configurations. Une *configuration* (architecturale), aussi appelée topologie, décrit un assemblage de composants et de connecteurs. Les composants et connecteurs correspondent à des instances nommées de types de composants et de connecteurs et ce de façon à permettre leur référencement. Les relations entre interfaces des composants et rôles des connecteurs sont établies à l'aide de correspondances appelées « liaisons » (*bindings*). Une configuration fournit une information structurelle, qui peut cependant, dans le cas de certains ADL, être amenée à évoluer (les liaisons changeant par exemple suite aux communications entre composants).

Les configurations sont un élément clé des ADL. Elles permettent la vérification des architectures. La correction des correspondances entre interfaces et ports peut être vérifiée en utilisant les informations fournies par ces derniers (correspondance de noms, protocoles compatibles, informations de qualité de service compatibles). Il est aussi possible d'appliquer des métriques de qualité aux architectures décrites par les configurations (une architecture favorisant les connections multiples entre composants est par exemple moins réutilisable qu'une architecture où les composants communiquent par couches comme par exemple l'architecture OSI des protocoles réseaux).

Les configurations permettent le déploiement automatique de l'architecture définie. Les configurations permettent aussi dans certains ADL de construire des *composants composites*, c'est-à-dire des composants non atomiques, constitués d'une configuration de composants et de connecteurs. Pour cela, il doit être possible de relier les ports des sous-composants à ceux du composite. L'intérêt des composants composites est de permettre la réutilisation de configurations complètes mais aussi de supporter le développement de techniques compositionnelles de vérification formelle (la vérification est effectuée au niveau du particulier, un constituant de l'architecture, puis réutilisée au niveau de la vérification globale).

Dans la figure 5.1 nous donnons un méta modèle qui reprend les concepts de base des ADL.

Dans la figure 5.2 nous donnons dans la syntaxe d'Acme un exemple d'architecture Client-Serveur avec communication de type RPC repris de [ABL] et dans la figure 5.3 un exemple d'architecture *pipe-and-filter* (recherche et tri des constantes définies dans un fichier `.h`) où un composant composite est défini à l'aide d'une configuration (notation inspirée d'Acme).

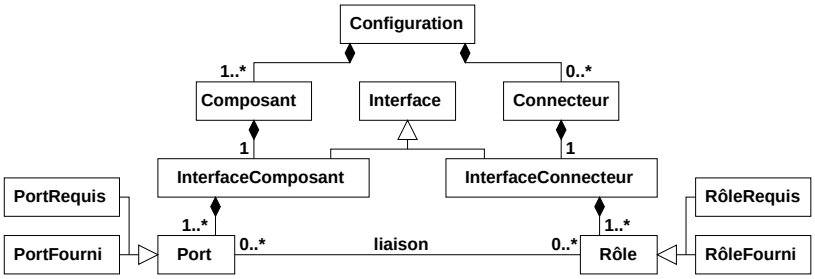


Figure 5.1. *Méta modèle simplifié des ADL*

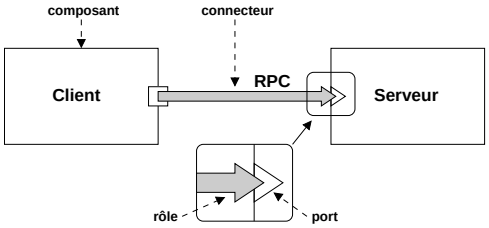


Figure 5.2. *Exemple d'architecture : client-serveur*

Caractéristiques des éléments architecturaux. Les éléments d'une architecture peuvent partager un certain nombre de caractéristiques :

Interfaces : les interfaces des composants correspondent à une définition explicite des services fournis et requis par un composant. Les interfaces des connecteurs correspondent aux différents rôles qu'un composant peut jouer dans une interaction. Les configurations peuvent aussi avoir des interfaces. Elles permettent aux configurations de communiquer entre elles dans le cas de couplage de sous-systèmes. Dans les trois cas (composants, connecteurs, configurations), une décomposition minimale des interfaces se fait entre ports (ou rôles) requis et ports (ou rôles) fournis. La vérification de la correction des liaisons entre ports fournis et rôles requis d'un côté et entre ports requis et rôles fournis de l'autre permet un premier niveau d'analyse de la correction des architectures.

Types : les composants, et parfois les connecteurs, peuvent naturellement être vus comme des types instanciables au sein de configurations architecturales. Comme en programmation, les types améliorent la compréhension et l'analyse des architectures. Les connecteurs n'ont de type (et ne sont donc instanciables) que lorsqu'il s'agit d'entités de première classe.

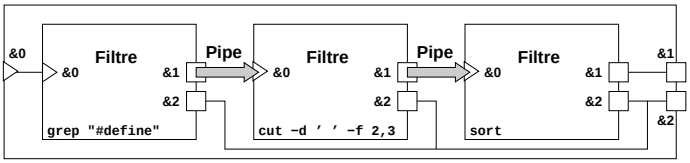


Figure 5.3. *Exemple d'architecture : pipe-and-filter*

Composition et hiérarchie : les éléments d'une configuration peuvent donner lieu à la définition d'un (type de) composant hiérarchique appelé composant composite ou composition. Cette possibilité permet d'améliorer la structuration de l'architecture décrite ou bien de s'y intéresser à différents niveaux de détail. La composition améliore aussi les possibilités de réutilisation, non seulement de types de composants et le cas échéant de connecteurs, mais aussi d'architectures complètes. Pour cela, il est indispensable que l'ADL permette la définition d'interfaces de configurations et la connexion de ces interfaces à celles des composants constituants de la configuration.

Propriétés comportementales (ou protocoles) : les ADL fournissent principalement une information statique (topologique) sur les architectures. Certains (il s'agit des ADL formels que nous détaillerons section 5.3) permettent de préciser les enchaînements possibles des suites de demandes et de fournitures de services par les composants et dans certains cas par les connecteurs. Les services fournis par un composant peuvent potentiellement ne pas toujours être disponibles (car dépendant de son état). Dans un tel cas, un composant qui essayerait d'accéder à ce service pourrait se bloquer. Plus généralement, des blocages mutuels entre composants peuvent survenir et causer un blocage général de l'architecture. La prise en compte de propriétés comportementales permet donc d'analyser les architectures pour s'assurer par exemple de l'absence de blocage. La notion de types peut être redéfinie pour prendre en compte non seulement les noms de services et leur direction (fournis/requis) mais aussi les protocoles.

Autres propriétés non fonctionnelles : il est possible d'associer d'autres propriétés non fonctionnelles (c'est-à-dire non directement liées aux services des composants ou à l'interaction décrite par un connecteur) aux constituants d'une architecture. Les propriétés choisies le sont généralement par rapport à un domaine d'application particulier des ADL. Ainsi, des propriétés de temps d'exécution ou de niveau de priorité pourront être utilisées dans le cas de description d'architectures embarquées temps-réel. La possibilité de définir des propriétés constitue un élément clé de l'extensibilité d'un ADL. Dans la section 5.4, nous verrons comment AADL utilise intensivement ces propriétés.

D'autres caractéristiques telles que la prise en compte de contraintes ou encore l'hétérogénéité et l'évolutivité des architectures sont détaillées dans [MED 00]. Les

caractéristiques liées à la sémantique et à la formalité des constituants architecturaux sont détaillées dans la section suivante ainsi que dans la section 5.3 qui porte sur les ADL formels .

5.2.2. *Concepts additionnels*

Les ADL se différencient par leur instanciation des concepts que nous avons vu précédemment. Ils se différencient aussi par leur gestion d'un certain nombre de concepts additionnels. Dans la cadre de cet ouvrage, nous présentons deux concepts additionnels qui nous semblent importants car liés à la possibilité de travailler sur des architectures définies à l'aide d'ADL qui ne soient pas (seulement) des boîtes et des lignes.

Styles architecturaux. Un style architectural est une notion qui n'existe que dans certains ADL. Un style architectural permet la définition d'un certain nombre de contraintes sur les architectures définies à l'aide d'un ADL comme :

Contraindre les types de composants et/ou les connecteurs utilisables dans l'architecture. Le style *pipe-and-filter* limite ainsi par exemple les composants à être une source de données (un fichier par exemple), un puits de données (une sortie écran par exemple) ou un filtre, tandis que le seul connecteur autorisé est le *pipe*.

Contraindre les liaisons possibles entre composants et connecteurs. Ces contraintes peuvent être spécifiées au niveau des types (par exemple un puits ne peut servir de source de données mais il est possible de connecter deux filtres par un *pipe*) ou au niveau plus précis des ports et des rôles (par exemple dans un style client-serveur, un port de sortie d'un client ne peut être connecté — via un connecteur — à plus d'un port d'entrée de serveur tandis que les ports de sortie des serveurs peuvent être connectés à plusieurs ports d'entrée de clients en mode diffusion). Des topologies spécifiques peuvent aussi être imposées (réseaux en anneau, en étoile, en grille).

Contraindre le nombre de composants et de connecteurs au sein d'une configuration. Les styles se basent ici en général sur l'idée qu'un trop grand nombre de constituants dans une configuration peut conduire à des problèmes ou limite sa réutilisabilité.

Le respect par une architecture des contraintes édictées par un style architectural peuvent être vérifiées à l'aide d'outils comme par exemple AcmeStudio. Lorsque l'ADL est un ADL formel qui prend aussi en compte des informations comportementales, il est possible d'associer des règles au niveau de styles architecturaux qui

assureront de bonnes propriétés dynamiques des architectures (comme l'absence de blocage par exemple).

De nombreux styles architecturaux sont présentés dans [GAR 93], comme le style *pipe-and-filter* et plus généralement flots de données, les types abstraits et l'orientation objet (les connecteurs étant des appels d'opération ou de procédure), le style basé évènements, les systèmes en couches, les systèmes à *dépôt* (*repository*) de données (connaissances, tableaux) et les systèmes distribués.

Formalité et outillage. Les ADL supportent tous la prise en compte de la sémantique des composants, même si cela se fait à différents degrés de formalité. Les ADL formels (section 5.3) permettent la spécification du comportement des composants soit par la prise en compte d'invariants et de pré-/post-conditions comme C2, soit d'expressions comportementales comme Rapide ou Wright qui utilisent respectivement les structures d'évènements et l'algèbre de processus CSP. Certains ADL se spécialisent sur l'interaction entre composants et leur composition, possiblement dynamique et évolutive, comme Darwin dont une sémantique a été donnée avec le π-calcul.

Une motivation importante derrière la définition d'ADL est la possibilité de faire supporter le développement d'architectures logicielles par des outils. Si la communauté universitaire a défini de nombreux ADL et outils associés (par exemple AcmeStudio, ArchStudio, Software Architect's Assistant ou l'environnement de Wright), il est cependant possible de citer les grandes fonctionnalités qui se retrouvent (pour partie éventuellement) dans ces derniers :

Spécification de types de composants, de types de connecteurs, de configurations. Le support se fait habituellement à la fois sous une forme textuelle (plus adéquate pour l'échange) et sous une forme graphique (plus adéquate pour l'utilisateur, la spécification des liaisons, etc.). Des formats d'échange structurés (XML) sont aussi possibles.

Vues multiples : elles rendent possible la spécification et/ou la vue de différents aspects de l'architecture par les différents intervenants (architectes, développeurs, utilisateurs). Elles permettent aussi de voir les architectures à différents niveaux de détail, un modèle plus conceptuel ne nécessitant pas le même niveau de détail qu'un modèle plus proche de l'implantation ou dédié à l'interconnexion de composants existants.

Analyse : l'analyse des architectures logicielles a fait l'objet de nombreux travaux [BER 03, POI 04]. Au niveau purement statique il est possible de vérifier le respect de contraintes de connexion entre ports et rôles ou plus généralement le respect d'un style architectural. La prise en compte d'informations non fonctionnelles spécifiques, comme par exemple des comportements (section 5.3) ou

les extensions proposées par AADL (section 5.4), permet d'effectuer des analyses complémentaires.

Raffinement et implantation : le raffinement des architectures est une tâche complexe qui consiste à partir d'une architecture de plus haut niveau et la transformer de façon correcte vers une architecture proche de choix d'implantation. Les outils peuvent se baser sur des règles de transformation de modèle (remplacement d'un composant par un composant composite ou remplacement de connecteurs abstraits par des connecteurs liés à la plateforme de composants choisie par exemple) et éventuellement être supportés par des techniques formelles de vérification du raffinement (vérification que l'architecture raffinée a toujours certaines propriétés de l'architecture initiale). La traçabilité entre les exigences d'une architecture et leur implémentation est aussi une tâche qui peut être supportée par des outils. L'aide au raffinement peut être poussée jusqu'à aider à l'implantation des architectures par génération de code (code des composants et/ou code de déploiement et de communication entre composants).

5.3. Les ADL formels

Nous présentons ici les deux approches principales pour la spécification formelle d'architectures : Wright et l'approche qui a pour origine Darwin. Nous insisterons plus particulièrement sur Wright qui est très complet. Il serait long de présenter en détail l'ensemble des ADL formels. En effet, le principe même des ADL formels est souvent d'être construit pour répondre à la spécification et la vérification de propriétés particulières (non-blocage, propriétés temporelles, mobilité) d'un type particulier d'architecture (architectures abstraites en conception ou plus proches de l'implantation sur un support d'exécution donné). Pour plus de détails, il est possible de se référer à [BER 03, POI 04] par exemple.

5.3.1. *Intérêt des approches formelles pour les architectures logicielles*

La prise en compte de propriétés formelles associées aux interfaces de composants (définition formelle de protocoles sous la forme de systèmes de transitions, de réseaux de Petri ou d'algèbres de processus) permettent un certain nombre de moyens d'analyse outillables [POI 04] :

Les équivalences comportementales permettent de s'assurer que l'implantation d'un composant correspond à son interface ;

Les pré-ordres et relations de raffinement ou de sous-typage comportemental permettent de définir des substitutions correctes d'un composant ou d'un connecteur par un autre dans une architecture ;

La vérification de propriétés permet de s'assurer qu'une architecture globale satisfait à de bonnes propriétés (absence de blocage, disponibilité des composants, bornitude des files d'attente des composants) ;

La construction automatique d'adaptateurs qui, lorsqu'ils sont rajoutés à une architecture incorrecte (par exemple bloquante), pallient les problèmes.

Lorsque les équivalences mises en œuvre sont des congruences, un raisonnement compositionnel peut s'appliquer pour pallier la complexité de l'architecture. Globalement des techniques de vérification à la volée ou basées sur les symétries deviennent indispensables lorsque le nombre de composants est trop grand.

5.3.2. *Wright*

L'objectif de Wright [ALL 97b, ALL 97a] est la spécification et la vérification des architectures logicielles ainsi que le respect de styles architecturaux. Wright se base pour cela sur un langage qui peut être traduit en CSP et des vérifications qui sont faites avec l'outil FDR. Wright ne propose pas de générateur de code et la simulation n'est possible que dans les limites du langage CSP (donc avec les outils dédiés). Un reproche souvent fait à Wright est sa difficulté d'appréhension (concepts introduits, expressivité et difficulté inhérente à l'utilisation de CSP).

Connecteurs. Un connecteur est défini en Wright par un ensemble de rôles et une *glu* (*Glue* en anglais). Chaque rôle définit le comportement attendu du port connecté à ce rôle. La glu coordonne le comportement des rôles d'un connecteur en indiquant comment ils doivent se synchroniser ou s'entrelacer, les séquences possibles, etc.

Les comportements sont définis dans une syntaxe proche de CSP où les évènements sont surlignés lorsqu'ils sont sensés être initiés par l'objet (rôle, glu) en cours de définition, $\rightarrow$ désigne le préfixe, `STOP` désigne un processus qui ne peut rien faire, $\surd$ l'évènement de succès, § est défini comme $\surd \rightarrow$`STOP`, $\Box$ est le choix externe (le choix dépend de l'environnement et l'objet défini doit s'y plier) et $\sqcap$ est le choix indéterministe interne (les choix sont faits par l'objet défini). La notation permet de modéliser la réception (`ev?x`), l'émission (`ev!x`) et le paramétrage de processus ($\mathtt{P_v} = \mathtt{ev!v.P_{v+1}}$) par des données.

Ainsi, dans l'exemple (Remote Procedure Call) suivant :

```
Connector RPC
  Rôle Appelant = appel  →  retour  →  Appelant ⊓ §
  Rôle Appelé   = appel  →  retour  →  Appel  □ §
```

```
Glue = Appelant.appel  →  Appel.appel  →  Glue
       □ Appel.retour  →  Appelant.retour  →  Glue
       □ §
```

Il faut noter :

– les deux rôles possibles à tenir par rapport à ce connecteur, appelant ou appelé ;

– l'appelant décide d'initier la procédure ou pas (utilisation du choix interne), à l'inverse l'appelé offre l'option d'un appel de procédure mais ce sont les autres (ici l'appelant) qui décident si un appel aura lieu ou pas ;

– la glu qui synchronise les deux rôles, qui indique par préfixage à quel rôle font référence les évènements et qui, seule, donne une sémantique complète à un connecteur.

Composants. De façon similaire aux connecteurs, les composants sont définis par leurs ports (et leur comportement) et optionnellement par la description des interactions entre les ports (*computation*) :

```
Component Client
  Port Demande = appel  →  retour  →  Demande ⊓ §
  Computation = Demande.appel  →  Demande.retour  →  Computation ⊓ §
Component Serveur
  Port Service = appel  →  retour  →  Service □ §
  Computation = Service.appel  →  calculInterne
                →  Service.retour  →  Computation □ §
```

Configurations. Une configuration est la donnée d'un ensemble de types de composants, d'un ensemble de types de connecteurs, ainsi que des instances de ces types et des attachements entre ports et rôles.

Nous reprenons ici un exemple défini dans la thèse d'Allen [ALL 97b], celui d'une architecture *pipe-and-filter*, figure 5.4 (notation inspirée d'Acme), qui lit un flot de caractères sur son entrée et renvoie sur sa sortie le même flot mais avec un caractère sur deux en majuscules :

```
Configuration Capitalize
    Component SplitFilter [... définition ...]
    Component UpperCase   [... définition ...]
    Component MergeFilter [... définition ...]
```

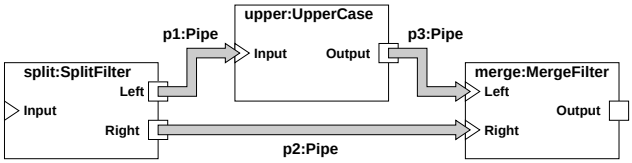


Figure 5.4. *Architecture d'exemple pour Wright*

```
    Connector Pipe        [... définition ...]
  Instances
    split:SplitFilter
    upper:UpperCase
    merge:MergeFilter
  Attachments
    split.Left   as p1.Source
    upper.Input  as p1.Sink
    split.Right  as p2.Source
    merge.Right  as p2.Sink
    upper.Output as p3.Source
    merge.Left   as p3.Sink
End Capitalize
```

Spécificités de Wright. Wright permet la définition :

– de composants composites. La partie *computation* d'un composant (ou la *glu* d'un connecteur) pouvant être définie par une description architecturale, on utilise une liste de *bindings* liant les ports des sous-composants aux ports du composite (ici en souligné) :

```
Configuration ClientServeur
  Connector RPC    [...]
  Component Client
    Port Demande = [...]
  Component Serveur
    Port Service = [...]
    Computation
      Configuration ServeurSur
        Component Coordinateur [...]
        Component GestSécurité [...]
        Instances
          coord:Coordinateur
          secman:SecurityManager
          cx:RPC
        Attachments
          coord.DemandeSecurité as cx.Appelant
```

```
          secman.ServiceSécurité as cx.Appelé
      End ServeurSur
    Bindings
      c.entrée = service
    End Bindings
  Instances
    c:Client
    s:Serveur
    rpc:RPC
  Attachments
    c.Demande as rpc.Appelant
    s.Service as rpc.Appelé
End ClientServeur
```

– de types d'interfaces qui définissent des patrons généraux d'interaction et peuvent être utilisés dans les ports et dans les rôles :

```
Interface Type DataInput = [...]
Interface Type DataOutput = [...]
Component UpperCase
  Port Input = DataInput
  Port Output = DataOutput
  Computation = [...]
```

– la paramétrisation des types par des comportements ou par des constantes pour paramétrer le nombre de ports (resp. de rôles) d'un composant (resp. d'un connecteur) ou de composants dans une configuration ;

– la définition de contraintes de style comme par exemple pour le style *pipe-and-filter* :

```
Style Pipe-and-Filter
   Connector Pipe [définition]
    Interface Type DataInput = [...]
    Interface Type DataOutput = [...]
Constraints
  ∀ c:Connectors . Type(c) = Pipe
  ∧ ∀ c:Components; p:Port | p∈ Ports(c) . Type(p) = DataInput
  ∨ Type(p) = DataOutput
    ...
```

Vérification. Les restrictions faites par Wright sur le sous-ensemble de CSP utilisé (le nombre de processus est toujours fini) rendent possible la vérification automatique des propriétés des architectures. Dans le cas de Wright, ces vérifications sont entre autres :

– pour les composants : cohérence entre les ports et le calcul d'un composant ;

– pour les connecteurs : absence de blocage d'un connecteur, absence de blocage d'un rôle ;

– pour les configurations : cohérence des liens port/rôle, complétude des attachements, respect des styles architecturaux ;

– pour les styles : cohérence de la définition du style.

D'autres vérifications sont spécifiques à Wright (initiateurs uniques pour les évènements, vérifications au niveau des paramètres et invervalles utilisés, etc.).

5.3.3. *Darwin et approches basées sur FSP et LTSA*

Darwin [MAG 96] est un ADL qui permet avant tout de décrire des configurations. C'est pourquoi Darwin est souvent qualifié de langage de configuration. L'un des intérêts de Darwin est sa prise en compte de la dynamique de la topologie des architectures (création/suppression de composants, modification des liaisons). Une sémantique de Darwin dans le π-calcul est donnée dans [MAG 95]. Contrairement à Wright, Darwin possède à la fois une notation textuelle et une notation graphique, utilisée dans l'outil Software Architect's Assistant.

Supposons `n` filtres à combiner en série pour obtenir un composant composite (la figure 5.3 en est un exemple). En Darwin cela s'écrit :

```
component filter {
  provide output<stream char>;
  require input<stream char>;
};

component pipeline(int n) {
  provide output;
  require input;
  array F[n]:filter;
  forall k:0..n-1 {
    inst F[k]@k+1;
    when k < n-1
      bind F[k+1].input -- F[k].output;
  }
  bind
    F[0].input -- input;
    output -- F[n-1].output;
}
```

Listing 5.1 – Connecteur en FSP

Plusieurs points sont à remarquer.

– Darwin est un ADL de plus bas niveau que Wright (utilisation des types C/C++ dans les définitions de configurations). Darwin utilise en fait des éléments syntaxiques qui le relient directement à Regis, un cadre (écrit en C++) permettant de construire et exécuter des programmes distribués spécifiés avec Darwin. Ainsi, dans l'exemple précédent, `<stream char>` désigne un port qui supporte des données de types caractères en mode flot ;

– Darwin permet l'instantiation de composants avec la primitive `inst` (`@` permettant de spécifier sur quelle machine) et la liaison dynamique entre ports (composant/-composant ou composite/sous-composant) avec `bind` ;

– en Darwin il n'y a pas de connecteurs explicites, ou plus exactement un nombre prédéfini de connecteurs possibles entre ports.

L'objectif premier de Darwin est l'implantation. L'approche Darwin a cependant été étendue dans le projet TRACTA [GIA 99, MAG 99b] pour prendre en compte le comportement des composants à l'aide de LTS puis de l'algèbre de processus FSP [MAG 99a], dialecte de CSP. Une architecture donnée Darwin étendue avec FSP peut alors être vérifiée à l'aide de l'outil LTSA. Ceci ne s'applique cependant que pour une architecture fixe dont la topologie ne varie pas.

Un exemple de spécification de composant en FSP est celle d'un appel de procédure RPC (`|` est le choix externe, `->` la séquence et `@` permet de définir l'interface du composant) :

```
range T = 0..1
RPC = (appelant.appel[x:T]->appele.appel[x]->RPC
      |appele.retour[x:T]->appelant.retour[x]->RPC
      )@{appelant.appel,appele.appel,
         appele.retour,appelant.retour}.
```

Listing 5.2 – Connecteur en FSP

Notons l'utilisation d'un composant-connecteur pour définir ce connecteur puisque Darwin n'a pas de constructeur explicite de connecteur. La construction d'architectures consiste à composer en parallèle les composants des configurations à l'aide des opérateurs de concurrence (`||`), de renommage (`/{nouveau-nom/ancien-nom}`) et de masquage (`\{...}`) de FSP :

```
range T = 0..1

RPC = (appelant.appel[x:T]->appele.appel[x]->RPC
      |appele.retour[x:T]->appelant.retour[x]->RPC
      )@{appelant.appel,appele.appel,
         appele.retour,appelant.retour}.
```

```
7
8  CLIENT = (demande[0]->reponse[x:T]->CLIENT).
9
10 SERVEUR = (query[x:T]->reply[(x+1)%2]->SERVEUR).
11
12 ||CLIENTSERVEUR = (appelant : (CLIENT
13                               /{appel/demande, retour/reponse})
14                 || appele :  (SERVEUR
15                               /{appel/query, retour/reply})
16                 || RPC)
17                 \{ appelant.appel,appelant.retour,
18                    appele.appel,appele.retour}.
```

Listing 5.3 – Configuration en FSP

L'outil LTSA permet ensuite la vérification d'architectures :

– par animation (l'utilisateur peut interagir avec les ports des configurations (vues comme des composants-composites) ;

– par vérification d'absence de blocages (une trace témoin est proposée en cas de blocage) ;

– il est possible de définir un automate particulier, appelé *property automata*, qui définit les traces légales pour un alphabet d'évènements donné et contient éventuellement des états `ERROR`. Cet automate est ensuite combiné en parallèle avec le processus correspondant à la configuration dans le but de procéder à une analyse d'atteignabilité (ou non) des états `ERROR` et à la vérification de la correspondance entre traces de la configuration et traces désirées. Cette technique permet de vérifier des propriétés de sécurité et de vivacité.

5.3.4. *Synthèse*

Les ADL formels, en se plaçant à un haut-niveau d'abstraction, permettent la spécification formelle et la vérification des descriptions architecturales. En se basant principalement sur des algèbres de processus comme le π-calcul, CSP ou FSP, ces ADL fournissent un cadre suffisemment expressif pour la description comportementale au niveau des composants et des connecteurs (comme avec Wright et les algèbres de processus simples, CSP et FSP) mais aussi pour la description d'architectures (topologies) elles-même dynamiques (comme Darwin avec le π-calcul). En contrepartie, ces ADL sont plus complexes à utiliser et sont parfois jugés trop abstraits pour permettre la description de concepts architecturaux de plus bas niveau comme ceux qui sont pris en compte dans les ADL tels qu'AADL (section suivante).

5.4. Les ADL d'implantation

Nous avons étudié dans la section précédente des ADL formels. Nous présentons ici quelques langages de description d'architecture conçus pour assister la configuration ou la génération de systèmes exécutables. Ils sont basés sur la définition et l'assemblage de composants, et supposent l'utilisation d'une bibliothèque permettant d'exécuter l'architecture décrite. Parmi ces langages, AADL est en pleine émergence et est encore absent de la littérature ; nous l'étudions donc tout particulièrement.

5.4.1. *UML*

UML (*Unified Modeling Language*) est issu des travaux de l'OMG [OMG 06]. Il peut être classé dans les ADL d'implantation dans la mesure où beaucoup d'outils ont été développés pour générer automatiquement des applications à partir de leur description. Compte-tenu du grand nombre d'ouvrages déjà disponibles sur le sujet, nous ne développerons cependant pas outre mesure cette section.

Présentation du langage. Dans sa version 2.0, UML définit un ensemble de onze syntaxes, chacune correspondant à un type de *diagramme* : diagramme de classes, de paquetages, de structures composites, de composants, de déploiement, de cas d'utilisation, d'états, d'activités et d'interactions. Ces diagrammes définissent des notations standard permettant de décrire tous les aspects de l'application à modéliser — tant architecturaux que comportementaux. Dans le cadre de ce chapitre, nous nous intéressons essentiellement aux diagrammes architecturaux.

Les diagrammes de classe sont les plus connus. Ils permettent de décrire l'organisation des éléments de l'application, et sont largement inspirés des structures que l'on retrouve dans les langages de programmation orientée objet. Les diagrammes de paquetages permettent d'organiser les classes en sous-ensembles logiques, et ainsi de structurer les éléments applicatifs. Les diagrammes d'objets décrivent les instances des classes. Ils permettent donc de figurer l'état de l'application à un instant donné.

Les diagrammes de composants se situent à un niveau supérieur : ils permettent de représenter l'architecture comme un ensemble d'éléments proposant des interfaces qu'il est possible de connecter. Un composant peut ainsi requérir une connexion avec un autre élément fournissant une interface adéquate ; cette notion est absente des diagrammes de classes.

Les diagrammes de structures composites permettent de décrire les relations entre les différentes entités qui ont pu par exemple être définies dans les diagrammes de composants. Les diagrammes de structures composites permettent donc typiquement de représenter les connecteurs et les interfaces (c'est-à-dire les ports) ; ils permettent de décrire comment les différentes entités architecturales interagissent.

Les diagrammes de déploiement permettent de décrire la façon dont les éléments de l'application doivent être installés. Ils permettent notamment de représenter les éléments matériels sur lesquels les éléments applicatifs vont s'exécuter.

Les autres diagrammes permettent de décrire le comportement des entités. UML est donc plus qu'un ADL, puisqu'il ne se limite pas à la description des architectures. Il propose ainsi un ensemble de syntaxes pour décrire tous les aspects des applications.

Particularisation de la syntaxe UML. Du fait qu'UML définit une syntaxe qui laisse une grande liberté pour la description des systèmes, il est souvent nécessaire d'enrichir les notations, afin de pouvoir préciser la sémantique des éléments. Ces enrichissements sont appelés *profils*. Ainsi, le profil SPT (*Scheduling, Performance and Time*) introduit des notions de qualité de service associées à la description des systèmes temps-réel (temps de réponse, tailles de files d'attente, etc.). Son successeur MARTE (*Modeling and Analysis of Real-Time and Embedded*) permet aussi de prendre en compte les contraintes liées aux systèmes embarqués. D'autres profils, comme Omega, intègrent l'usage de sémantiques formelles au sein d'une modélisation UML. Du fait de cette versatilité, la syntaxe UML est parfois utilisée comme notation de substitution pour d'autres ADL, comme AADL ou ROOM [RUM 99].

5.4.2. *AADL*

AADL (*Architecture Analysis & Design Language*) est un ADL issu des travaux de la SAE (*Society of Automotive Engineers*). Il s'agit d'une évolution d'un ADL plus ancien, MetaH [FEI 00]. La première version d'AADL date de 2004 [SAE04], et le langage évolue régulièrement depuis. Contrairement à UML, qui était initialement conçu pour décrire des systèmes d'information, AADL est explicitement axé sur la description des systèmes embarqués temps-réel. Bien que les champs d'application d'AADL soient à l'origine l'aéronautique et le spatial, de plus en plus d'acteurs de l'automobile s'y intéressent.

Le standard actuel d'AADL définit différentes syntaxes : une syntaxe textuelle, une représentation XML, et une syntaxe graphique. Un profil UML est également en cours de développement [SAE]. AADL a été créé avec le souci de faciliter l'interopérabilité des différents outils ; c'est pourquoi la syntaxe de référence est textuelle, semblable à un langage de programmation. La représentation XML permet de faciliter la création de parseurs pour des applications existantes. La notation graphique se pose en complément de la notation textuelle, pour faciliter la description des architectures ; elle permet une représentation beaucoup plus claire que le texte ou XML, mais demeure moins expressive.

Principaux éléments d'une modélisation AADL. Nous décrivons ici les principaux aspects du langage. Nous nous appuyons sur la syntaxe textuelle, qui est actuellement

la plus pratique à manipuler. Nous illustrons les différents aspects du langage par des exemples, de façon à montrer la description progressive d'un simple système constitué d'un capteur de pression relié à un calculateur.

Une modélisation en AADL repose sur une hiérarchie de composants connectés entre eux. Une description AADL consiste en la déclaration des composants.

Chaque composant a une interface (*component type*) à laquelle correspondent zéro, une ou plusieurs implantations (*component implementation*). AADL définit plusieurs catégories de composants, répartis en trois grandes familles :

– les composants logiciels définissent les éléments applicatifs de l'architecture ;

– les composants de la plate-forme d'exécution modélisent les éléments matériels ;

– les systèmes permettent de regrouper différents composants en entités logiques pour structurer l'architecture.

Il existe cinq catégories de composants logiciels. Les fils d'exécution (*threads*) sont les éléments logiciels actifs. Ils peuvent être comparés aux processus légers tels que définis dans les systèmes d'exploitation. Les groupes de *threads* permettent de regrouper des *threads* afin de former des hiérarchies. Les processus définissent des espaces mémoire dans lesquels s'exécutent les *threads* ; un processus doit contenir au moins un *thread*. Les sous-programmes modélisent les procédures telles que définies dans les langages de programmation impératifs : ils ne renvoient pas de résultats, mais peuvent avoir des paramètres d'entrée et/ou de sortie. Enfin, les données représentent les structures de données qui peuvent être stockées ou échangées entre les composants.

Il existe quatre catégories de composants de plate-forme. Les processeurs représentent des ensembles constitués d'un microprocesseur combiné à un ordonnanceur ; il s'agit donc d'un processeur associé à un système d'exploitation minimal. Les mémoires représentent tous les dispositifs de stockage : disque dur, mémoire vive, etc. Les dispositifs (*devices*) permettent de représenter des éléments dont la structure interne est ignorée ; il peut s'agir par exemple de capteurs dont on ne connaît que l'interface et les caractéristiques externes. Les bus modélisent toutes les sortes de réseaux ou de bus, depuis le simple fil jusqu'à un réseau de type Internet. Les bus doivent être branchés aux processeurs, mémoires et *devices* afin de transporter les communications entre ces composants.

Les systèmes ne correspondent pas à des éléments concrets, mais permettent de regouper les composants en entités logiques. Ainsi, l'unité centrale d'un ordinateur pourra être modélisée par un système regroupant un processeur, une mémoire et un bus pour les faire communiquer.

Le type d'un composant définit son interface tandis que l'implantation décrit sa structure interne. Une implantation fait donc toujours référence à un type existant,

```
1 processor i486
2 end i486;
3
4 processor implementation i486.Intel
5 end i486.Intel;
6
7 processor implementation i486.AMD
8 end i486.AMD;
```

Listing 5.4 – Types et implantations de composants AADL

même si l'ordre de déclaration n'importe pas : il est possible de déclarer une implantation avant le type correspondant. Le listing 5.4 illustre la relation entre les types et les implantations.

Il est par ailleurs possible d'étendre une déclaration de composant (type ou implantation) afin d'y ajouter ou d'en préciser des éléments (listing 5.5).

Les éléments d'interface (*features*) d'un composant sont déclarés dans son type. De cette façon, toutes les implantations d'un type de composant offrent la même interface aux autres composants ; les différentes implantations peuvent donc être interchangées. Il existe plusieurs sortes de *features*.

Les ports correspondent aux interfaces de communication des composants. Ils sont déclarés comme ports d'entrée, de sortie ou d'entrée/sortie. Les ports d'évènement (*event ports*) correspondent à la transmission d'un signal. Ils peuvent être assimilés aux signaux des systèmes d'exploitation. Ils peuvent également déclencher l'exécution des threads. Les ports de données (*data ports*) correspondent à la transmission de données. Contrairement aux ports d'évènements, ils ne déclenchent rien à la réception ; ils peuvent donc modéliser un registre mémoire mis à jour de façon asynchrone. Les ports d'évènement/données (*event data ports*) sont la synthèse des deux premiers types de ports : ils permettent de transporter des données associées à un évènement ; ils permettent donc de modéliser un message, la donnée étant celle transportée par le message et l'évènement signalant l'arrivée du message. Pour les sous-programmes, on parle de paramètres. Les paramètres ont une sémantique équivalente aux ports de données ou d'évènement/données.

Les accès à composants sont de deux types : pour les bus et les données. Dans les deux cas, ils sont déclarés comme étant des accès fournis (*provides*) ou requis (*requires*). Dans le cas d'un composant de données, un accès permet de modéliser une variable partagée ; l'accès à un bus permet de représenter l'interconnexion entre les processeurs, les dispositifs et les mémoires.

Les groupes de ports (*port groups*) permettent d'agréger plusieurs *features*. Ils peuvent être comparés à des câbles parallèles.

Les sous-programmes peuvent également être déclarés dans l'interface d'un composant (un *thread* ou un processus) pour modéliser un appel de sous-programme distant.

```
1  data donnee
2  end donnee;
3
4  process prog
5  features
6    a : in data port;
7  end prog;
8
9  process programme extends prog
10 features
11   a : refined to in event data port donnee;
12 end programme;
```

Listing 5.5 – Exemples d'interfaces

Les implantations de composants peuvent contenir des *sous-clauses* (*subclauses*), comme illustré sur le listing 5.6.

L'implantation d'un composant peut contenir des sous-composants. Un sous-composant est l'instanciation de la déclaration d'un composant. De cette façon, une architecture modélisée en AADL est une arborescence d'instances de composants. La déclaration d'un sous-composant doit être associée à une catégorie de composant, et éventuellement à un type ou une implantation.

Les appels de sous-programmes modélisent les appels de procédures dans les langages impératifs. Ils sont regroupés en séquences d'appels dans les sous-programmes et les *threads*. Bien que la syntaxe utilisée soit semblable à la déclaration d'un sous-composant, un appel de sous-programme ne représente pas une instance de sous-programme.

Les connexions permettent de relier les *features* des différents sous-composants à celles d'autres sous-composants ou aux *features* du composant parent.

AADL décrit des architectures aux dimensions clairement définies : il n'est par exemple pas possible d'exprimer le fait qu'un composant puisse avoir un nombre quelconque de sous-composants. Le langage ne se limite pourtant pas à la description d'architectures statiques : les modes permettent d'introduire un aspect dynamique

dans les modèles AADL. Ils sont définis dans les implantations des composants et correspondent à des configurations de fonctionnement auxquelles peuvent être associés des sous-composants, des connexions, etc. AADL permet également de décrire les conditions de changement de mode, afin de définir une machine à états pour l'implantation du composant.

Une description AADL est basée sur un assemblage de sous-composants connectés entre eux ; la circulation des données est portée par les connexions, qui sont point-à-point. Afin de faciliter l'analyse de l'architecture, AADL permet de décrire des flux (*flows*) portés par les connexions. De cette façon, il est possible de décrire le cheminement logique des communications à travers toute l'architecture.

```
1  thread execution_thread
2  features
3    a : in event data port donnee;
4  end execution_thread;
5
6  thread implementation execution_thread.impl
7  calls
8    {appel1 : subprogram une_procedure;};
9  connections
10   parameter a -> appel1.a;
11 end execution_thread.impl;
12
13 subprogram une_procedure
14 features
15   a : in parameter donnee;
16 end une_procedure;
17
18 process implementation programme.mono_thread
19 subcomponents
20   fil1 : thread execution_thread.impl;
21 connections
22   event data port a -> fil1.a;
23 end programme.mono_thread;
```

Listing 5.6 – Implantation de composants

Les déclarations de *features*, sous-composants et appels de sous-programmes sont en général associés à des déclarations de composants. Il est cependant possible de ne spécifier que la catégorie du sous-composant ; la description est alors relativement imprécise mais syntaxiquement correcte, ce qui permet de décrire facilement une architecture dans les premiers stades de la conception. Il est ainsi possible de varier le degré de précision de la description. Les flux, modes et connexions correspondent

à des entités n'ayant aucune relation avec les déclarations extérieures au composant dans lequel ils sont déclarés.

Afin d'organiser les déclarations, le langage fournit la notion d'espace de nom (listing 5.7). Les espaces de noms comprennent l'espace de nom anonyme (*anonymous namespace*), qui est l'espace de nom par défaut, et les paquetages (*packages*). Les paquetages sont déclarés dans l'espace de nom anonyme, et ne peuvent pas eux-même contenir d'espaces de nom. Ils possèdent une partie publique et éventuellement une partie privée. Les déclarations contenues dans la partie publique peuvent être référencées depuis l'extérieur du paquetage, tandis que les déclarations faites dans la partie privée ne sont visibles que depuis le paquetage. Le nommage des paquetages AADL permet d'exprimer une hiérarchie ; il n'existe cependant pas de notion de sous-paquetage en tant que telle.

L'utilisation des paquetages permet ainsi de regrouper plusieurs déclarations AADL pour former des ensembles logiques, de la même façon que les paquetages de Java ou les espaces de noms du C++. Les paquetages structurent les déclarations tandis que systèmes structurent l'architecture.

```
1  package Capteurs
2  public
3    device un_capteur
4    features
5      res : requires bus access Reseau.Impl_A;
6    end un_capteur;
7
8    bus Reseau
9    end Reseau;
10
11   bus implementation Reseau.Impl_A
12   properties
13     Propagation_Delay => 1 ns .. 2 ms;
14   end Reseau.Impl_A;
15 end Capteurs;
```

Listing 5.7 – Paquetages

Les propriétés constituent un aspect fondamental d'AADL. Elles permettent d'exprimer les différentes caractéristiques des composants. Il est ainsi possible de décrire les contraintes s'appliquant à l'architecture. Par exemple, les propriétés sont utilisées pour spécifier le temps d'exécution théorique d'un sous-programme, la période d'un thread, le protocole de file d'attente utilisé pour un port d'évènement/donnée, etc.

Une propriété se définit par un nom, un type, la liste des éléments auxquels elle peut s'appliquer, et éventuellement une valeur par défaut (listing 5.8). Les déclarations

de propriétés sont regroupées dans des ensembles de propriétés (*property sets*), semblables aux paquetages. Il existe trois types de déclarations : les types de propriétés, les constantes et les noms de propriétés.

```
property set Utilisateur is
  Compiler : aadlstring => "gcc" applies to (subprogram, thread);
  Pressure : type units (Pa, hPa => Pa * 100);
  Pressure_Range : range of Pressure applies to (device);
  Version : aadlinteger applies to (all);
end Utilisateur;
```

Listing 5.8 – Déclarations de propriétés

Les associations de propriétés permettent d'attribuer une valeur à une propriété (c'est-à-dire associer une valeur à un nom de propriété). Les associations de propriétés peuvent intervenir dans trois situations différentes : déclarées dans un paquetage, déclarées dans la section « properties » d'un composant, ou directement attachées à une sous-clause (voir listing 5.9).

Si l'association est réalisée au niveau d'un paquetage, elle concerne tous les composants déclarés dans le paquetage correspondant à la déclaration `applies to` du nom de la propriété. Si l'association est réalisée au niveau d'un composant, elle concerne ce composant. La déclaration s'applique au composant en question, sauf si l'association contient le mot-clé `applies to` ; dans ce cas, la propriété s'applique à la déclaration désignée par `applies to`. Cette déclaration peut être une sous-clause du composant, ou une sous-clause du composant référencé par la sous-clause, et ainsi de suite, selon ce qu'indique le `applies to`. Si l'association est réalisée au niveau d'une sous-clause, elle concerne cette sous-clause. Si le mot-clé `applies to` est utilisé, la propriété s'applique à une sous-clause du composant référencé par la sous-clause. Il s'agit donc du même principe que pour le cas précédent, mais appliqué à une sous-clause au lieu de la déclaration d'un composant.

```
processor i486r extends i486
features
  res : requires bus access Capteurs::Reseau.Impl_A;
end i486r;

processor implementation i486r.Intel
end i486r.Intel;

system Calculateur
features
  res : requires bus access Capteurs::Reseau.Impl_A;
  e : in event data port donnees;
  config : in event port;
```

```
14 end system;
15
16 system implementation Calculateur.impl
17 subcomponents
18   proc : processor i486r.Intel {Clock_Period => 10 ns;};
19   appli1 : process programme.mono_thread;
20   appli2 : process programme.mono_thread in modes (double);
21 connections
22   bus access res -> proc.res;
23   event data port e -> appli1.a;
24   event data port e -> appli2.a in modes (double);
25 modes
26   simple : initial mode;
27   double : mode;
28   simple -[ bascule ]-> double;
29   double -[ bascule ]-> simple;
30 properties
31   Actual_Processor_Binding => reference proc applies to (appli1);
32   Actual_Processor_Binding => reference proc applies to (appli2)
33   in modes (double);
34 end Calculateur.impl;
```

Listing 5.9 – Associations de valeurs aux propriétés

Les annexes sont un autre moyen d'associer des informations aux éléments d'une description (voir listing 5.10). Elles permettent d'insérer, au niveau de la description des composants, des informations exprimées dans une syntaxe indépendante. Il est possible d'utiliser une annexe pour spécifier le comportement des composants ou d'exprimer des contraintes.

```
1 thread implementation execution_thread.impl2
2 extends execution_thread.impl
3 annex OCL {**
4   pre : 0 < a < MaxValue;
5 **};
6 end execution_thread.impl2;
```

Listing 5.10 – Annexes AADL

Les annexes et les propriétés permettent d'ajouter des informations à la description architecturale. Alors que les annexes ne concernent que les composants, les propriétés peuvent être associées à tous les éléments d'une description : composants, connexions,

features, etc. Les annexes permettent d'incorporer des éléments rédigés dans une syntaxe différente de celle d'AADL, facilitant ainsi la prise en charge par des outils existants. Les propriétés sont plus intégrées dans la syntaxe AADL, et donc plus adaptées pour décrire les caractéristiques des architectures.

La syntaxe d'AADL permet de décrire une suite de déclarations ; les sous-composants font référence à des instances des déclarations de composants. Afin d'obtenir la description architecturale en elle-même, il est nécessaire d'instancier les différentes déclarations, et ainsi obtenir une arborescence de composants instanciés. Pour cela, il faut définir un composant initial qui servira de racine à l'arborescence. Ce composant doit être un système ; il n'a pas d'interface, puisqu'il décrit l'ensemble de l'architecture. Un tel système est représenté au listing 5.11.

```
1  system global
2  end global
3
4  device Capteur_Pression extends Capteurs::un_capteur
5  features
6    s : out event data port donnees;
7  properties
8    Utilisateur::Pressure_Range => 10 Pa .. 5000 hPa;
9  end Capteur_Pression;
10
11 device Interrupteur
12 features
13   bascule : out event port;
14 end Interrupteur;
15
16 system implementation global.i
17 subcomponents
18   capt : device Capteur_Pression;
19   calc : system Calculateur.impl;
20   res : bus Reseau.Impl_A;
21   inter : device Interrupteur;
22 connections
23   bus access res -> capt.res;
24   bus access res -> calc.res;
25   dat_cnx : event data port capt.s -> calc.e;
26   basc : event port inter.bascule -> calc.bascule;
27 properties
28   Actual_Connection_Binding => reference res applies to (dat_cnx);
29   Actual_Connection_Binding => reference res applies to (basc);
30   Utilisateur::Version => 1;
```

```
31 end global.i;
```

Listing 5.11 – Le système global

Exploitation d'AADL.

AADL se caractérise par une approche de modélisation très concrète, avec pour objectif la génération effective des applications décrites.

AADL se focalise essentiellement sur les descriptions architecturales, et permet de modéliser les architectures avec une représentation unique (qui peut être exprimée selon plusieurs syntaxes). Par conséquent certains aspects, comme la modélisation du comportement des composants, n'entrent pas directement dans le champ d'application d'AADL. L'usage des propriétés et des annexes permet d'associer les descriptions comportementales aux composants, que ce soit en spécifiant les codes sources correspondants ou en utilisant des descriptions plus formelles.

Par ailleurs, la syntaxe d'AADL permet une grande souplesse : il n'est pas nécessaire de fournir tous les détails d'une architecture pour pouvoir en exploiter la description. Ainsi, il est inutile de spécifier la taille des codes sources si nous souhaitons juste vérifier l'ordonnançabilité : il suffira d'indiquer en propriété les temps d'exécution, les périodes des *threads*, etc. De la même façon, il n'est pas indispensable de préciser le type des sous-composants : on peut se contenter d'indiquer leur catégorie. Il est alors facile de décrire très simplement le déploiement d'une architecture répartie sur différents nœuds.

Par sa capacité à transporter toutes les informations nécessaires à la description des architectures, AADL peut servir de langage fédérateur pour un processus de conception et de génération : il est ainsi possible d'utiliser plusieurs outils pour tester l'ordonnançabilité, pour s'assurer du respect des contraintes de place mémoire, pour simuler l'architecture à partir des descriptions comportementales puis finalement pour générer l'architecture.

AADL peut donc servir de support pour une approche de conception basé sur du prototypage par raffinement [VER 05], représenté sur la figure 5.5. La première étape du développement est une modélisation architecturale assez abstraite, qui est ensuite raffinée jusqu'à obtenir une description précise. À chaque étape du raffinement il est possible de vérifier la validité théorique de l'architecture par simulation ou contrôle des propriétés. Ces différentes étapes aboutissent à une description concrète de l'application, qu'il est possible de générer automatiquement afin d'en tester le déploiement. En fonction des résultats des tests effectués sur le système généré, le modèle peut encore être modifié.

La génération d'un système exécutable à partir de sa description architecturale nécessite l'utilisation d'un exécutif qui pilotera le code généré à partir des composants

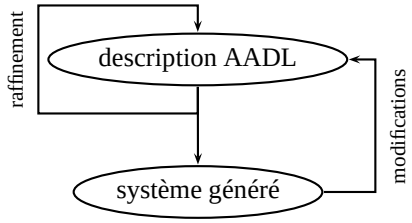


Figure 5.5. *Cycle de conception avec AADL*

AADL. Plusieurs stratégies sont possibles : soit utiliser un exécutif basé sur un intergiciel de communication, soit utiliser un exécutif minimal basé sur une bibliothèque de *threads* systèmes. L'utilisation d'un intergiciel permet de simplifier la gestion des communications entre les différents nœuds de l'application. En se reposant sur un intergiciel adaptable tel que PolyORB (voir chapitre 11, page 265), cette approche permet d'obtenir rapidement un prototype de l'application complète et convient donc aux premières étapes du prototypage. L'utilisation d'un exécutif minimal implique de décrire en AADL les composants assurant la gestion des communications. Dans ce cas, le travail du concepteur s'en trouve augmenté, mais la description architecturale est plus précise, et donc potentiellement plus facile à vérifier.

Outils pour AADL. AADL est avant tout un langage textuel ; la coopération entre différents outils spécialisés s'en trouve facilitée. Nous présentons ici quelques uns de ces outils.

OSATE [SAE] est une extension pour l'environnement de développement Eclipse. Elle fournit un éditeur de texte ainsi que divers outils d'analyse et de statistique. OSATE implante un méta-modèle d'AADL basé sur EMF, et facilite ainsi l'ajout d'autres extensions. L'une d'elles est issue du projet TopCased et fournit un éditeur graphique.

STOOD [ELL] est un outil de modélisation graphique basé sur la méthode HOOD. Il prend en charge les notations HOOD, UML et AADL et permet de créer du code Ada ou C à partir des modèles, ou au contraire de construire un modèle à partir de codes sources.

Ocarina [VER 06] est un compilateur pour AADL. Il permet de générer une application répartie en utilisant l'intergiciel PolyORB comme exécutif de haut niveau. Le frontal d'Ocarina peut être réutilisé et intégré dans d'autres applications pour leur permettre d'utiliser AADL.

Cheddar [SIN 04] est un simulateur d'ordonnanceur temps-réel. Il permet d'évaluer différents critères de performance (temps d'exécution, tailles des files d'attentes, etc.) s'appliquant à un système temps-réel.

5.4.3. *ADL spécialisés*

UML et AADL sont des langages évolués permettant de décrire une application répartie complète, tant du point de vue des composants logiciels que du déploiement de ces composants. Tous les ADL ne visent cependant pas un objectif aussi large : il existe de nombreux projets visant à utiliser un ADL comme langage de configuration logicielle.

Dans cette section nous présentons de tels langages de configuration. Bien que spécialisés, ils présentent tous une approche commune basée sur la connexion de composants, typique des ADL.

ArchJava. ArchJava [ALD 01] est une extension au langage Java destinée à associer les informations architecturales au code source. La démarche suivie est donc l'inverse de celle suivie par des langages comme AADL, lequel découple les deux aspects pour intégrer le code source dans la description architecturale.

Comme son nom le suggère, ArchJava s'appuie sur le langage Java. Il ajoute les notions de composants et de ports, de façon à structurer les applications : chaque composant définit des ports, lesquels constituent des interfaces fournissant et requérant des méthodes ou des variables. ArchJava permet ainsi de construire une application en connectant les ports des composants. Il facilite ainsi la conception de l'application. Étant couplé à un seul langage support, il ne permet pas d'assembler des composants écrits dans des langages différents. Le grand avantage de cette approche est qu'il relativement simple à utiliser.

Fractal ADL. Fractal est une approche d'assemblage de composants mise en place par le consortium ObjectWeb. Elle est basée sur la définition de conteneurs encapsulant le code des différents composants. Fractal repose ainsi sur une description de l'assemblage des composants de l'application, permettant la reconfiguration (c'est-à-dire l'évolution dynamique de l'assemblage des composants). Il est ainsi possible de décrire l'assemblage des composants à l'aide d'un langage spécialisé.

Fractal consiste en un exécutif et une interface de programmation pour différents langages. À ces éléments s'ajoute un langage de configuration. Par souci de généralité, les auteurs de Fractal n'imposent pas de langage de configuration particulier ; Fractal peut tout à fait être appliqué pour assembler des composants écrit dans des langages différents. Cependant, l'implantation de référence de Fractal (qui fournit un exécutif en Java) repose sur un ADL en syntaxe XML [BRU 04] pour décrire l'assemblage

des composants. Le champ d'application de ce langage est très restreint, puisqu'il est spécialisé pour la description des interfaces et de la composition des éléments de l'application.

SOFA. SOFA [PLá 98] est une architecture ayant pour objectif l'assemblage de composants réutilisables tout en permettant de faire évoluer ces composants de façon dynamique. SOFA fournit un langage pour décrire les interfaces des composants, qui sont alors désignés comme des *SOFAnodes* assemblés au sein d'un canevas (*SOFAnet*). Les composants actifs sont associés à un environnement d'exécution (DCUP) qui fournit un environnement à chaque nœud SOFAnode.

DCUP permet de séparer la partie fonctionnelle de la partie contrôle. Sur chaque nœud, la partie contrôle s'occupe de mettre à jour le composant, de contrôler son exécution et d'assurer l'interface entre le composant et les autres nœuds.

Principes communs. Fractal ADL, ArchJava et SOFA sont en fait des canevas pour la construction d'application. Ils reposent sur un environnement d'exécution déterminé, soit général (une machine virtuelle Java pour ArchJava) soit spécifique (Fractal et SOFA).

Dans tous les cas, la conception des applications repose sur un assemblage de composants, décrit à l'aide d'un langage de description d'architecture associé. Le langage décrit les interfaces des composants, ainsi que leurs connexions, reprenant ainsi les caractéristiques des ADL.

5.4.4. *Synthèse*

Les ADL d'implantations se caractérisent par une approche concrète de la modélisation : il s'agit à chaque fois d'utiliser le langage comme support pour la configuration, la génération ou la vérification du système à implanter. Pour assurer ces différentes opérations, le rôle de l'ADL n'est pas le même dans tous les cas.

Ainsi, l'approche suivie par UML consiste à définir un ensemble de syntaxes universelles qui doivent couvrir tous les aspects de la modélisation (tant architecturaux que comportementaux). On peut donc imaginer qu'à terme la syntaxe UML viendrait se substituer à toutes les autres syntaxes. Cet avantage se paye en termes de précision dans la sémantique de la description : pour être vraiment exploitable, UML doit souvent être associé à un profil ou à des extensions. Des langages comme AADL sont plus spécialisés, et offrent donc une sémantique plus précise. L'un des points forts d'AADL est qu'il peut être associé à d'autres langages, à travers les annexes ou les propriétés : AADL peut donc servir de langage fédérateur entre les différents aspects

de la modélisation, en conservant les formalismes existants. Dans les deux cas, la génération du système final fait intervenir un exécutif qui supportera le fonctionnement du système généré.

Les ADL peuvent aussi être utilisés dans un cadre beaucoup plus spécialisé, comme langage de configuration. C'est le cas de Fractal, ou SOFA, qui s'associent à des exécutifs particuliers. ArchJava est un cas intermédiaire, puisqu'il n'est pas attaché à une machine virtuelle Java particulière. Ces langages se limitent cependant à l'assemblage de composants, sans prendre en compte d'éventuels aspects formels.

5.5. Conclusion

Les langages de description d'architecture ont maintenant fait leurs preuves comme moyen de remédier à la complexité des systèmes. Ils peuvent servir de support à une analyse descendante (moyens de description de sous-composants constituant le système ainsi que des relations existantes entre aux), mais également à une analyse ascendante (réutilisation et composition de composants pré-existants par exemple).

Ils ont atteint leur maturité à la fin des années 1990 avec les ADL dits de première génération. Alors principalement développés par la communauté universitaire, ces ADL se sont tout naturellement intéressés aux aspects formels de la description des architectures. Si les concepts mis en jeu dans ces ADL sont souvent simples ou peu nombreux, ils sont le support à l'analyse formelle et outillée des architectures. Les années 2000 voient l'arrivée d'ADL dont l'objectif est d'être utilisables aussi dans le cadre industriel. Certains de ces ADL se basent sur des notations existantes et bien acceptées, comme UML. D'autres fournissent de nouvelles notations mettant à disposition de l'architecte logiciel de nombreux concepts proches du domaine d'application ; c'est le cas d'AADL pour les systèmes embarqués.

L'un des enjeux des années à venir est de faire le lien entre des ADL plus concis — et donc vérifiables — et des ADL plus expressifs, proches de l'implantation et donc permettant la génération de code mais trop complexes à vérifier dans l'ensemble de la notation.

L'approche MDA (raffinement de configurations architecturales abstraites et formelles en configurations architecturales de plus bas niveau et implantables) devrait fournir des solutions. Des approches mixtes comme COTRE, basé sur AADL, cherchent à étendre des ADL industriels avec ce qui a fait le succès des ADL universitaires ; ces approches pourraient aussi, dans un cadre donné (profil d'extension) amener des solutions.

5.6. Bibliographie

[ABL] ABLE GROUP, The Acme Architectural Description Language, `http://www.cs.cmu.edu/~acme/`.

[ALD 01] ALDRICH J., CHAMBERS C., NOTKIN D., « Component-Oriented Programming in ArchJava », *OOPSLA'01*, 2001.

[ALL 97a] ALLEN R. J., GARLAN D., « A Formal Basis for Architectural Connection », *ACM Transactions on Software Engineering and Methodology*, vol. 6, n°3, p. 213–249, 1997.

[ALL 97b] ALLEN R. J., A Formal Approach to Software Architecture, PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[BER 03] BERNARDO M., INVERARDI P., Eds., *Formal Methods for Software Architectures*, vol. 2804 de *Lecture Notes in Computer Science*, Springer Verlag, 2003.

[BRU 04] BRUNETON E., Developing with Fractal, Consortium ObjectWeb, mars 2004.

[CAN 06] CANAL C., MURILLO J. M., POIZAT P., « Software Adaptation », *L'Objet*, vol. 12, p. 9–31, 2006.

[DAS 01] DASHOFY E. M., VAN DER HOEK A., N. T. R., « A Highly-Extensible, XML-Based Architecture Description Language », *Working IEEE / IFIP Conference on Software Architecture (WICSA)*, IEEE Computer Society, p. 103–112, 2001.

[ELL] ELLIDISS, « STOOD », `http://www.ellidiss.com/stood.shtml`.

[FEI 00] FEILER P. H., LEWIS B., VESTAL S., Improving Predictability in Embedded Real-Time Systems, Rapport n°CMU/SEI-2000-SR-011, université Carnegie Mellon, décembre 2000, `http://la.sei.cmu.edu/publications`.

[FIL 05] FILMAN R. E., ELRAD T., CLARKE S., AKŞIT M., Eds., *Aspect-Oriented Software Development*, Addison-Wesley, 2005.

[GAR 93] GARLAN D., SHAW M., « *Advances in Software Engineering and Knowledge Engineering* », vol. 2 de *Series on Software Engineering and Knowledge Engineering*, Chapitre An Introduction to Software Architecture, p. 1–39, World Scientific Publishing, 1993.

[GIA 99] GIANNAKOPOULOU D., KRAMER J., CHEUNG S., « Behaviour Analysis of Distributed Systems Using the Tracta Approach », *Journal of Automated Software Engineering*, vol. 6, n°1, p. 7–35, 1999.

[MAG 95] MAGEE J., DULAY N., EISENBACH S., KRAMER J., « Specifying Distributed Software Architectures », *European Software Engineering Conference (ESEC)*, vol. 989 de *Lecture Notes in Computer Science*, Springer Verlag, p. 137–153, 1995.

[MAG 96] MAGEE J., KRAMER J., « Dynamic Structure in Software Architectures », *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, p. 3–14, 1996.

[MAG 99a] MAGEE J., KRAMER J., *Concurrency : State Models and Java Programs*, Wiley, 1999.

[MAG 99b] MAGEE J., KRAMER J., GIANNAKOPOULOU D., « Behaviour Analysis of Software Architectures », *First Working IFIP Conference on Software Architecture (WICSA)*, IFIP Conference Proceedings, Kluwer, p. 35–50, 1999.

[MED 00] MEDVIDOVIC N., TAYLOR R. R., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, n°1, p. 70–93, 2000.

[MED 02] MEDVIDOVIC N., ROSENBLUM D. S., REDMILES D. F., ROBBINS J. E., « Modeling Software Architectures in the Unified Modeling Language », *ACM Transactions on Software Engineering and Methodology*, vol. 11, n°1, p. 2–57, 2002.

[MEY 91] MEYER B., *Conception et programmation par objets*, InterEditions, 1991.

[OMG 06] OMG, « UML Resource Page », `http://www.uml.org`, 2006.

[OUS 05] OUSSALAH M., Ed., *Ingénierie des composants. Concepts, techniques et outils.*, Vuibert Informatique, 2005.

[PAR 72] PARNAS D. L., « On the Criteria to be Used in Decomposing Systems into Modules », *Communications of the ACM*, vol. 15, p. 1053–1058, 1972.

[PER 92] PERRY D. E., WOLF A. L., « Foundations for the Study of Software Architectures », *ACM SIGSOFT Software Engineering Notes*, vol. 17, n°4, p. 40–52, 1992.

[PLá 98] PLÁŠIL F., BÁLEK D., JANEČEK R., « SOFA/DCUP : Architecture for Component Trading and Dynamic Updating », *ICCDS'98*, IEEE, mai 1998.

[POI 04] POIZAT P., ROYER J.-C., SALAÜN G., « Formal Methods for Component Description, Coordination and Adaptation », *First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT)*, p. 89–100, 2004.

[RUM 99] RUMPE B., SCHOENMAKERS M., RADERMACHER A., SCHÜRR A., « UML + ROOM as a Standard ADL ? », *ICECCS'99*, 1999.

[SAE] SAE, « SAE AADL Information Site », `http://www.aadl.info`.

[SAE04] SAE, Architecture Analysis & Design Language (AADL), sept. 2004.

[SIN 04] SINGHOFF F., LEGRAND J., NANA L., MARCÉ L., « Cheddar : a Flexible Real Time Scheduling Framework », *ACM Ada Letters*, vol. 24, n°4, p. 1–8, 2004.

[SZY 98] SZYPERSKI C., *Component Software : Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[VAL 00] VALLECILLO A., HERNÁNDEZ J., TROYA J. M., « New issues in object interoperability », *Object-Oriented Technology*, vol. 1964 de *Lecture Notes in Computer Science*, p. 256–269, Springer Verlag, 2000.

[VER 05] VERGNAUD T., PAUTET L., KORDON F., HUGUES J., « Rapid Development Methodology for Customized Middleware », *RSP'05*, IEEE, juin 2005.

[VER 06] VERGNAUD T., ZALILA B., Ocarina, a Compiler for the AADL, `http://ocarina.enst.fr`, 2006.

DEUXIÈME PARTIE

Techniques de vérification

Chapitre 6

Panorama de la vérification

6.1. Introduction

La diversité des méthodes de vérification est un facteur de confusion pour l'ingénieur confronté au problème de choisir les techniques adéquates à l'analyse de son système. Ce chapitre a pour objectif de clarifier les bases d'un tel choix en discutant de trois questions fondamentales associées au processus de vérification :

– Comment choisir le modèle formel, support de la vérification ?

– Comment exprimer les propriétés attendues du système modélisé ?

– Quelles méthodes de vérification appliquer au modèle ?

Souvent le modélisateur est tenté de choisir un formalisme très expressif afin de capturer tous les détails de son système. Cependant, cette expressivité complique de manière significative le processus de vérification. La complexité de la méthode de vérification associée peut ainsi limiter dans la pratique les capacités de vérification à des systèmes « jouets ». Pire, la satisfaction des propriétés peut s'avérer un problème indécidable et nécessiter l'application de semi-algorithmes. De manière indépendante, le modélisateur souhaite vérifier certaines propriétés sans fixer les valeurs de certains paramètres (nombre de nœuds d'un réseau, nombre de ressources, etc.). Ceci conduit à des modèles paramétrés dont la vérification devient encore plus complexe.

La spécification des propriétés doit répondre à la question suivante. Comment définir le « bon » comportement d'un système ? Parmi les réponses possibles, on peut suggérer :

Chapitre rédigé par Serge HADDAD.

– une famille de propriétés dites *génériques* exprimant le comportement du modèle formel indépendamment du système modélisé (e.g. l'absence de blocage, le caractère fini, etc.) ;

– un langage de propriétés adapté aux systèmes dynamiques et plus particulièrement aux systèmes concurrents (e.g. une logique modale telle qu'une logique temporelle linéaire – LTL – ou arborescente – CTL) ;

– une équivalence de comportement avec un autre modèle représentant la spécification du comportement attendu (e.g. la bisimulation) ;

– la réponse à une séquence de tests (e.g. des tests de refus ou d'acceptation).

On distingue généralement les méthodes de vérification selon des critères élémentaires. A quels formalismes s'applique la méthode ? La méthode travaille-t-elle au niveau structurel (i.e. en analysant la description du modèle) ou au niveau comportemental (i.e. en construisant une représentation éventuellement partielle ou abstraite de son comportement) ? Le processus de vérification est-il entièrement ou partiellement automatique ? Quels types de propriétés la méthode est-elle capable de vérifier ?

Enfin, en vue de combiner différentes méthodes de verification, il est nécessaire de comprendre quel bénéfice une méthode tire des résultats d'une autre. De plus, cette combinaison influence aussi le processus de specification : par exemple le système pourrait être modélisé de manière très abstraite afin d'obtenir l'ensemble des réponses attendues à faible coût, puis il pourrait être itérativement raffiné en conservant à chaque étape le plus de résultats possibles de l'étape précédente.

Afin d'illustrer notre propos et devant la diversité des formalismes, nous nous focaliserons sur la famille des réseaux de Petri et de ses extensions. La suite du chapitre est organisée de la manière suivante. Dans la section 6.2, nous discuterons des critères de choix des modèles. Puis dans la section 6.3, nous introduirons les différentes manières d'exprimer les propriétés d'un système. Dans la section 6.4, nous examinons les principales catégories de méthodes de vérification. Enfin à la section 6.5, nous concluons ce chapitre par un survol des méthodes présentées dans cette partie du livre.

6.2. Modèles formels pour la vérification

6.2.1. *Expressivité versus décidabilité et complexité*

Les modèles dédiés à la vérification doivent répondre à deux objectifs contradictoires. D'une part, ils doivent être suffisamment expressifs afin que le passage du modèle de conception au modèle de vérification ne s'accompagne pas d'une trop grande abstraction. D'autre part, cette expressivité est nécessairement limitée afin d'obtenir des algorithmes efficaces de vérification. Le lecteur intéressé est encouragé

à consulter des ouvrages de référence sur la complexité des algorithmes comme par exemple [PAP 94].

Illustrons ce point à l'aide du problème de l'accessibilité dans les réseaux de Petri : étant donné un réseau, un marquage initial et un marquage final, il s'agit de décider s'il existe une séquence de franchissements dans le réseau conduisant du marquage initial au marquage final. Comme le nombre de marquages d'un réseau est infini, il n'est pas évident *a priori* que ce problème soit décidable. Cependant certaines propriétés telles que le caractère borné du réseau et le problème de couverture (une variante de l'accessibilité dans lequel on cherche à atteindre un marquage supérieur ou égal au marquage final) sont décidables à l'aide d'une construction conceptuellement simple, *le graphe de couverture* [KAR 69]. Aussi il est plausible que l'accessibilité soit décidable et de fait un tel algorithme existe [MAY 84, KOS 82]. Malheureusement cet algorithme n'est pas *primitif récursif*, autrement dit sa complexité est imprédictible et même sa mise en œuvre informatique est problématique ce qui explique qu'aucun outil actuel ne résout ce problème dans toute sa généralité.

Par la suite, les travaux de recherche se sont tournés dans deux directions opposées. D'une part, on restreint le modèle des réseaux de Petri afin de diminuer la complexité du problème. Ainsi dans les réseaux « réversibles » (i.e. où chaque transition est accompagnée d'une transition dont l'incidence arrière est l'incidence avant de la première et vice versa), l'accessibilité devient un problème *Expspace-complet* [MAY 82]. Dans les réseaux sains (i.e. où toute place d'un marquage accessible ne compte pas plus d'un jeton), l'accessibilité devient un problème *Pspace-complet* [CHE 95]. En se restreignant aux réseaux à choix libres (i.e. où deux transitions qui partagent une entrée, ont la même incidence arrière) sains et vivants, le problème est maintenant *NP-complet* [ESP 98]. Enfin pour les graphes d'évènements (i.e. où chaque place a une transition entrée et une transition sortie) l'accessibilité se résout en temps polynomial [COM 71].

D'un point de vue théorique, il est intéressant de cerner les frontières entre décidabilité et indécidabilité. Ainsi les réseaux de Petri à arcs inhibiteurs ont le pouvoir de calcul des machines de Turing et aucune propriété intéressante, dont l'accessibilité, n'est décidable. Cependant en se restreignant à un arc inhibiteur (ou à une structure particulière d'arcs inhibiteurs), ce problème reste décidable [REI 95]. Les arcs « reset » qui vident les places connectées lors du franchissement de la transition associée ne confèrent pas aux réseaux de Petri le pouvoir de calcul des machines de Turing. Aussi certaines propriétés restent décidables mais pas l'accessibilité [DUF 98]. Enfin les réseaux de Petri récursifs dont un état courant correspond à un arbre (dynamique) de « threads », chacun disposant de son marquage propre étendent considérablement le pouvoir d'expression des réseaux de Petri tout en conservant la décidabilité de l'accessibilité [HAD 99].

6.2.2. *Caractéristiques souhaitables d'un modèle*

Orthogonalement à la question précédente plutôt d'ordre théorique, se pose la question de l'adaptation du modèle au système à analyser. En effet, deux modèles équivalents du point de vue de l'expressivité, ne le sont souvent pas du point de vue de la facilité de la modélisation. De nombreux critères sont envisageables : la modularité, la hiérarchie, la représentation du temps, etc. Nous illustrons ce point avec deux phénomènes que nous souhaitons modéliser à l'aide des réseaux de Petri : l'occurrence de pannes et la gestion du temps.

Supposons qu'un sytème ait été modélisé par un réseau de Petri et qu'on souhaite étendre cette modélisation afin de représenter la possibilité à tout instant d'une panne qui ramène le système à son état initial. Si le réseau de Petri est borné et qu'on connait un sur-ensemble de l'ensemble d'accessibilité, une telle modélisation est théoriquement réalisable. Malheureusement elle requiert d'introduire une place complémentaire par place du réseau et surtout une transition par marquage potentiellement accessible. Si le réseau de Petri est non borné alors on démontre qu'une telle construction n'existe pas. A l'aide d'arcs « reset », la modélisation redevient possible. Nous en donnons les grandes lignes : on contrôle le fonctionnement nominal du réseau avec une place marquée ($normal$) qui boucle autour de chaque transition, l'occurrence de la panne est représentée par une transition qui démarque $normal$ et marque $panne$, enfin une deuxième transition vide les places du réseau, démarque $panne$, marque $normal$ et marque les places du réseau selon la valeur du marquage initial. Une construction similaire est aussi facilement réalisable à l'aide des réseaux de Petri récursifs.

Le temps s'introduit dans les réseaux de Petri de multiples façons. Nous nous limitons aux deux modèles les plus fréquemment utilisés et qui ont fait par ailleurs l'objet de nombreuses études théoriques : les réseaux de Petri temporels (TPN) [BER 91] et les réseaux de Petri temporisés (TdPN) [FRU 00]. Un intervalle temporel est attaché à chaque transition d'un TPN. Lorsqu'une transition devient franchissable, son franchissement ne peut avoir lieu que si le temps écoulé depuis sa sensibilisation appartient à l'intervalle. De plus, le temps ne s'écoule que si la durée de sensibilisation de toute transition ne quitte pas son intervalle de référence. Ce choix sémantique permet de modéliser le caractère *urgent* d'un phénomène comme l'expiration d'un « time-out ». Cependant, il ne permet pas de modéliser des situations ou l'écoulement du temps invaliderait le franchissement d'une transition. Ainsi il est démontré que certains automates temporisés ne sont équivalents à aucun TPN [BÉR 05]. Dans les TdPN, les jetons sont *âgés* et chaque arc est un multi-ensemble d'intervalles. Pour qu'une transition soit franchie, il faut sélectionner des jetons dont l'âge appartient à l'intervalle correspondant. Lorsque les jetons sont produits, leur âge initial est non déterministe mais doit appartenir à l'intervalle correspondant de l'arc.

À l'inverse des TPN, les TdPN sont capables d'exprimer l'inhibition d'une transition par le passage du temps mais incapables de modéliser l'urgence temporelle.

Signalons enfin que du point de vue de la vérification, toutes les propriétés deviennent indécidables pour les TPN non bornés alors que certaines propriétés restent décidables pour les TdPN comme la couverture (mais pas l'accessibilité) [ABD 01].

6.2.3. *De la conception à la vérification : la sémantique formelle*

Une fois le modèle choisi, il faut « traduire » le système exprimé à l'aide du modèle de conception. Idéalement cette phase doit être automatisée. Cependant cette automatisation requiert l'assurance que le comportement du nouveau modèle est équivalent à celui du modèle original (via une équivalence appropriée). Une des possibilités consiste à fournir une sémantique formelle (le plus souvent opérationelle) des constructeurs du modèle de conception en termes de sous-modèles du modèle de vérification.

Nous illustrons ce point à l'aide de deux modèles de réseaux de Petri de haut niveau : les réseaux de Petri colorés [JEN 97] et le « *Petri box calculus* » [KOU 94]. Les réseaux de Petri colorés sont définis par des domaines de couleur attachés aux nœuds du réseau et des fonctions de couleur attachés aux arcs allant du domaine de la transition au multi-ensemble des couleurs du domaine de la place. Il y a deux sémantiques (équivalentes) possibles pour un tel formalisme. Dans le premier cas, la règle de franchissement est définie directement en sélectionnant une transition et une couleur de son domaine, en évaluant les fonctions des arcs en entrée, en consommant les multi-ensembles de jetons des places en entrée, en évaluant les fonctions des arcs en sortie et en produisant les multi-ensembles de jetons des places en sortie.

Dans le deuxième cas, on définit un réseau de Petri dont les places sont des couples composés d'une place du réseau coloré et d'une couleur de son domaine et les transitions sont des couples similaires. En évaluant les fonctions pour les couleurs de transition et en projetant sur les couleurs des places, on détermine les valuations des arcs de ce réseau *déplié*. Le comportement du réseau coloré est alors défini comme celui du réseau déplié. Si du point de vue par exemple de la simulation, cette sémantique est *a priori* moins appropriée que la première, elle présente des avantages du point de vue de la vérification. Enoncer une proprité du réseau coloré revient à énoncer une propriété du réseau déplié. De plus, la validité des méthodes de vérification s'établit en prouvant qu'elles sont correctes au niveau du réseau déplié (e.g. les réductions de réseau).

Le *Petri box calculus* est un formalisme de type algèbre de processus (comme par exemple CCS [MIL 95]). L'une de ses sémantiques (équivalentes) consiste à construire un réseau de Petri à partir de sous-réseaux et de règles opérationnelles pour chaque opérateur (séquence, choix, exécution parallèle, etc.). L'intérêt de ce formalisme est double. Il permet de fournir une sémantique de véritable concurrence pour les algèbres de processus mais aussi de bénéficier des méthodes de vérification des

réseaux de Petri. Ainsi l'accessibilité indécidable en CCS devient décidable du fait du choix sémantique des réseaux de Petri. De même le calcul d'invariants qui est une technique propre aux réseaux de Petri est adaptée au *Petri box calculus.*

6.3. Expression des propriétés

L'expression des propriétés d'un modèle soulève le même problème que celui du choix du formalisme. S'appuyer sur un vaste ensemble de propriétés interdit le développement d'algorithmes specialisés efficaces tandis que resteindre cet ensemble de propriétés ne permet pas d'exprimer les propriétés variées des protocoles et des applications réparties.

6.3.1. *Propriétés génériques*

Si on choisit de restreindre l'ensemble des propriétés à vérifier alors celles-ci doivent être génériques au sens suivant : elles expriment le comportement du système modelisé dans des contextes les plus différents possibles. Examinons comment une telle interprétation est réalisable. Nous donnons ci-dessous une liste non exhaustive de propriétés qui visent cette généricité (toujours dans le cadre des réseaux de Petri) :

– **quasi-vivacité** « Toute transition est franchie au moins une fois » exprime une spécification syntaxiquement correcte au sens où toute activité ou évènement doit apparaître au moins une fois dans le comportement du système.

– **absence de blocage** « Il n'y a pas de marquage mort » signifie qu'un blocage global n'apparaît jamais dans le système.

– **vivacité** « Toute transition est franchie au moins une fois depuis n'importe quel marquage accessible » signifie que le système ne perd jamais ses capacités.

– **caractère borné** « Le marquage de toute place reste borné » assure que le système atteindra un comportement *stationnaire* sur le long terme. Notons que plusieurs comportements stationnaires sont possibles.

– **état d'accueil** « Il existe un marquage atteignable de tout marquage accessible » représente la possibilité pour le système de se réinitialiser.

– **état inévitable** « Il existe un marquage qui ne peut être évité indéfiniment » indique que le système doit nécessairement se réinitialiser.

6.3.2. *Propriétés spécifiques*

En dépit de la généralité des propriétés précédentes, Il y aura toujours des aspects du comportement du système qui ne pourront être exprimés par un ensemble fixé de propriétés. Par exemple, « Le franchissement de t_1 sera nécessairement suivi ultérieurement du franchissement de t_2 » est une propriété d'équité très fréquemment

recherchée et qui ne se réduit pas à une combinaison des propriétés précédentes. Evidemment, elle pourrait être incluse dans cette liste, mais de nombreuses variations sont possibles.

Aussi il est plus rationnel d'adopter un langage de propriétés adapté aux systèmes dynamiques et plus particulièrement aux systèmes concurrents. Parmi ces langages, le cadre des logiques temporelles (e.g. logique temporelle lineaire - LTL [SIS 85] -, logique temporelle arborescente - CTL [CLA 86], CTL* [EME 86] -, etc.) a été intensivement étudié. La raison de ce développement est double : la plupart des propriétés intéressantes de systèmes concurrents sont exprimées à l'aide de formules élementaires et les méthodes de vérification associées (appelées dans ce cas « *model checking* ») sont très faciles à développer.

Enfin la plupart des logiques temporelles usuelles sont relativement intuitives. Nous en donnons maintenant une description informelle. Les formules sont, soit des formules d'état, soit des formules de chemin. Une formule d'état élémentaire s'évalue localement sur l'état considéré. Dans le contexte des réseaux de Petri, ce pourrait être « Les places p_1 et p_2 sont simultanément marquées » ou encore « Le marquage de p_1 est strictement supérieur à celui de p_2 ». Puis on construit d'autres formules d'état avec des formules de type « Pour tout (resp. Il existe un) chemin d'exécution à partir de cet état, (resp. tel qu') une formule de chemin donnée est satisfaite ». Les formules de chemin élémentaires sont des formules d'état qui s'évaluent sur le premier état du chemin. Puis on construit d'autres formules de chemin avec des opérateurs de chemin comme « Le suffixe du chemin obtenu en supprimant le premier état vérifie une formule donnée de chemin » ou encore « Tout (resp. Il existe un) suffixe du chemin (resp. qui) vérifie une formule donnée de chemin ».

6.3.3. *Equivalence de modèles*

Le cadre de la logique temporelle est intéressant si on veut vérifier un ensemble de propriétés qui caractérise le comportement attendu du système. Cependant, partir d'un comportement global attendu spécifié par exemple par un ensemble de services nécessite un effort important afin de parvenir à déterminer les formules caractérisant ce comportement. De plus, le concepteur est amené à construire des formules de plus en plus complexes. La signification de telles formules devient alors mystérieux. Dans des cas pareils, il est beaucoup plus simple de représenter l'ensemble des services par un modèle (e.g. un réseau de Petri) et de comparer le comportement du modèle de services avec celui du modèle du protocole.

Néanmoins il est nécessaire de définir ce qu'est une équivalence entre modèles. Tout d'abord, on doit distinguer les transitions internes (implémentant le protocole) et les transitions externes (associées à l'interface du service). Une première équivalence

possible est alors l'équivalence des langages observables (i.e. les séquences d'exécution projetées sur les transitions externes). Cependant l'équivalence de langage ne capture pas les choix d'exécution *offerts* par un état. De nombreuses définitions d'équivalence relatives au language et aux choix ont été proposées dont la plus courante est la relation de bisimulation [MIL 95]. De manière grossière, la relation de bisimulation indique que les états des deux modèles peuvent être couplés de telle façon que quoique puisse faire l'un des modèles en partant de son état, l'autre modèle peut le faire et atteindre un état couplé avec le nouvel état du premier modèle.

Du point de la vérification, l'intérêt de la bisimulation est double :

– La vérification n'est pas plus complexe que la vérification de formules de logique temporelle lorqu'elle est appliquée aux graphes d'accessibilité (à condition que ceux-ci soient finis).

– Pour des modèles comme les algèbres de processus, l'axiomatisation de l'équivalence est possible au niveau structurel (i.e. à partir du modèle).

6.3.4. *Test de modèles*

Une autre possibilité (la dernière que nous examinerons ici) est la réponse d'un modèle à une séquence de tests. Une application typique de test peut être décrite ainsi :

1) Elle démarre avec le spécification des comportements acceptables ou d'échecs,

2) puis elle génère un objet intermédiaire prenant en compte les séquences à tester et les capacités de choix des états atteints,

3) cet objet est transformé en un système de transitions appelé testeur canonique,

4) le *produit synchrone* (i.e. l'exécution synchronisée sur les actions communes) du modèle étudié et du testeur canonique est construite,

5) l'analyse de la nature des états morts de ce produit fournit l'information recherchée sur le comportement du modèle vis-à-vis des tests.

6.4. Méthodes de vérification

Nous survolerons ici les différentes catégories de méthodes de vérification en insistant sur les principes plus que sur les aspects techniques. Ceux-ci seront détaillés dans les trois chapitres suivants à chaque fois dans un cadre précis. La principale dichotomie entre les méthodes consiste à choisir de développer une représentation de tout ou partie du comportement du modèle ou à analyser exclusivement la structure du modèle pour en déduire ses propriétés. Chacune des approches présente des avantages et des inconvénenients respectifs que nous mettrons ici en lumière.

6.4.1. *Vérification automatique ou semi-automatique ?*

L'un des objectifs des modèles formels est la vérification assistée par ordinateur. A première vue, parmi ce type de vérification, la vérification entièrement automatique semble la plus intéressante [CLA 00]. Cependant il existe certaines limitations insurmontables à la vérification automatique que le modélisateur doit connaître :

– de nombreuses propriétés sont indécidables ;

– même pour les propriétés décidables, la complexité de la vérification peut en interdire son application ;

– la vérification automatique analyse le modèle sans prendre compte les spécificités du système modélisé.

Un autre avantage de la vérification semi-automatique est qu'elle augmente la compréhension du concepteur sur son système [BER 04]. Par contre, elle est sujette à erreurs et requiert souvent une technicité et une habileté trop importante de la part du « vérificateur humain ». Soulignons tout de même la dualité entre vérification automatique et semi-automatique. Par exemple, la réduction (ou l'abstraction) de modèles se fait de manière automatique alors que le raffinement de modèles requiert la participation du concepteur. Pourtant, ce sont deux aspects d'une même theorie.

Une méthode automatique peut vérifier des propriétés spécifiées par le modélisateur ou générer certaines propriétés satisfaites par le modèle. Chaque méthode a ses inconvénients. Etablir une propriété est parfois problématique. Ainsi la preuve par induction consiste à démontrer que la propriété est vraie initialement et qu'elle le reste sous l'effet de n'importe quelle action du système. Ce type de vérification s'automatise facilement. Malheureusement, une propriété peut être vraie sans être inductive, alors qu'une propriété plus forte aurait été inductive. A l'inverse, la génération automatique de propriétés est limitée par nature : par exemple un invariant non linéaire d'un réseau de Petri ne peut être obtenu par le calcul de flots.

6.4.2. *Les méthodes structurelles*

Nous allons illustrer sur le modèle des réseaux de Petri la différence entre méthodes structurelles et comportementales. Voici quatre méthodes dont les deux premières sont structurelles car elles ne nécessitent pas d'appliquer la règle de franchissement pour obtenir des résultats.

– Un réseau de Petri net est un graphe et le flot des jetons doit suivre les arcs de ce graphe ; ainsi l'analyse de ce graphe permet par exemple de découvrir des blocages.

– Un réseau de Petri net est une transformation linéaire du marquage vu comme un vecteur ; l'analyse de cette transformation conduit au calcul d'invariants linéaires.

– Un réseau de Petri engendre une structure d'évènements dotée d'une relation de causalité et d'une relation de conflit. Les méthodes d'ordre partiel tirent profit de cette structure afin de réduire la complexité de la construction du graphe d'accessibilité.

– Les méthodes dite de symétrie tirent profit du fait qu'il n'est pas nécessaire de connaître l'identité des couleurs mais simplement leur « état » dans le marquage courant pour en déduire le comportement futur d'un réseau coloré.

Détaillons tout d'abord les méthodes structurelles.

Analyse du graphe d'un réseau de Petri. L'examen de ce graphe conduit à deux familles complémentaires de méthodes soit locales soit globales. La structure locale de certains sous-réseaux rend possible des analyses indépendamment du reste du réseau. C'est le point clef de la théorie des réductions où par exemple l'agglomération de transitions consiste à transformer une séquence (non atomique) de transitions en une transition (atomique) [BER 87, HAD 06]. Même si une telle réduction n'élimine qu'une seule transition, son impact est considérable. En effet, au niveau du graphe d'accessibilité, cette réduction supprime tous les états intermédiaires entre le début d'une séquence non atomique et sa fin. Autrement dit, une agglomération divise approximativement par un facteur 2, l'espace d'accessibilité et par conséquent n agglomérations ont un facteur de réduction d'approximativement 2^n.

L'analyse de la structure globale du graphe fournit dans des sous-classes de réseaux (e.g. réseaux à choix libre [DES 95]) des algorithmes efficaces pour les propriétés génériques (e.g. vivacité, caractère borné). Appliqués à la classe générale des réseaux de Petri, des algorithmes similaires fournissent soit des conditions nécessaires soit des conditions suffisantes pour les propriétés génériques.

Techniques d'algèbre linéaire. Les techniques d'algèbre lineaire reposent sur l'équation de changement d'état qui exprime le fait qu'un marquage accessible est obtenu par la somme du marquage initial et le produit de la matrice d'incidence par le vecteur d'occurrences de la séquence de franchissement. Aussi une pondération du marquage qui annule la matrice d'incidence (appelée un flot) est laissée invariante par le franchissement de n'importe quelle séquence de franchissement. De manière similaire un vecteur d'occurrences de transition qui annule la matrice d'incidence (appelé un rythme) laisse le marquage invariant.

Par conséquent, on peut fixer deux objectifs aux techniques d'algèbre linéaire : le calcul d'une famille génératrice de flots (resp. de rythmes) puis l'utilisation des flots (resp. des rythmes) à l'analyse du réseau. La calcul de flots est plus ou moins facile selon les contraintes sur les coefficients des flots. Par exemple, la complexité du calcul des flots à coefficients quelconques est polynomiale tandis que malheureusement le clacul de flots à coefficients positifs ne l'est pas [KRU 87]. En effet, les flots à coefficients positifs sont souvent plus utiles que les flots quelconques et les

chercheurs ont développé de nombreuses heuristiques afin de diminuer la complexité expérimentale de ce calcul [COL 91]. Les algorithmes sont maintenant bien établis. Les applications des flots et des rythmes sont multiples : ils généralisent les conditions d'application des réductions, ils caractérisent un sur-ensemble de l'ensemble d'accessibilité, ils fournissent des bornes sur les degrés de franchissement maximum, on peut en déduire des distances *synchroniques* entre transitions, etc.

6.4.3. *Les méthodes comportementales*

Les méthodes comportementales sont certainement les plus appliquées avec les techniques de simulation. Aussi avant de décrire certaines d'entre elles, il est important d'expliquer les deux principales façons de traiter l'explosion combinatoire inhérente à ces méthodes : la gestion de l'espace lors de la construction du graphe d'accessibilité et la construction d'un graphe équivalent mais plus petit.

Une gestion efficace de la construction du graphe présente un avantage significatif. Elle est largement indépendante du modèle formel qui engendre le graphe et s'applique ainsi aux réseaux de Petri, aux algèbres de processus, etc. Nous exposons les principes de deux techniques fréquemment utilisées : les diagrammes de décision binaires et la vérification à la volée.

Les diagrammes de décisions binaires (BDD). Ces techniques ont été proposées afin de réduire la représentation d'expressions booléennes [AKE 78]. Une expression booléenne est représentée par un graphe enraciné acyclique où les nœuds non terminaux, étiquetés par les variables de l'expression ont deux successeurs (correspondant aux valeurs possibles de la variable) et où les nœuds terminaux sont au nombre de deux (*vrai* et *faux*). Afin d'évaluer l'expression pour une valeur donnée des variables, on parcourt le graphe de la racine à un nœud terminal en choisissant le successeur d'un nœud selon la valeur de la variable étiquetant le nœud. Comme les sous-expressions apparaissant plus d'une fois dans l'expression sont factorisées, le gain peut être très important.

L'application de cette technique à la réduction de graphes d'état repose sur l'identification d'un état à un vecteur de bits. Un chemin du BDD est donc équivalent à un état et le BDD dénote le sous-ensemble des états dont le chemin associé se termine par le nœud *vrai*. Les opérations sur les BDD s'interprètent alors comme des opérations ensemblistes. L'une de ces opérations permet de calculer le sous-ensemble d'états successeurs d'un état du sous-ensemble courant. Ainsi le graphe d'état est construit par saturation. Cette technique est étendue aussi à la vérification de formules de logique temporelle (très efficacement dans le cas de CTL). Dans le papier fondateur [BUR 90], les expérimentations exhibent des encodages de graphes dont le nombre d'états atteint 10^{20}. Ces performances sont maintenant largement dépassées. Un inconvénient

de cette méthode est l'imprédictabilité du facteur de compression même de manière grossière.

Vérification à la volée. La vérification à la volée est basée sur deux idées simples : les propriétés atomiques s'évaluent localement sur chaque état et il n'y a pas de chemin infini où tous les états sont différents dans un graphe fini. Aussi au lieu de construire le graphe complet, on développe uniquement les chemins élémentaires du graphe. La mémoire requise pour ce parcours est donc proportionnelle au plus long chemin élémentaire [HOL 87]. Dans le cas le plus défavorable, il n'y a pas de gain mais en moyenne celui-ci est important.

Ici aussi les techniques sont étendues afin de vérifier les formules de logique temporelle [JAR 89]. L'astuce consiste à développer dynamiquement les chemins du produit synchronisé du graphe d'état avec un automate (e.g. un automate de Büchi pour les formules de LTL) [COU 90a]. Ce qui est particulièrement intéressant avec cette méthode est son adaptation à l'espace mémoire de la machine. En effet, on peut ajouter un cache d'états mémorisant des états déjà rencontrés mais hors du chemin courant, arrêtant ainsi de manière précoce l'exploration d'un chemin si un état du cache est rencontré. Un autre aspect fructueux de cette méthode est qu'elle se combine facilement avec d'autres techniques de réduction (par exemple, les méthodes d'ordre partiel discutées ci-dessous).

Les méthodes d'ordre partiel. Les méthodes d'ordre partiel exploitent des critères structurels afin de réduire le graphe. Elles s'implémentent de manière efficace pour les réseaux de Petri. Deux de ces méthodes - *sleep set* et *stubborn set* - associent un ensemble de transitions à un état construit afin de restreindre les développements futurs du graphe. Ces ensembles de transitions sont déterminés à partir d'une relation d'indépendance entre transitions (éventuellement calculée à partir du marquage courant). Deux transitions sont indépendantes si leur franchissement ne sont pas mutuellement exclusifs. La relation structurelle revient à tester si les transitions ne partagent pas de place en entrée tandis la relation comportementale teste si la somme pondérée de leur pré-conditions n'excède pas le marquage courant.

La méthode dite des *sleep sets* mémorise dans le marquage courant certaines transitions indépendantes de celles franchies pour atteindre ce marquage [GOD 90]. La construction garantit que franchir une transition du *sleep set* conduit à un marquage déjà construit. Aussi cette technique supprime des arcs du graphe mais laisse inchangé le nombre d'états.

Etant donné un marquage, un *stubborn set* de transitions est tel que toute séquence composée par d'autres transitions n'inclut que des transitions indépendantes de cet ensemble [VAL 89]. Aussi restreindre le franchissement aux transitions de cet ensemble préserve la possibilité de compléter ce franchissement par une séquence ignorée. La construction du graphe réduit est similaire à la construction ordinaire excepté que :

– lorsqu'un état est examiné, l'algorithme calcule un *stubborn set* incluant au moins une transition franchissable si le marquage n'est pas mort ;

– les successeurs du marquage courant sont ceux atteints par les transitions franchissables de cet ensemble.

Une conséquence immédiate de la nature de ces ensembles est que l'existence de marquages morts se vérifie sur le graphe réduit. Cette méthode requiert plus de calculs que la méthode des *sleep set* puisque ces ensembles ne sont pas déterminés incrémentalement et qu'ils incluent des transitions non franchissables. En revanche, le facteur de réduction est généralement plus important car certains états sont supprimés. De plus, combiner les deux méthodes ne présente pas de difficultés majeures [GOD 91]. L'inconvénient de cette méthode est qu'elle doit être modifiée pour prendre en compte la propriété à vérifier. La vérification des propriétés de sûreté n'entraine que des modifiations accessoires de l'algorithme tandis que la vérification des propriétés de vivacité nécessite d'inclure des tests relatifs aux circuits du graphe [VAL 93].

Une autre méthode d'ordre partiel est basée sur les dépliages de réseaux de Petri. Un dépliage de réseau est un réseau de Petri acyclique où les places représentent les jetons des marquages du réseau original et les transitions représentent les franchissements du réseau original.

La construction débute avec les places correspondant au marquage initial et développe les transitions associées aux franchissements des transitions initialement franchissables en connectant les places entrées (i.e. les jetons consommés) à une nouvelle transition et en créant et connectant les places de sortie (i.e. les jetons produits). Puis on itère le processus. Evidemment si le réseau contient une séquence infinie alors le dépliage est infini et par conséquent ce dépliage doit être stoppé. Afin de produire des dépliages finis (mais complets du point de vue de l'accessibilité), différentes méthodes de troncatures ont été proposées [MCM 92, ESP 96]. Le dépliage est une représentation très compacte de l'ensemble d'accessibilité et par conséquent les propriétés de sûreté se vérifient en utilisant bien moins d'espace mémoire (la complexité en temps peut être aussi réduite mais pas de façon significative). La méthode a par ailleurs été étendue afin de supporter la vérification de formules de logique temporelle [COU 96]. Son principe consiste à construire un graphe de dépliages où les transitions à observer pour évaluer la formule sont exclusivement des transitions du graphe.

Analyse de modèles de haut niveau.

Lorsqu'on souhaite développer des méthodes de vérification pour un modèle de haut niveau dont la sémantique s'exprime à l'aide d'un modèle de bas niveau, deux directions sont possibles. Soit on adapte les méthodes applicables au modèle de bas niveau en prenant en compte que ce modèle a été engendré par un modèle de haut niveau. Soit on développe des méthodes spécifiques au modèle de haut niveau. Nous illustrons maintenant notre propos avec le modèle des réseaux de Petri colorés.

L'analyse de la coloration d'un réseau de Petri a de nombreuses applications théoriques. Nous nous limitons ici à présenter trois de ces développements. Avant tout notons, qu'une méthode de vérification s'applique soit aux réseaux colorés soit aux réseaux colorés *paramétrés* si la méthode ne requiert pas de fixer la taille des domaines de couleur ou les intègrent sous forme de variables.

La théorie des réductions pour réseaux colorés repose sur les principes suivants [HAD 89] :

1) Choisir une réduction pour réseaux de Petri.

2) Ajouter des conditions *colorées* aux conditions structurelles (i.e. des conditions relatives aux fonctions de couleur étiquetant les arcs) ; ces conditions colorées doivent être si possibles les conditions minimales définissables au niveau du réseau coloré afin de garantir des conditions structurelles d'un ensemble de réductions du réseau déplié.

3) Vérifier qu'il y a un ordre possible d'application de ces réductions dans le réseau déplié.

4) Définir la transformation par une transformation structurelle semblable à la réduction originelle complétée par une transformation des domaines et des fonctions ; cette transformation doit correspondre aux réductions successives du réseau déplié.

La paramétrisation de la méthode est plus ou moins facile et s'obtient en substituant aux conditions colorées des conditions syntaxiques relatives aux expressions dénotant les fonctions de couleur [EVA 05].

Le calcul de flots pour réseaux colorés nécessite une analyse plus approfondie de la structure des fonctions de couleur. Il s'avère que le fondement de ce calcul est le concept algébrique des inverses généralisés. De plus, un algorithme élégant adapté de l'élimination de Gauss a été conçu avec un calcul itératif d'inverses generalisés [COU 90b]. La complexité en temps et en espace est réduite de manière impressionnante et les flots obtenus sont représentés d'une façon compacte qui se prête à une interprétation intuitive.

Malheureusement, la paramétrisation de cette méthode n'est pas possible. Aussi les chercheurs se sont tournés dans une autre direction : identifier les expressions des fonctions de couleur à des polynômes. Il s'agit alors d'appliquer une élimination gaussienne dans un anneau de polynômes. La principale difficulté réside dans la transformation (et la transformation réciproque) d'une fonction de couleur en un polynôme. Certaines sous-classes de réseaux de Petri bien formés[1] ont été étudiées avec succès (réseaux réguliers, réseaux ordonnés [HAD 88]) à l'aide de cette technique. Une

1. Les réseaux de Petri symétriques étaient auparavant appelés *réseaux bien formés*. Le nom de ce type de réseaux a changé récemment, à l'occasion du processus de normalisation des réseaux de Petri (norme ISO/IEC 15909).

autre façon d'obtenir des méthodes *paramétrées* consiste à relacher l'exigence que la famille de flots obtenue soit une famille génératrice. Ainsi on obtient des méthodes élémentaires s'appliquant à des classes de réseaux très générales et fournissant une information significative (bien qu'incomplète).

La construction du graphe symbolique d'accessibilité des réseaux de Petri symétriques [CHI 97] exploite la symétrie des fonctions de couleur : permuter les couleurs d'un marquage est équivalent à appliquer cette permutation à tous les comportements futurs du réseau. Cette symétrie conduit à une relation d'équivalence entre marquages et entre franchissements de transition. Une fois qu'une représentation canonique d'une classe d'équivalence de marquages (et de franchissements de transition) est définie, la construction du graphe symbolique est semblable à la construction ordinaire. Ici encore, les méthodes ont été étendues pour vérifier les formules de logique temporelle. De plus, les plus récentes d'entre elles sont capables de traiter le cas de symétries partielles du système et/ou de la formule [HAD 00].

6.5. Plan de la partie II

Le chapitre 7 présente les techniques structurelles applicables aux réseaux de Petri (et aux réseaux de Petri de haut niveau). Il débute par les réductions de réseaux où l'analyse locale du graphe « réseau de Petri » permet d'effectuer des transformations de sous-réseaux. Puis le calcul d'invariants linéaires est détaillé en insistant sur la spécificité des algorithmes selon la nature de la famille de flots recherchée. Ce chapitre illustre ensuite comment l'analyse globale du graphe permet d'élaborer des algorithmes efficaces de vérification des propriétés génériques pour des sous-classes de réseaux de Petri. Il s'achève par une présentation des réductions et du calcul de flots pour les réseaux de Petri de haut niveau.

Le chapitre 8 est consacré à l'analyse des systèmes à états finis en s'appuyant sur les réseaux de Petri. Trois familles de méthodes sont décrites : les méthodes d'ordre partiel pour les réseaux de Petri, la construction de graphes d'accessibilité basé sur les diagrammes de décision et la construction de graphes quotients pour les réseaux de Petri de haut niveau à l'aide de l'exploitation des symétries.

Le chapitre 9 étudie l'intérêt des systèmes à états infinis et les moyens de les analyser. Deux formalismes illustrent les deux directions possibles pour ce type de recherche. Les réseaux de Petri récursifs sont définis en enrichissant les réseaux de Petri afin de modéliser des patrons de systèmes à évènements discrets tout en conservant la décidabilité du plus de propriétés possibles. Le π-calcul, quant à lui, est une algèbre de processus très expressive où les méthodes de vérification s'obtiennent en se restreignant à des sous-classes de ce modèle.

6.6. Bibliographie

[ABD 01] ABDULLA P. A., NYLÉN A., « Timed Petri Nets and BQOs. », *ICATPN*, p. 53-70, 2001.

[AKE 78] AKERS S., « Binary decision diagrams », *IEEE transactions on Computers*, vol. C-27(6), p. 509-516, June 1978.

[BER 87] BERTHELOT G., « Transformations and Decompositions of Nets », BRAUER W., REISIG W., ROZENBERG G., Eds., *Advances in Petri Nets '86 - Part I*, vol. 254 de *LNCS*, p. 359-376, Springer Verlag, Bad Honnef, Allemagne, Février 1987.

[BER 91] BERTHOMIEU B., DIAZ M., « Modeling and Verification of Time Dependent Systems Using Time Petri Nets. », *IEEE Trans. Software Eng.*, vol. 17, n°3, p. 259-273, 1991.

[BER 04] BERTOT Y., CASTÉRAN P., *Interactive Theorem Proving and Program Development Coq'Art : The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag, 2004.

[BÉR 05] BÉRARD B., CASSEZ F., HADDAD S., LIME D., ROUX O. H., « When Are Timed Automata Weakly Timed Bisimilar to Time Petri Nets ? », *FSTTCS*, p. 273-284, 2005.

[BUR 90] BURCH J., CLARK E., MCMILLAN K., DILL D., L.J. H., « Symbolic Model Checking : 10^{20} states and beyond », *Proceedings of the 5^{th} IEEE Symposium on Logic in Computer Science*, 1990.

[CHE 95] CHENG A., ESPARZA J., PALSBERG J., « Complexity results for 1-safe nets », *Theoretical Computer Science*, vol. 147, p. 117–136, 1995.

[CHI 97] CHIOLA G., DUTHEILLET C., FRANCESCHINIS G., HADDAD S., « A Symbolic Reachability Graph for Coloured Petri Nets. », *Theor. Comput. Sci.*, vol. 176, n°1-2, p. 39-65, 1997.

[CLA 86] CLARKE E. M., EMERSON E. A., SISTLA A. P., « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. », *ACM Trans. Program. Lang. Syst.*, vol. 8, n°2, p. 244-263, 1986.

[CLA 00] CLARKE E. M., GRUMBERG O., PELED D. A., *Model Checking*, MIT Press, 2000.

[COL 91] COLOM J. M., SILVA M., « Convex Geometry and Semiflows in P/T Nets. A Comparative Study of Algorithms for Computation of Minimal P-Semiflows. », *Lecture Notes in Computer Science ; Advances in Petri Nets 1990*, vol. 483, p. 79-112, Springer-Verlag, 1991.

[COM 71] COMMONER F., HOLT A. W., EVEN S., PNUELI A., « Marked Directed Graphs. », *J. Comput. Syst. Sci.*, vol. 5, n°5, p. 511-523, 1971.

[COU 90a] COURCOURBETIS C., VARDI M., WOLPER P., YANNAKAKIS M., « Memory efficient Algorithms for the Verification of temporal Properties », *Proceedings of Computer Aided Verification 90*, vol. 30 de *DIMACS*, North-Holland, 1990.

[COU 90b] COUVREUR J. M., « The General Computations of Flows for Coloured Nets. », *Proceedings of the 11th International Conference on Application and Theory of Petri Nets, 1990, Paris, France*, p. 204–223, 1990.

[COU 96] COUVREUR J., POITRENAUD D., « Model Checking Based on Occurence Net Graph », GOTZHEIN R., BREDEREKE J., Eds., *Formal Description Techniques IX, Theory Applications and Tools, FORTE/PSTV'96*, vol. 663 de *LNCS*, Kaiserslautern, Germany, Chapman-Hall, p. 380-395, October 1996.

[DES 95] DESEL J., ESPARZA J., « Free Choice Petri Nets », vol. 40, Cambridge University Press, 1995.

[DUF 98] DUFOURD C., FINKEL A., SCHNOEBELEN P., « Reset nets between decidability and undecidability », *(ICALP'98) L.N.C.S*, vol. 1443, p. 103-115, Juillet 1998.

[EME 86] EMERSON E. A., HALPERN J. Y., « "Sometimes" and "Not Never" revisited : on branching versus linear time temporal logic. », *J. ACM*, vol. 33, n°1, p. 151-178, 1986.

[ESP 96] ESPARZA J., ROMER S., W. V., « An Improvment of McMillan's Unfolding Algorithm », *Proceedings of the second International Workshop TACAS'96*, vol. 1055 de *LNCS*, Passau, Germany, Springer-Verlag, p. 87-106, March 1996.

[ESP 98] ESPARZA J., « Reachability in Live and Safe Free-choice Petri Nets is NP-complete », *Theoretical Computer Science*, vol. 198, n°1–2, p. 211–224, 1998.

[EVA 05] EVANGELISTA S., HADDAD S., PRADAT-PEYRE J.-F., « Syntactical Colored Petri Nets Reductions. », *ATVA*, p. 202-216, 2005.

[FRU 00] DE FRUTOS-ESCRIG D., RUIZ V. V., ALONSO O. M., « Decidability of Properties of Timed-Arc Petri Nets. », *ICATPN*, p. 187-206, 2000.

[GOD 90] GODEFROID P., « Using Partial Orders to Improve Automatic Verification Methods », *Proceedings of Computer Aided Verification 90*, vol. 30 de *DIMACS*, North-Holland, p. 321-340, 1990.

[GOD 91] GODEFROID P., WOLPER P., « Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties », *Proceedings of Computer Aided Verification 91*, vol. 575 de *LNCS*, Springer-Verlag, July 1991.

[HAD 88] HADDAD S., COUVREUR J. M., « Validation of Parallel Systems with Coloured Petri Nets. », COSNARD, M. et al., Eds., *Parallel Processing. Proceedings of the IFIP WG 10.3 Working Conference, 1988, Pisa, Italy*, Amsterdam, The Netherlands, North-Holland, p. 377–390, 1988.

[HAD 89] HADDAD S., « A Reduction Theory for Coloured Nets », *Advances in Petri Nets 1989*, vol. LNCS 424, p. 209–235, 1989.

[HAD 99] HADDAD S., POITRENAUD D., « Theoretical Aspects of Recursive Petri Nets », *Proc. $20^t h$ Int. Conf. on Applications and Theory of Petri nets*, vol. 1639 de *Lecture Notes in Computer Science*, Williamsburg, VA, USA, Springer Verlag, p. 228-247, Juin 1999.

[HAD 00] HADDAD S., ILIÉ J.-M., AJAMI K., « A Model Checking Method for Partially Symmetric Systems. », *FORTE*, p. 121-136, 2000.

[HAD 06] HADDAD S., PRADAT-PEYRE J., « New Efficient Petri Nets Reductions for Parallel Programs Verification », *Parallel Processing Letters*, vol. 16, n°1, p. 101-116, World Scientific, 2006.

[HOL 87] HOLZMANN G., « Automated Protocol Validation in argos : Assertion proving and scatter searching », *IEEE transactions on Software Engineering*, vol. 13(6), p. 683-696, June 1987.

[JAR 89] JARD C., JERON T., « On-line Model Checking for finite linear temporal logic specifications », *Proceedings of Automatic Verification Methods for Finite State Systems*, vol. 407 de *LNCS*, Springer-Verlag, p. 189-196, 1989.

[JEN 97] JENSEN K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*, Monographs in Theoretical Computer Science, Springer-Verlag, 1997.

[KAR 69] KARP R., MILLER R., « Parallel Program Schemata », *Journal of Computer and System Sciences*, vol. 3, n°2, p. 147-195, 1969.

[KOS 82] KOSARAJU S., « Decidability of Reachability in Vector Addition Systems », *Proc. 14^{th} ACM Symp. Theory of Computing (STOC'82)*, San Franciscp, CA, p. 267-281, Mai 1982.

[KOU 94] KOUTNY M., ESPARZA J., BEST E., « Operational Semantics for the Petri Box Calculus. », *CONCUR*, p. 210-225, 1994.

[KRU 87] KRUCKEBERG C., JAXY M., « Mathematical Methods for Calculating Invariants in Petri Nets », GOOS G., HARTMANIS J., Eds., *Advances in Petri Nets 1987*, vol. 266 de *LNCS*, Springer-Verlag, p. 104-131, 1987.

[MAY 82] MAYR E., MEYER A., « The complexity of the Word Problems for Commutative Semigroups and Polynomial Ideals », *Advance in Mathematics*, vol. 46, p. 305-329, 1982.

[MAY 84] MAYR E., « An Algorithm for the General Petri Net Reachability Problem », *SIAM Journal of Computing*, vol. 13, p. 441-460, 1984.

[MCM 92] MCMILLAN K., « On-the-fly verification with stubborn sets », *Proceedings of Computer Aided Verification 93*, vol. 663 de *LNCS*, Montreal, Canada, Springer-Verlag, p. 164-175, June 1992.

[MIL 95] MILNER R., *Communication and concurrency*, Prentice Hall International (UK) Ltd., 1995.

[PAP 94] PAPADIMITRIOU C., *Computational Complexity*, Addison-Wesley, Reading, Mass., 1994.

[REI 95] REINHARDT K., Reachability in Petri Nets with inhibitor arcs, Manuscript non publié. accessible via http ://www-fs.informatik.uni-tuebingen.de/ reinhard/, 1995.

[SIS 85] SISTLA A. P., CLARKE E. M., « The Complexity of Propositional Linear Temporal Logics », *J. ACM*, vol. 32, n°3, p. 733-749, 1985.

[VAL 89] VALMARI A., « Stubborn Sets for Reduced Space Generation », *Proc. 10^{th} Intern. Conference on Application and Theory of Petri Nets*, Bonn, Germany, June 1989.

[VAL 93] VALMARI A., « On-the-fly verification with stubborn sets », *Proceedings of Computer Aided Verification 93*, vol. 697 de *LNCS*, Springer-Verlag, p. 397-408, 1993.

Chapitre 7

Approches structurelles de vérification

Dans ce chapitre, nous développons les principales techniques structurelles applicables aux réseaux de Petri. Les réductions de réseaux consistent à transformer le réseau en un réseau plus petit mais équivalent vis-à-vis de propriétés standard en s'appuyant sur des structures particulières de sous-réseaux. Le calcul d'invariants linéaires repose sur l'algèbre linéaire et permet de déduire des propriétés comportementales du réseau. Enfin l'analyse des siphons et des trappes d'un réseau fournit des conditions suffisantes ou nécessaires pour certaines propriétés standard telles que la vivacité. Pour des sous-classes de réseaux, cette analyse conduit à des algorithmes efficaces d'évaluation de ces propriétés.

7.1. Introduction

Les méthodes d'analyse structurelle cherchent à faire face au problème de l'explosion de l'espace des états inhérent aux techniques d'analyse fondées sur le graphe des marquages accessibles (ou systèmes de transition associés).

La théorie associée, dite structurelle, consiste à étudier les relations entre les propriétés comportementales d'un modèle d'une part et ses caractéristiques structurelles (exprimées en termes d'algèbre linéaire ou de théorie des graphes) et de son marquage initial d'autre part. Il s'agit d'une approche naturelle en raison de la description intentionnelle (et non extensionnelle) du formalisme des réseaux de Petri. Elle conduit à une compréhension approfondie du système étudié.

Chapitre rédigé par Kamel BARKAOUI et Jean-François PRADAT-PEYRE.

Ces techniques permettent, en raisonnant sur la structure du modèle, de vérifier de manière paramétrée certaines propriétés comportementales. Par ailleurs, elles aident à la conception d'opérateurs de synthèse (composition ou raffinement) capables de préserver des propriétés recherchées.

Nous détaillerons dans ce chapitre, les deux principales familles de techniques d'analyse structurelle :

– les techniques d'analyse basées sur l'équation d'état (algèbre linéaire)

– les techniques d'analyse basées sur la relation de flot (théorie des graphes)

Dans le cas général, seules des conditions nécessaires ou suffisantes sont établies à l'aide de ces deux types de techniques complémentaires. On aboutit à des conditions nécessaires et suffisantes dès lors que certaines restrictions sur les schémas de synchronisation sont assurées.

Nous préciserons plus loin ces restrictions et nous présenterons également les classes de réseaux les plus larges possibles pour lesquelles certaines propriétés comportementales deviennent vérifiables et de manière paramétrée à l'aide de ces techniques.

Mais auparavant, nous commencerons par détailler les conditions structurelles des règles de réduction de base. Chacune d'elles permet de substituer un sous réseau par un autre plus simple (*i.e.* de taille réduite) tout en préservant les propriétés initiales du modèle.

7.2. Réductions structurelles de réseaux de Petri

Soient $\mathcal{M}$ un modèle et π une propriété. Il est possible dans certains cas de construire un modèle $\mathcal{M}'$ tel que vérifier π sur $\mathcal{M}$ est équivalent (en termes de validité) à vérifier π sur $\mathcal{M}'$ mais est moins coûteux (en incluant le coût de construction de $\mathcal{M}'$ à partir de $\mathcal{M}$).

Cette méthodologie est connue dans le cadre des réseaux de Petri sous le nom de réduction. Une réduction est une transformation de réseau qui réduit la taille du réseau tout en préservant un ensemble de propriétés. Trois points caractérisent une réduction :

– les conditions d'applications qui définissent les cas où la réduction est applicable ;

– la transformation de réseau qui définit le réseau réduit à partir du réseau initial ;

– les propriétés préservées ; i.e. les propriétés pour lesquelles il est équivalent d'opérer sur le réseau initial ou sur le réseau réduit.

La définition d'une réduction repose alors sur un compromis entre des exigences contradictoires :

– les conditions d'applications doivent être simples à vérifier ; il est en particulier important de ne pas avoir à construire le graphe des marquages accessibles pour tester ces conditions. Les comportements doivent donc être capturés par des conditions structurelles ou algébriques ; de plus, la réduction doit correspondre à des cas de modélisation fréquemment employés ;

– la transformation de réseau doit être utile dans le sens où elle doit réduire de façon importante le nombre de places, le nombre de transitions ou le nombre de marquages accessibles. Ceci induit implicitement une transformation éventuelle des comportements ;

– le nombre de propriétés préservées doit être élevé ; ceci implique que le réseau ne doit pas être trop profondément modifié ou que l'on ait une idée *a priori* sur le comportement du réseau.

Les travaux initiaux sur les réductions dans les réseaux de Petri sont dus à G. Berthelot qui définit dans sa thèse d'état dix réductions [BER 80, BER 83, BER 86]. Parmi ces réductions, trois sont particulièrement utiles : les pré- et post-agglomérations de transitions (qui consistent à fusionner des transitions franchies en séquence) et la suppression de place implicite (qui consiste à supprimer une place inutile dans le modèle). L'intérêt des agglomérations de transitions est de permettre de réduire considérablement la taille de l'ensemble des marquages accessibles en supprimant un état intermédiaire (l'application d'une seule agglomération peut réduire la taille de cet ensemble par deux). L'intérêt de la suppression de la place implicite est nul quant à la réduction de la taille de l'ensemble d'accessibilité mais très importante dans un processus de réduction puisqu'elle permet l'application d'autres réductions par la simplification du réseau de Petri.

Le schéma général des agglomérations de transitions consiste à isoler deux ensembles de transitions H et F tels que le franchissement d'une transition f de F soit toujours précédé du franchissement d'une transition h de H. La définition de ces deux réductions repose sur des conditions d'applications purement structurelles (donc simples à vérifier).

7.2.1. *Pré-agglomération de transitions*

Pour la pré-agglomération on considère une transition h et un ensemble de transitions F (n'incluant pas h).

Les conditions structurelles suivantes garantissent qu'il est équivalent de considérer qu'il y a un état intermédiaire entre le franchissement de h et d'une transition de

F ou de considérer que le franchissement des séquence $h.f$ est effectué de façon atomique. Le terme équivalent signifie ici équivalent pour une propriété de base (vivacité, caractère borné, ...) [BER 83], [BER 86] ou pour toute formule LTL n'obervant pas la transition h [POI 00].

DÉFINITION 7.1 (Transitions ordinaires pré-agglomérables) *Soit (R, m_0) un réseau de Petri. Un ensemble de transitions F est pré-agglomérable avec une transition h n'appartenant pas à F si et seulement si les conditions suivantes sont vérifiées :*

1) Il existe une place p modélisant un état intermédiaire entre le franchissement de h et d'une transition de F :

a) $m_0(p) = 0$ *;*

b) ${}^\bullet p = \{h\}$ *et* $p^\bullet = F$ *;*

c) $Post(p, h) = 1$ *et* $\forall f \in F, Pre(p, f) = 1$.

2) h ne produit des marques que dans p : $h^\bullet = \{p\}$ *;*

3) h n'est en conflit avec aucune autre transition : $\forall q \in {}^\bullet h$, $q^\bullet = \{h\}$ *;*

4) h a au moins une pré-condition : ${}^\bullet h \neq \emptyset$.

La première hypothèse garantit que toute occurrence de franchissement d'une transition f de F est précédée du franchissement de h. La seconde assure que h n'est utile que pour le franchissement d'une transition de F. La troisième implique que si h est franchissable alors elle le reste jusqu'à ce qu'elle soit franchie (h peut donc être retardée). Enfin la quatrième hypothèse permet d'assurer l'équivalence pour le caractère borné entre le réseau réduit et l'original.

DÉFINITION 7.2 (Réseau pré-aggloméré) *Une pré-agglomération sur le réseau (R, m_0) produit le réseau (R_r, m_{0r}) défini par :*

1) Places et transitions du réseau réduit :

- $P_r = P \setminus \{p\}, T_r = T \setminus \{h\}$ *;*
- $\forall q \in P_r, m_{0r}(q) = m_0(q)$.

2) Pré et Post conditions inchangées : $\forall t \in T_r$,

- $\forall q \in P_r, Post_r(q, t) = Post(q, t)$
- $\forall q \in P_r \setminus {}^\bullet h$, $Pré_r(q, t) = Pré(q, t)$

3) Nouvelles pré-conditions : $\forall f \in F$,

- $\forall, q \in {}^\bullet h$, $Pré_r(q, f) = Pré(q, h)$

7.2.2. *Post-agglomération de transitions*

Pour cette réduction, on considère deux ensembles de transitions H et F (d'intersection vide). Les conditions d'application de la post-agglomération assurent que

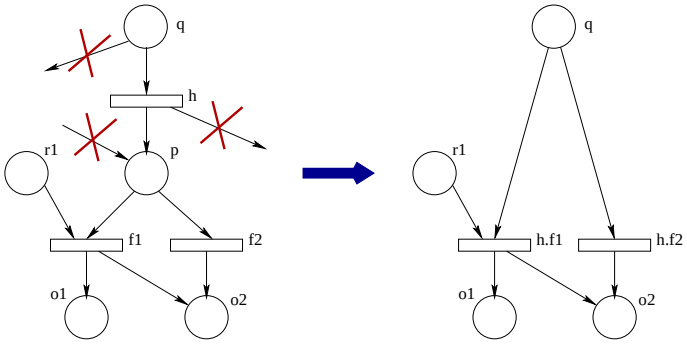


Figure 7.1. *Illustrations des contraintes et de l'application de la réduction*

toute transition f de F est immédiatement franchissable après le franchissement d'une transition h de H. Ceci permet de garantir que pour tout couple (h, f), $h \in H$, $f \in F$, il est équivalent (en termes de vivacité, caractère borné ou pour la véracité d'une formule LTL n'observant pas la transition f) de franchir les transitions h et f en séquence ou de façon atomique.

DÉFINITION 7.3 (Transitions post-agglomérables) *Soit (R, m_0) un réseau de Petri. Un ensemble de transitions F est post-agglomérable avec un ensemble de transitions H disjoint de F ($H \cap F = \emptyset$) si et seulement si les conditions suivantes sont vérifiées :*

1) Il existe une place p modélisant un état intermédiaire entre le franchissement d'une transition de H et d'une transition de F :

a) $m_0(p) = 0$ *;*

b) ${}^\bullet p = H$ *et* $p^\bullet = F$ *;*

c) $\forall h \in H, Post(p, h) = 1$ *;*

d) $\forall f \in F$ *et* $Pre(p, f) = 1$ [1]*.*

2) Les transitions de F n'ont pas d'autre pré-condition que p : ${}^\bullet F = \{p\}$ *;*

3) Il existe une transition f de F ayant une post-condition : $F^\bullet \neq \emptyset$*.*

Comme pour la pré-agglomération, la première hypothèse garantit un séquencement entre le franchissement des transitions de H et de celles de F. La seconde hypothèse (point clef de la post-agglomération) assure que toute transition f de F est franchissable dès que p est marquée (on peut donc "avancer" le franchissement des transitions f). La dernière assure l'équivalence pour le caractère borné entre le réseau réduit et le réseau original.

1. La version originale de Berthelot autorise des valuations hétérogènes. Cependant, ce cas se rencontre peu en pratique et complique inutilement la définition de cette réduction.

DÉFINITION 7.4 (Réseau post-aggloméré) *Une post-agglomération sur le réseau* (R, m_0) *produit le réseau réduit* (R_r, m_{0_r}) *défini par :*

1) Places et transitions du réseau réduit :
$P_r = P \setminus \{p\}$ *et* $T_r = T \cup (H \times F) \setminus (H \cup F)$ *;*
on note hf *la transition* (h, f) *de* $H \times F$ *;*

2) Partie du réseau inchangée : $\forall t \in T_r \setminus (H \times F), \forall q \in P_r$,
- $Pré_r(q,t) = Pré(q,t)$ *et* $Post_r(q,t) = Post(q,t)$ *;*
- $m_{0_r}(q) = m_0(q)$

3) Les nouvelles transitions : $\forall hf \in (H \times F), \forall q \in P_r$,
- $Pré_r(q, hf) = Pré(q, h)$ *;*
- $Post_r(q, hf) = Post(q, h) + Post(q, f)$.

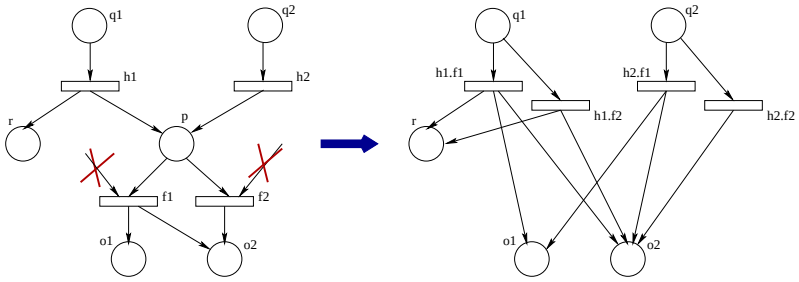


Figure 7.2. *Illustrations des contraintes et de l'application de la réduction*

L'exemple suivant (Figure 7.3) modélise le fonctionnement d'une piscine. Les clients qui arrivent prennent un droit d'entrée dans le hall de la piscine (transition `t0`). Le nombre de clients dans le hall est limité à K (place `y`). Ensuite, un client réserve une cabine pour se changer (transition `t1`), le nombre de cabines est limité à $K1$ (place `r1`). Il prend ensuite un panier pour déposer ses affaires et libère une place dans le hall de la piscine (transition `t2`) ; le nombre de paniers est limité à $K2$ (place `r2`). Une fois déshabillé il libère la cabine et va se baigner (transition `t3`). Puis, pour pouvoir se rhabiller il doit obtenir une cabine (transition `t4`). Une fois rhabillé, il libère le panier (transition `t4`), puis libère la cabine et sort de la piscine (transition `t5`). Selon les valeurs de K, $K1$ et $K2$ ce réseau présente ou non un interblocage.

Avant d'analyser (dans une prochaine sous-section) ce modèle du point de vue de la présence ou non d'interblocage , il est possible de le réduire considérablement. En effet une pré-agglomération de `t0` avec `t1` est possible. On obtient alors le modèle de gauche de la figure 7.4. Il est alors possible d'appliquer une post-agglomération de la transition `t5` avec la transition `t6`, puis, une post-agglomération de `t2` avec `t3`. On obtient alors le modèle de la figure 7.5 qui est équivalent au modèle initial pour la propriété analysée (ici l'absence d'interblocage).

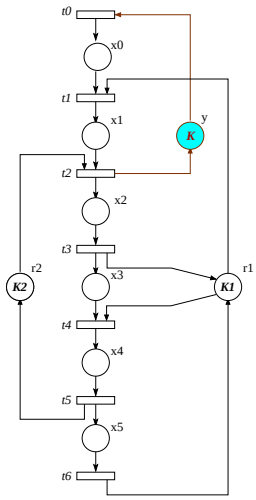


Figure 7.3. *L'exemple de la piscine*

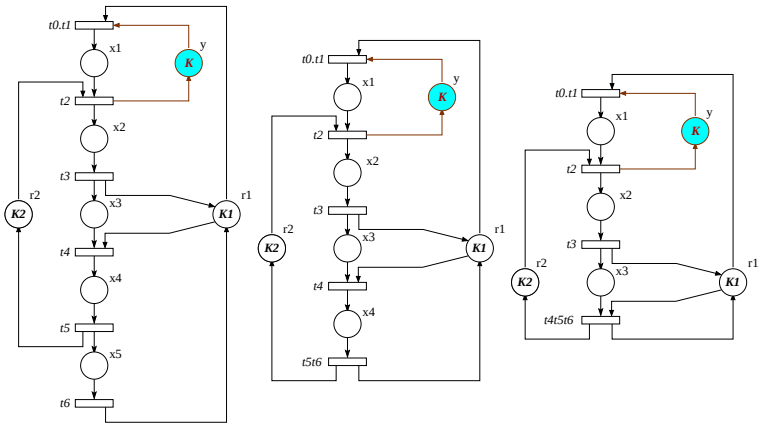


Figure 7.4. *Suite de réductions de l'exemple de la piscine*

Pour une valeur de $K = 14$, et de $K1 = K2 = 15$, le modèle initial a un graphe d'accessibilité comprenant 449601 nœuds (marquages accessibles) et 2465327 arcs. Le graphe du modèle réduit ne comprend plus que 240 nœuds et 659 arcs. On le voit, la réduction est importante. Il faut également noter que les réductions effectuées sont indépendantes du marquage initial (analyse paramétrée possible) et qu'elles sont tout à fait compatibles avec d'autres techniques de réductions opérées lors de la construction du graphe d'accessibilité (telles que l'utilisation des ensembles persistants, des ensembles dormants ou des dépliages). Cependant, dans le cas général, les réductions

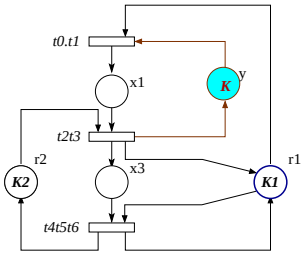


Figure 7.5. *L'exemple de la piscine réduit par une suite d'agglomérations*

ne permettent pas de montrer directement une propriété (sauf dans le cas où le modèle est réduit à une transition). Nous allons étudier des techniques structurelles qui le permettent, mais auparavant nous étudions un des piliers des techniques structurelles, le calcul d'invariants.

7.3. Calcul et application des invariants linéaires

Le calcul des invariants linéaires exploite l'équation de changement d'état pour exhiber des vecteurs indexés par les places (resp. les transitions), appelés flots ou flots positifs (resp. rythmes ou semi-rythmes) selon qu'il s'agit de vecteur d'entiers relatifs ou naturels.

Un flot (ou flot positif) caractérise l'invariance du nombre global de marques contenues dans un sous-ensemble de places moyennant la pondération induite par le flot et donne une information pertinente sur l'évolution possible des marquages. Comme de plus, les flots se calculent indépendamment du marquage initial les invariants déduits des flots permettent de démontrer de nombreuses propriétés qui soit sont indépendantes du marquage initial (comme le caractère borné) soit laissent le marquage initial comme un paramètre de cette preuve (par exemple l'exclusion mutuelle de deux places, la vivacité du réseau, etc.). De façon plus générale, un ensemble de flots définit un sur-ensemble de l'espace d'accessibilité qui est utilisé pour réduire l'ensemble des marquages à considérer dans certaines preuves.

Un rythme traduira de façon orthogonale une invariance sur le nombre d'occurrence des transitions dans toute séquence dont le marquage initial et le marquage final sont les mêmes. Ces rythmes servent à énoncer des conditions nécessaires comme par exemple sur le caractère borné et vivant d'un réseau.

7.3.1. *Algorithmes de calculs d'invariants linéaires*

Dans cette section nous considérons un réseau de Petri $N = (R, M_0)$ où W denote la matrice d'incidence de ce réseau.

DÉFINITION 7.5 (flots) *Un vecteur $f \in \mathbb{Q}^P$ (resp. $(\mathbb{Q}^+)^P$) est un flot (resp. flot positif) ssi ${}^t f.W = \overrightarrow{0}$. On notera un flot f par la somme formelle $f = \sum_{p \in P} f(p).\overrightarrow{p}$ (et on omettra de faire apparaître une place dont la pondération est nulle).*

Par exemple, le vecteur $f_1 = \overrightarrow{x1} + \overrightarrow{y}$ désigne le flot du modèle de droite de la figure 7.5 qui associe la pondération 1 aux places y et $x1$ et 0 aux autres. On vérifiera sans difficulté que ${}^t f_1.W = \overrightarrow{0}$ où W est la matrice d'incidence de ce modèle.

$$
{}^t f = \begin{array}{c} \begin{array}{ccccc} y & x1 & x3 & r1 & r2 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}, \quad W = \begin{array}{c} \\ y \\ x1 \\ x3 \\ r1 \\ r2 \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \end{array}
$$

La proposition suivante justifie l'intérêt des flots (ou flots positifs) quant à la caractérisation (partielle) des marquages accessibles.

PROPOSITION 7.6 *Si f est un flot ou un flot positif du réseau alors*

$$\forall M \in Acc(R, M_0), \sum_{p \in P} f(p).M(p) = cst = \sum_{p \in P} f(p).M_0(p)$$

Par exemple, le flot $f_1 = \overrightarrow{x1} + \overrightarrow{y}$ donnera l'invariant $\forall M \in Acc(R, M_0), m(x1) + m(y) = K$.

Calculer les flots d'un réseau revient à résoudre un système linéaire à solution dans $\mathbb{Q}$ ou $\mathbb{Q}^+$. Calculer une famille génératrice de flots se fait à l'aide de l'algorithme d'élimination de Gauss.

Avant d'énoncer cet algorithme nous définissons tout d'abord les notions de famille génératrice et de plus petite famille génératrice.

DÉFINITION 7.7 (famille génératrice) *Soit A une matrice où P est l'ensemble des indices de lignes et T l'ensemble des indices de colonnes. Une famille génératrice des solutions dans $\mathbb{Q}$ (resp. $\mathbb{Q}^+$) de l'équation ${}^t X.A = 0$ est une famille $\{V_1, \ldots, V_m\}$ de vecteurs non nuls de $\mathbb{Q}^P$ (resp. $(\mathbb{Q}^+)^P$) telle que :*

– $\forall\ V_i,\ {}^t V_i.A = 0$

– $\forall\ V\ \in\ \mathbb{Q}^P$ *(resp.* $(\mathbb{Q}^+)^P$*) avec* ${}^tV.A = 0$*,*
$\exists \lambda_1, \ldots, \lambda_m \in Q\ (resp.\ \mathbb{Q}^+)$ *tels que* $V = \lambda_1.V_1 + \ldots + \lambda_m.V_m$

La famille est une plus petite famille génératrice si la privation d'un de ces éléments la rend non génératrice.

Le principe de l'algorithme de Gauss consiste à obtenir la famille génératrice pour le système réduit aux i premières équations à partir de celui réduit aux $i-1$ premières équations en conservant les vecteurs qui vérifient l'équation (i) puis en choisissant un pivot parmi les les vecteurs pour lequel le résultat de l'équation (i) est non nul et en le combinant avec tous les autres vecteurs dans ce cas. L'ensemble initial des solutions est la base canonique de $\mathbb{Q}^P$.

```
1  Algorithme Gauss
2  début
3      S :={p⃗ | p ∈ P}
4      %%S est la base canonique de E = Q+
5      pour tous les t ∈ T faire
6          %%On cherche un pivot pour l'équation t
7          S0 :={s ∈ S | ts.At = 0}
8          S1 :={s ∈ S | ts.At ≠ 0}
9          %%On fait les combinaisons linéaires si nécessaire
10         si S1 ≠ ∅ alors
11             Soit s_pivot ∈ S1
12             pour tous les s0 ∈ S1 \ s_pivot faire
13                 S :=S ⋃ {(ts_pivot.At).s − (ts.At).s_pivot}
14     %%S est une plus petite famille génératrice de tX.W = 0, X ∈ Q^P
15 fin
```

Algorithme 1 : Calcul d'un famille génératrice minimale de flots.

Cet algorithme, de complexité polynomiale, s'implémente simplement. Il peut également s'utiliser pour calculer les flots manuellement en indiquant sur les lignes les solutions partielles. Voyons son déroulement pour le calcul des flots du modèle de figure 7.5. On démarre l'algorithme avec la matrice d'incidence et la base canonique :

$$W = \begin{array}{c} \\ \overrightarrow{y} \\ \overrightarrow{x1} \\ \overrightarrow{x3} \\ \overrightarrow{r1} \\ \overrightarrow{r2} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \end{array}$$

Choisissons $t2.t3$ comme équation à « annuler », prenons $\overrightarrow{y}$ comme pivot et combinons le à toutes les autres lignes. On obtient alors le système :

$$W_1 = \begin{array}{c} \\ \overrightarrow{y} - \overrightarrow{x1} \\ \overrightarrow{y} - \overrightarrow{x3} \\ \overrightarrow{y} - \overrightarrow{r1} \\ \overrightarrow{y} + \overrightarrow{r2} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \end{array}$$

On constate que la transition $t2.t3$ est bien « annulée ». Annulons alors la transition $t0.t1$ et prenons $\overrightarrow{y} - \overrightarrow{x3}$ comme pivot. On obtient alors le système :

$$W_2 = \begin{array}{c} \\ \overrightarrow{y} + \overrightarrow{x1} \\ \overrightarrow{y} - \overrightarrow{r1} \\ -\overrightarrow{x3} - \overrightarrow{r2} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

La matrice est nulle. La base de flots est donc $\{\overrightarrow{y} + \overrightarrow{x1}, \overrightarrow{y} - \overrightarrow{r1}, \overrightarrow{x3} + \overrightarrow{r2}\}$.

Si l'on cherche une famille génératrice de flots **positifs** (souvent plus commode à utiliser) il faut appliquer un autre algorithme (issu d'un lemme dû à Farkas). Le principe de cet algorithme consiste à obtenir la famille génératrice pour le système réduit aux i premières équations à partir de celui réduit aux $i-1$ premières équations en conservant les vecteurs qui vérifient l'équation (i) et en combinant deux à deux les vecteurs pour lequel le résultat de l'équation (i) est positif avec ceux pour lequel le résultat est négatif.

L'inconvénient bien connu de cette méthode est que le nombre d'inconnues peut augmenter considérablement. On essaie donc dans la mesure du possible d'éliminer les solutions non nécessaires et l'on parle alors de méthode minimale [COL 91].

La notion de support d'un vecteur permet de caractériser simplement à la fois le côté générateur et le côté minimal d'une plus petite famille génératrice. Les deux caractérisations ainsi obtenues sont à la base des méthodes minimales : on rejette les solutions non nécessaires.

DÉFINITION 7.8 (support d'un vecteur) *Soit V un vecteur indicé par un ensemble Y, le support de V noté Supp(V), ou bien encore $\|V\|$, est le sous-ensemble de Y défini par :*

Supp(V) = { $y \in Y \mid V_y \neq 0$ }.

Caractérisation 7.3.1 (dite de génération [COL 91]) *Soit F une famille de vecteurs non nuls de $(\mathbb{Q}^+)^P$ solutions de l'équation ${}^tX.A = 0$, $F = \{V_1, \ldots, V_m\}$. Alors F est une famille génératrice si et seulement si :*

$\forall\ V \in (\mathbb{Q}^+)^P$ *avec* ${}^tV.A = 0$ *et* $V \neq 0, \exists\, V_i$ *avec* $Supp(V_i) \subseteq Supp(V)$ *(on dit dans ce cas que V_i est de support plus petit ou égal à V).*

On peut chercher à minimiser la taille d'une famille génératrice. La caractérisation qui suit répond à ce problème.

Caractérisation 7.3.1 (dite de minimalité [COL 91]) *Une famille génératrice F des solutions dans $\mathbb{Q}^+$ de l'équation $A.X = 0$ est une plus petite famille génératrice si et seulement si :*

$\forall\ V \neq V' \in F\ Supp(V') \not\subseteq Supp(V)$

Nous présentons maintenant une version de l'algorithme de Farkas sous une forme minimale : on ne calcule que les solutions de support minimal, les autres, non nécessaires, étant éliminées.

De nombreuses optimisations de cet algorithme peuvent être apportées afin de diminuer sa complexité théorique - cet algorithme est en effet exponentiel en espace et en temps -. Les deux plus importantes reposent sur les idées suivantes :

– On ne choisit pas au hasard l'équation à annuler.

– Afin d'éliminer les solutions de support non minimal, on ne compare pas les solutions obtenues à l'étape (i) entre elles mais uniquement avec les solutions obtenues à l'étape précédente. Ceci a un double avantage : on effectue un test unidirectionnel au lieu d'un test bidirectionnel, et surtout, on peut montrer qu'il est possible d'éliminer plus rapidement des solutions qui sont minimales à l'étape (i) mais qui seront minimalisées par la suite.

L'étude de différentes optimisations, dont celles que nous venons de mentionner, a été effectuée de façon très complète dans [COL 91] ; nous invitons le lecteur a s'y référer. On pourra également trouver en annexe de cette thèse quelques détails pratiques sur l'implémentation de cet algorithme.

1 Algorithme **Farkas**
2 **début**
3 S :=$\{\overrightarrow{p} \mid p \in P\}$
4 **%%**S est la base canonique de $E = (\mathbb{Q}^+)^P$
5 **pour tous les** $t \in T$ **faire**
6 **%%**On sépare les vecteurs de S en fonction de leur signe par rapport à l'équation t
7 S^0 :=$\{s \in S \mid {}^t s.A_t = 0\}$
8 S^+ :=$\{s \in S \mid {}^t s.A_t > 0\}$
9 S^- :=$\{s \in S \mid {}^t s.A_t < 0\}$
10 **%%**On fait les combinaisons linéaires
11 **pour tous les** $(s^+, s^-) \in S^+ \times S^-$ **faire**
12 S :=$S \bigcup \{({}^t s^+.A_t).s^- - ({}^t s^-.A_t).s^+\}$
13 **%%**On élimine les solutions de supports non minimal
14 **pour tous les** $(s, s') \in S \times S$, $s \neq s'$ **faire**
15 **si** $Supp(s') \subseteq Supp(s)$ **alors**
16 S :=$S \setminus \{s\}$
17 **%%**S est une plus petite famille génératrice de ${}^t X.W = 0, X \in (\mathbb{Q}^+)^P$
18 **fin**

Algorithme 2 : Calcul d'un famille génératrice minimale de flots positifs.

Voyons le déroulement de cet algorithme sur le modèle de la figure 7.5. Le point de départ est le même que celui de l'algorithme de Gauss.

$$W = \begin{array}{c} \\ \overrightarrow{y} \\ \overrightarrow{x1} \\ \overrightarrow{x3} \\ \overrightarrow{r1} \\ \overrightarrow{r2} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} -1 & 1 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \\ -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \end{array}$$

On annule la transition `t2.t3.t4` (ce n'est pas le choix le plus judicieux mais cela permet de faire apparaître des solutions de support non minimal et d'illustrer ainsi un point de l'algorithme). Pour cette transition, l'ensemble S^0 est vide, l'ensemble $S^+ = \left\{\overrightarrow{y}, \overrightarrow{x3}, \overrightarrow{r1}\right\}$ et l'ensemble $S^- = \left\{\overrightarrow{x1}, \overrightarrow{r2}\right\}$. La combinaison de ces vecteurs

conduit à la matrice suivante :

$$W_1 = \begin{array}{c} \\ \overrightarrow{y} + \overrightarrow{x1} \\ \overrightarrow{y} + \overrightarrow{r2} \\ \overrightarrow{x3} + \overrightarrow{x1} \\ \overrightarrow{x3} + \overrightarrow{r2} \\ \overrightarrow{r1} + \overrightarrow{x1} \\ \overrightarrow{r1} + \overrightarrow{r2} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} \end{array}$$

On annule alors `t0.t1` en combinant $\overrightarrow{x3} + \overrightarrow{x1}$ avec $\overrightarrow{y} + \overrightarrow{r2}$ et $\overrightarrow{r1} + \overrightarrow{r2}$ et on obtient la matrice

$$W_2 = \begin{array}{c} \\ \overrightarrow{y} + \overrightarrow{x1} \\ \overrightarrow{x3} + \overrightarrow{r2} \\ \overrightarrow{r1} + \overrightarrow{x1} \\ \overrightarrow{y} + \overrightarrow{r2} + \overrightarrow{x3} + \overrightarrow{x1} \\ \overrightarrow{r1} + \overrightarrow{r2} + \overrightarrow{x3} + \overrightarrow{x1} \end{array} \begin{array}{c} \begin{array}{ccc} t0.t1 & t2.t3 & t4.t5.t6 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

La matrice est nulle, donc la famille calculée est une famille génératrice. Cepdendant, les deux dernières solutions ne sont pas de support minimal (par exemple $Supp(\overrightarrow{y} + \overrightarrow{x1}) \subseteq Supp(\overrightarrow{y} + \overrightarrow{r2} + \overrightarrow{x3} + \overrightarrow{x1})$). Elles peuvent donc être supprimées sans enlever le caractère générateur de la famille calculée. Une plus petite famille génératrice est donc la famille $\{\overrightarrow{y} + \overrightarrow{x1}, \overrightarrow{x3} + \overrightarrow{r2}\overrightarrow{r1} + \overrightarrow{x1}\}$.

Ces flots positifs fournissent les invariants linéaires suivants : $\forall m \in Acc(R, m_0)$ $m(y) + m(x1) = K$, $m(x3) + m(r2) = K2$ et $m(r1) + m(x1) = K1$. Ce qui s'interprète ainsi : le nombre de clients dans le hall d'attente est au plus de $min(K, K1)$ (premier et troisième flot) et, le nombre de baigneurs (place $x3$) est au plus de $K2$ (second flot).

7.3.2. *Calculer d'autres invariants*

Les invariants linéaires peuvent être utilisés pour caractériser partiellement l'espace d'accessibilité dans des systèmes de contraintes (programmes linéaires) caractérisant, par exemple, la franchissabilité (ou la non franchissabilité) d'une (ou de plusieurs) transition pour un marquage particulier, ou encore l'exclusion mutuelle entre plusieurs places. La construction de ces programmes linéaires se fait de la façon suivante : on associe à chaque place p une variable entière x_p qui compte le nombre de marques présentes dans la place. On traduit alors la propriété recherchée (franchissabilité, non franchissabilité de transition, exclusion mutuelle entre places) par des inéquations liant les variables x_p aux pré- et post-conditions des transitions étudiées ou aux variables associées aux places dont on analyse le marquage. On ajoute

ensuite les contraintes caractérisant le marquage considéré. Enfin, afin de restreindre les valeurs possibles pour les variables x_p, on augmente le système de contraintes de positivité sur ces variables et de contraintes induites par les flots du modèle. Le programme linéaire est alors résolu en nombres entiers ou en nombre rationnels (ce qui diminue légèrement la qualité du test dans certains cas mais le rend très peu coûteux).

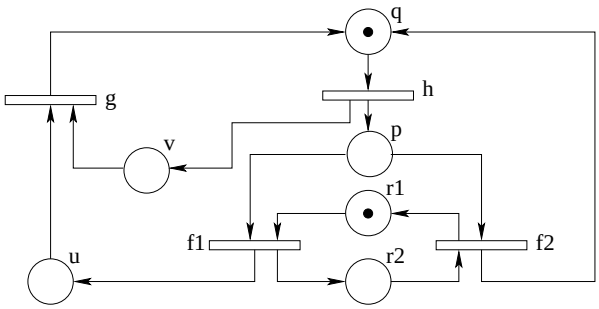


Figure 7.6. *Un réseau de Petri*

Prenons l'exemple de la figure qui suit. Un calcul de flots positifs sur ce modèle donne les deux flots :

$$\overrightarrow{p} + \overrightarrow{q} + \overrightarrow{u} \text{ et } \overrightarrow{r1} + \overrightarrow{r2}$$

Ces flots conduisent aux invariants linéaires suivants :

– $\forall m \in Acc(R, m_0), m(p) + m(q) + m(u) = 1$, ce qui caractérise un processus avec 3 états p, q et u ;

– $\forall m \in Acc(R, m_0), m(r1) + m(r2) = 1$ qui caractérise une alternance dans la présence de 2 ressources.

Supposons que l'on souhaite démontrer que :

– lorsque le processus est dans l'état p (i.e. p est marquée) il peut continuer son exécution immédiatement (soit $f1$ soit $f2$ est franchissable).

– lorsque le processus est dans l'état p, la transition g n'est pas franchissable.

Pour la première propriété nous construisons un programme linéaire dans lequel nous associons à chaque place p une variable x_p qui désigne le nombre de marques contenues dans p. Cet ensemble des variables va définir un marquage potentiellement accessible. Les contraintes du problème seront définies par les invariants, l'hypothèse que p est marquée et la négation de la conclusion recherchée (ni $f1$ ni $f2$ ne sont franchissables). Nous concluons que la propriété est vérifiée si le problème n'est pas satisfaisable (attention, il ne s'agit que d'une condition suffisante).

$$\left[\begin{array}{ll}
\forall i \in P, x_i \geq 0 & \text{Un marquage est positif} \\
\left.\begin{array}{l} x_q + x_p + x_u = 1 \\ x_{r1} + x_{r2} = 1 \end{array}\right\} & \text{Les contraintes associées aux invariants sont satisfaites} \\
x_p \geq 1 & \text{La place } p \text{ est marquée} \\
\left.\begin{array}{l} x_{r1} = 0 \\ x_{r2} = 0 \end{array}\right\} & \text{Ni la transition } f1 \text{ ni la transition } f2 \text{ ne sont franchissables}
\end{array}\right.$$

La seconde propriété s'exprime de manière similaire. Observons que la négation de la conclusion se traduit par des bornes inférieures relatives au marquage des places.

$$\left[\begin{array}{ll}
\forall i \in P, x_i \geq 0 & \text{Un marquage est positif} \\
\left.\begin{array}{l} x_q + x_p + x_u = 1 \\ x_{r1} + x_{r2} = 1 \end{array}\right\} & \text{Les contraintes associées aux invariants sont satisfaites} \\
x_p \geq 1 & \text{La place } p \text{ est marquée} \\
\left.\begin{array}{l} x_v \geq 1 \\ x_u \geq 1 \end{array}\right\} & \text{La transition } g \text{ est franchissable}
\end{array}\right.$$

7.4. Vérification structurelle basée sur la relation de flot

La vivacité d'un réseau est une propriété comportementale difficile à établir mais d'un intérêt fondamental pour la preuve de bon fonctionnement du système modélisé. D'un point de vue structurel, la vivacité peut être définie comme étant l'existence d'un marquage initial pour lequel le réseau soit vivant. Cette section est consacrée à l'apport essentiel de l'analyse structurelle au problème de la vivacité. Nous détaillerons les principales propriétés structurelles qui conditionnent ou qui assurent la vivacité.

Ceci aboutira à la reconnaissance de classes de réseaux de Petri où la vivacité est décidable structurellement et efficacement.

La structure d'un *réseau de Petri* $\langle N, M_0 \rangle$ est totalement décrite par son graphe connexe biparti, orienté et valué $N = \langle P, T, F, V \rangle$ (P étant l'ensemble des places, T l'ensemble des transitions, F sa relation de flot : $F \subseteq (P \times T) \cup (T \times P)$ et $V : F \rightarrow N^+$ la fonction de valuation). M_0 est son marquage initial. $M_0(p)$ désigne le nombre de jetons contenus initialement dans la place p. Soit x un sommet du graphe N, nous désignons par ${}^\bullet x$ l'ensemble des prédecesseurs de x et par $x^\bullet$, l'ensemble de ses successeurs. Cette notation s'étend naturellement aux sous-ensembles de nœuds. Nous introduisons maintenant la notion de sous-réseau que nous emploierons à plusieurs reprises dans la suite.

DÉFINITION 7.9 *Le sous-réseau induit par le sous-ensemble de places $P' \subseteq P$ est le réseau $N' = \langle P', T', F', V' \rangle$ défini comme suit :*

$T' = {}^\bullet P' \cup P'^\bullet$

$F' = F \cap ((P' \times T') \cup (T' \times P'))$

V' est la restriction de V sur F'.

On définit de manière analogue le sous-réseau induit par le sous-ensemble de transitions $T' \subseteq T$.

DÉFINITION 7.10 *Une place p est dite marquée dans un marquage M si $M(p) \geq Min_{t \in p^\bullet}\{V(t,p)\}$.*
Un sous-ensemble de places est dit marqué dans M ssi il contient une place marquée.
Une place p est dite à valuation homogène ssi $\forall t, t' \in p^\bullet, V(p,t) = V(p,t') = V(p)$.
Une place p est dite à valuation non bloquante ssi $p^\bullet \neq \emptyset \Rightarrow Min_{t \in {}^\bullet p}\{V(t,p)\} \geq Min_{t \in p^\bullet}\{V(t,p)\}$
Un réseau est à valuation homogène si toutes ses places sont à valuation homogène.

Pour des raisons de simplicité, les réseaux considérés par la suite sont à valuation homogène.

Nous introduisons les concepts et définitions de base sous-jacents à l'analyse structurelle basée sur la relation de flot. Nous commençons d'abord par rappeler les définitions des propriétés comportementales que nous cherchons à vérifier de manière structurelle.

7.4.1. *Propriétés comportementales de base*

Soit $\langle N, M_0 \rangle$ un *réseau de Petri*. Une transition t appartenant à T est *morte* pour un marquage M accessible de M_0 ssi il n'existe pas de marquage M^* accessible de M pour lequel t soit franchissable.

Un marquage M accessible de M_0 est dit *bloquant* (ou état de blocage) ssi toutes les transitions sont mortes pour ce marquage.

Un réseau $\langle N, M_0 \rangle$ est *pseudo-vivant* (non bloquant) ssi il existe une transition franchissable pour tout marquage accessible (il n'existe pas d'état de blocage).

Une transition t appartenant à T est *vivante* pour M_0 ssi pour tout marquage M accessible de M_0, il existe un marquage M' accessible de M pour lequel t soit franchissable (t non vivante ssi il existe un marquage accessible M' pour lequel t est morte).

Un réseau $\langle N, M_0 \rangle$ est dit *vivant* ssi toutes ses transitions sont vivantes pour M_0.

Une place p appartenant à P est dite *bornée* pour M_0 ssi il existe k appartenant à $\mathbb{N}$ tel que pour tout marquage accessible $M : M(p) \leq k$.

$\langle N, M_0 \rangle$ est dit *borné* si toutes ses places sont bornées pour M_0.

$\langle N, M_0 \rangle$ est dit *bien formé* ssi il est vivant et borné.

N est *structurellement vivant* ssi il existe un marquage initial M_0 tel que $\langle N, M_0 \rangle$ soit vivant.

N est *structurellement borné* ssi $\langle N, M_0 \rangle$ est borné pour tout marquage initial M_0.

DÉFINITION 7.11 *Un marquage accessible $M^* \in R(M_0)$ est un marquage stable ssi toute transition $t \in T$ est soit vivante, soit morte pour M^*. On obtient ainsi une partition de $T = (T_D, T_L)$ (ou T_D désigne l'ensemble des transitions mortes pour M^* et T_L désigne l'ensemble des transitions vivantes pour M^*).*

PROPOSITION 7.12 *Soit $\langle N, M_0 \rangle$ un réseau de Petri pseudo-vivant mais non vivant, il existe alors un marquage stable M^* pour lequel $T_D \neq \emptyset$ et $T_L \neq \emptyset$.*

7.4.2. *Propriétés structurelles de base*

Soit f un vecteur, on désignera par $||f||^+$ (resp. $||f||^-$) l'ensemble des places correspondant aux composantes positives (resp. négatives) du support de f : $||f||^+ = \{p \in P \mid f(p) > 0\}$, $||f||^- = \{p \in P \mid f(p) < 0\}$. On a $Supp(f) = ||f||^+ \cup ||f||^-$.

Un réseau N est dit conservatif ssi il existe un flot positif f tel que $Supp(f) = P$. De manière immédiate, si N est conservatif alors N est structurellement borné.

Etant donné le principe de localité associé au franchissement des transitions qui régit l'évolution du réseau, il est naturel de se focaliser sur la topologie du voisinage d'une transition décrivant l'interaction entre le conflit et la synchronisation portant sur les conditions qui lui sont associées.

On définit ainsi la notion de place racine d'une transition t donnée, qui permet d'établir une relation de causalité de non vivacité sur les transitions du voisinage de t [BAR 05].

DÉFINITION 7.13 (place racine) *Soit r une place de P. r est dite racine de la transition t ssi $\forall p \in {}^\bullet t, r^\bullet \subseteq p^\bullet$.*
On désigne par $Root(t)$ l'ensemble des places racines de t.

PROPOSITION 7.14 *Soit $N = \langle P, T, F, V \rangle$ un graphe de Petri $r \in P$ et $t \in r^\bullet$. Si r est une place racine de t alors $\forall t' \in r^\bullet$, ${}^\bullet t \subseteq {}^\bullet t'$.*

PROPOSITION 7.15 *Soit p une place de P : ${}^\bullet p \subseteq T_D \Rightarrow p^\bullet \subseteq T_D$. Si les transitions en entrée d'une place sont mortes alors ses transitions en sortie sont mortes aussi.*

PROPOSITION 7.16 *Soit p une place non bornée de P : $p^\bullet \subseteq T_D \Rightarrow {}^\bullet p \subseteq T_D$. Si les transitions en sortie d'une place non bornée sont mortes alors ses transitions en entrée sont mortes aussi.*

PROPOSITION 7.17 *Soit r une place de $Root(t)$: $t \in T_D \Rightarrow r^\bullet \subseteq T_D$.*
Si une transition t est morte alors les transitions en sortie des places racine de t sont mortes aussi.

DÉFINITION 7.18 (Transition ordonnée) *[BAR 05] Soit t une transition de T. t est dite ordonnée ssi $\forall p, q \in {}^\bullet t({}^\bullet t \neq \emptyset), p^\bullet \subseteq q^\bullet$ ou $q^\bullet \subseteq p^\bullet$.*

Si t est ordonnée, alors $Root(t) \neq \emptyset$, la réciproque est fausse.

DÉFINITION 7.19 (Réseau ordonné (ou à choix asymétrique)) *Soit $R = \langle N, M_0 \rangle$ un réseau de Petri. R est appelé un réseau ordonné ssi toute transition dans R est ordonnée.*

En particulier, si $Root(t) = {}^\bullet t, \forall t$, alors R est appelé réseau de type choix libre [DES 95].

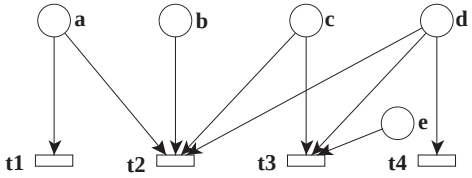


Figure 7.7. *Transitions ordonnées et non ordonnées*

Les transitions $t1, t3, t4$ de la figure 7.7 sont ordonnées mais non $t2$. Néanmoins, elles possèdent toutes une racine. $Root(t1) = \{a\}$, $Root(t2) = \{b\}$, $Root(t3) = \{e\}$, $Root(t4) = \{d\}$.

Nous donnons une condition nécessaire élémentaire pour qu'un réseau soit bien formé (vivant et borné) ce qui est en pratique souvent recherché.

PROPOSITION 7.20 *Soit $\langle N, M_0 \rangle$ un réseau de Petri. Si N est vivant et borné (bien formé) pour M_0, alors toute place (resp. transition) admet une transition (resp. place) en entrée et une transition (resp. place) en sortie.*

De même que des conditions algébriques ont été établies notamment à l'aide de la notion d'invariant pour décider du caractère borné d'un réseau, nous précisons par la suite les conditions sur la relation de flot qui sont nécessaires et/ou suffisantes pour la vivacité.

7.4.3. *La CS-propriété*

Une notion majeure de l'analyse structurelle est celle du siphon. Un siphon est un sous-ensemble de places tel que l'ensemble de ses transitions en entrée soit inclus dans l'ensemble de ses transitions en sortie.

DÉFINITION 7.21 (Siphon) *Soit $N = \langle P, T, F, V \rangle$ un graphe de Petri. S un sous-ensemble non vide de P. S est un siphon de N ssi ${}^\bullet S \subseteq S^\bullet$.*
S est dit minimal (au sens de l'inclusion) ssi il ne contient pas d'autre siphon que lui même.

Une condition nécessaire à la vivacité d'un réseau est que chacun de ses siphons minimaux doit constamment être marqué.

En effet de par sa structure un siphon devenu non marqué reste à jamais non marqué, toutes ses transitons en sortie deviennent alors mortes. Il est donc crucial de s'assurer du marquage des siphons lors de l'évolution du réseau.

PROPOSITION 7.22 *Soit S un siphon dans un réseau $\langle N, M_0 \rangle$. S'il existe un marquage M sur lequel S est non marqué alors l'ensemble des transitions en sortie de S sont mortes pour M ($S^\bullet \subseteq T_D$).*

DÉFINITION 7.23 (Siphon contrôlé) *Un siphon S d'un réseau de Petri $\langle N, M_0 \rangle$ est dit contrôlé ssi S est marqué dans tout marquage accessible i.e.*
$\forall M \in R(M_0), \exists$ une place p de S qui soit marquée dans M.

DÉFINITION 7.24 (CS-propriété) *Un réseau de Petri $\langle N, M_0 \rangle$ satisfait la CS-propriété (controlled siphon property) ssi tout siphon minimal de $\langle N, M_0 \rangle$ est contrôlé [BAR 96].*

La vérification de la CS-propriété recquiert le calcul des siphons minimaux. Les références [BAR 90, BAR 89] donnent une caractérisation efficace de la minimalité d'un siphon. Il existe différents algorithmes dans la littérature pour ce calcul. L'union de siphons est un siphon, mais pas l'intersection. Par ailleurs, un siphon non minimal n'est pas généralement décomposable en siphons minimaux [BAR 92].

Il est facile de vérifier que la CS-propriété est une condition nécessaire de vivacité et suffisante de la pseudo-vivacité.

PROPOSITION 7.25 (Condition nécessaire de vivacité) *Soit $\langle N, M_0 \rangle$ un réseau de Petri. Si $\langle N, M_0 \rangle$ est vivant alors $\langle N, M_0 \rangle$ satisfait la CS-propriété.*

PROPOSITION 7.26 (Condition suffisante de pseudo vivacité) *Soit $\langle N, M_0 \rangle$ un réseau de Petri. Si $\langle N, M_0 \rangle$ satisfait la CS-propriété alors $\langle N, M_0 \rangle$ est pseudo-vivant (absence d'état de blocage global).*

Nous présentons ici deux moyens structurels (non nécessaires) de base assurant le contrôle du marquage des siphons : le contrôle par trappe et le contrôle par invariant.

Le contrôle par trappe. La structure de trappe est duale à celle du siphon. Une trappe est un sous-ensemble de places tel que l'ensemble de ses transitions en sortie soit inclus dans l'ensemble de ses transitions en entrée. Une trappe marquée reste à jamais marquée. Le contrôle de siphon par trappe (ou contrôle interne) consiste à s'assurer de l'existence d'une trappe initialement marquée à l'intérieur même du siphon.

DÉFINITION 7.27 (Trappe) *Soit $N = \langle P, T, F, V \rangle$ un graphe de Petri. $\mathcal{T}$ un sous-ensemble non vide de P. $\mathcal{T}$ est une trappe de N ssi $\mathcal{T}^\bullet \subseteq {}^\bullet\mathcal{T}$. L'ensemble des transitions en sortie de $\mathcal{T}$ est inclus dans l'ensemble de ses transitions en entrée.*

PROPOSITION 7.28 *Soit S un siphon de $\langle N, M_0 \rangle$. Si S contient une trappe initialement marquée et dont les places sont non bloquantes alors S est contrôlé (S est dit contrôlé par trappe).*

Sous certaines conditions sur la relation de flot, ce premier type de contrôle s'avère nécessaire, car l'existence d'un siphon sans trappe fait qu'un réseau est structurellement non vivant (i.e. il n'existe pas de marquage initial pour lequel il soit vivant).

PROPOSITION 7.29 *Soit $N = \langle P, T, F, V \rangle$ un graphe de Petri Petri et A un sous-ensemble non vide de P ne contenant pas de trappe.*
Il existe alors une partition $(A_i)_{i=0...I} \subset \mathcal{P}(A)$ telle que $\forall p \in A_i, \exists t \in p^\bullet$ avec $t^\bullet \cap \cup_{k \geq i} A_k = \emptyset$.
On désigne par $Fuite(p) = \{t \in p^\bullet \mid t^\bullet \cap \cup_{k \geq i} A_k = \emptyset\}$.

PROPOSITION 7.30 *Soit $\langle N, M_0 \rangle$ un réseau de Petri. Si N contient un siphon minimal S sans trappe (S est dit strict) et tel que $\forall p \in S \; \exists t \in Fuite(p) \mid p \subseteq Root(t)$ alors S est non contrôlable (i.e. N est structurellement non vivant) [BAR 96].*

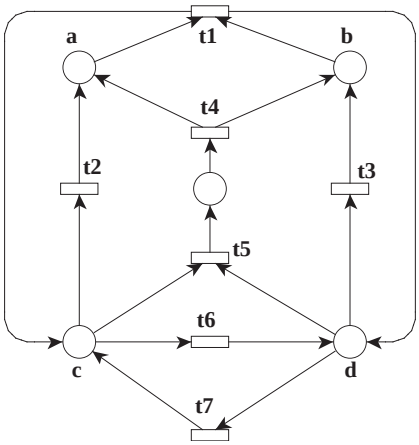


Figure 7.8. *Un réseau structurellement non vivant*

Bien que le réseau de la figure 7.8 possède de bonnes propriétés algébriques (conservatif, consistant, la conditition du théorème du rang), il est structurellement non vivant. En effet, son siphon minimal $\{b, c, d, e\}$ qui ne contient pas de trappe est non contrôlable (voir figure 7.9). De manière plus précise, on franchit $t4$ jusqu'à ce que e soit vide. Ensuite on franchit $t1$ jusqu'à ce que a ou b soit vide. Puis on vide d en franchissant $t7$ et enfin c en franchissant $t2$. Si b était vide après les franchissements de $t1$ alors le siphon est vide (et le réseau est mort). Si a était vide après les franchissements de $t1$ alors soit a, c et d sont vides et le réseau est mort. Enfin si a n'est plus vide alors l'écart entre a et b a diminué et par conséquent en itérant la procédure, on retombe nécessairement dans l'un des cas précédents.

La condition sur la relation de flot énoncée dans le théorème précédent est toujours satisfaite sur un réseau de type à choix libre (FC nets, EFC nets, Equal conflict nets [TER 96]). La CS-propriété (réduite au contrôle par trappe) est alors une condition nécessaire et suffisante de vivacité et est appelée propriété de Commoner [BAR 90, DES 95].

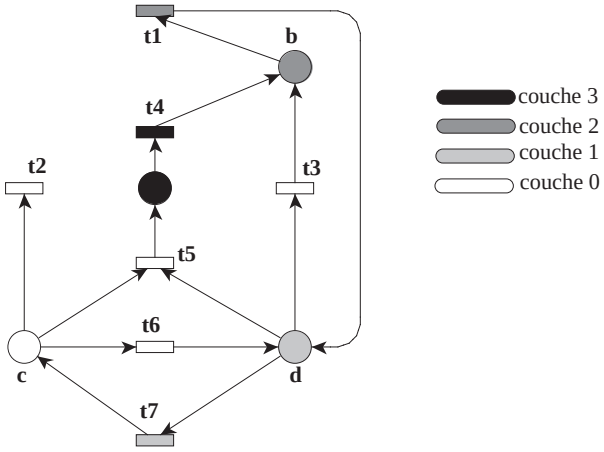


Figure 7.9. *Un siphon non contrôlable*

Par ailleurs, il faut remarquer que ce contrôle par trappe est *monotone*, c'est-à-dire qu'un siphon reste contrôlé pour toute augmentation du marquage de la trappe associée.

Dans les réseaux pour lesquels la propriété de Commoner est une condition nécessaire et suffisante, la vivacité est donc une propriété monotone.

Ceci explique pourquoi le contrôle par trappe est insuffisant pour faire face au cas général des réseaux où la vivacité est non monotone.

Le contrôle par invariant. Nous introduisons maintenant un second moyen de contrôle du marquage de siphon : le contrôle par invariant.

PROPOSITION 7.31 *Soit S un siphon de $\langle N, M_0 \rangle$. S'il existe un flot f satisfaisant les conditions suivantes*

- $||f||^+ \subseteq S$,
- $\forall p \in (||f||^- \cap S), V(p) = 1$
- $\sum_{p \in P}[f(p).M_0(p)] > \sum_{p \in S}[f(p).(V(p) - 1)]$

alors S est contrôlé par invariant.

Le mécanisme de ce contrôle [LAU 94, BAR 96] est basé sur l'existence d'un invariant dont le support positif est inclus dans le siphon. Contrairement au contrôle par trappe, il est non monotone. L'augmentation du marquage des places de l'invariant

associé ne préserve le contrôle que sous conditions. Ce type contrôle est d'autant plus réalisable que le réseau est conservatif.

Exemple :

Le modèle de la piscine (figure 7.10) possède les bonnes propriétés algébriques mais est structurellement non vivant. Son siphon minimal $S = r_1, x_2, r_2, x_4, x_5$ est non contrôlable. On peut, par ajout de la place C, dite place de contrôle (figure 7.11), introduire l'invariant $f = r_1 - C + r_2 + x_2 + 2 \cdot x_4 + x_5$ et assurer ainsi le contrôle de S pour tout marquage initial satisfaisant $K_1 - K_3 > 0$.

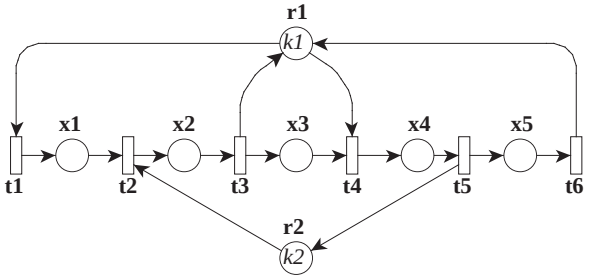


Figure 7.10. *Modèle de la piscine*

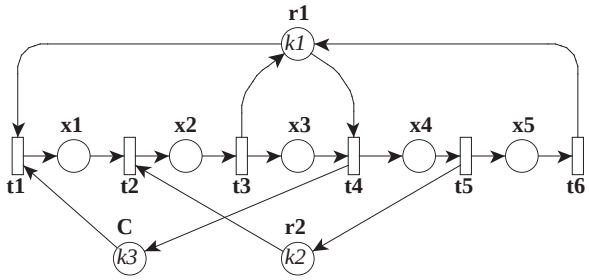


Figure 7.11. *Modèle contrôlé de la piscine*

Equivalence entre CS-propriété et vivacité. La CS-propriété étant une condition nécessaire de vivacité et une condition suffisante de pseudo-vivacité, un problème intéressant est de reconnaître des conditions structurelles qui peuvent assurer l'équivalence entre la CS-propriété et la vivacité.

On montre alors que pour les réseaux ordonnés (ou à choix asymétrique) qui incluent strictement la classe des réseaux à choix libre, cette équivalence a lieu.

Lemme 1 Soit $\langle N, M_0 \rangle$ un réseau de Petri satisfaisant la CS-propriété. Si N est non vivant alors il comporte une transition non ordonnée [BAR 96, BAR 05].

PROPOSITION 7.32 *Soit $\langle N, M_0 \rangle$ un réseau de Petri ordonné. Les deux énoncés suivants sont équivalents.*

1) $\langle N, M_0 \rangle$ satisfait la CS-propriété,

2) $\langle N, M_0 \rangle$ est vivant.

Pour un réseau ordonné (ou à choix asymétrique) borné ou non, le problème de la vivacité se ramène au problème de la vérification de la CS-propriété [BAR 96].

Exemple :

On peut vérifier que le modèle contrôlé de la piscine (figure 7.11) est un réseau ordonné. Il comporte les quatre siphons minimaux suivants :
$S1 = \{C, x_1, x_2\}; S2 = \{r_2, x_2, x_3, x_4\};$
$S3 = \{r_1, x_1, x_2, x_4, x_5\}; S4 = \{r_1, x_2, r_2, x_4, x_5\}$

Si $K_1 > 0$, $K_2 > 0$, $K_3 > 0$ et $K_1 - K_3 > 0$, la CS-propriété est vérifiée. Le réseau est donc vivant pour tout marquage initial satisfaisant ces conditions.

Il a été démontré que le problème de la vérification du caractère bien formé (vivant et borné) d'un réseau de cette classe est décidable en temps polynomial [DES 92, BAR 92]. Cela résulte du fait que pour qu'un réseau de type choix libre soit bien formé (vivant et borné), il faut et il suffit qu'il soit conservatif et que tout siphon soit aussi une trappe [DES 92, BAR 92].

7.4.4. *Caractérisation structurelle des K-systèmes*

Afin d'étendre cette équivalence entre vivacité et CS-propriété à des classes de réseaux non ordonnés, nous introduisons les K-systèmes.

Les K-systèmes [BAR 05], sont définis comme étant la classe des réseaux de Petri où il y a équivalence entre les trois propriétés suivantes : CS-propriété, pseudo-vivacité et vivacité.

DÉFINITION 7.33 (K-Système) *[BAR 05] Soit $\langle N, M_0 \rangle$ un réseau de Petri. $\langle N, M_0 \rangle$ est un K-Système ssi pour tout marquage stable M^*, $T_D(M^*) = T$ ou $T_L(M^*) = T$. Cette propriété est notée la K-propriété.*

Cette partition existe, mais n'est pas nécessairement unique.

Un K-Système est un réseau dans lequel toutes les transitions sont toutes mortes ou toutes vivantes i.e. où la pseudo-vivacité implique la vivacité.

Afin de pouvoir reconnaître structurellement l'appartenance d'un RdP à la classe des K-Systèmes, nous relâchons la condition de transition ordonnée en préservant la notion de place racine d'une transition permettant d'exploiter les relations de causalité entre transitions mortes.

Les réseaux DCS. On définit une première sous-classe des K-Systèmes appelée DCS (*Dead Closed Systems*) [BAR 05].

Soit $\langle N, M_0 \rangle$ un réseau de Petri, M^* un marquage stable qui lui est associé et t une de ses transitions, on désigne par $D(t)$ l'ensemble des transitions définies ainsi :

$$D(t) = \{t' \in T \mid t \in T_D(M^*) \Rightarrow t' \in T_D(M^*)\}$$

Si t est morte alors $D(t)$ (*Dead-closure* de t) contient toutes les transitions mortes.

Etant donnée une transition t_0, on pose $D_{sub}(t_0) = t_0$ et on l'élargit en appliquant les trois règles structurelles suivantes dérivées des propositions 7.17, 7.15 et 7.16 :

– R1 Soit p une place racine de t, $t \in D_{Sub}(t_0) \Rightarrow p^\bullet \subseteq D_{Sub}(t_0)$

– R2 Soit p une place de P, ${}^\bullet p \subseteq D_{Sub}(t_0) \Rightarrow p^\bullet \subseteq D_{Sub}(t_0)$

– R3 Soit p une place bornée de P, $p^\bullet \subseteq D_{Sub}(t_0) \Rightarrow {}^\bullet p \subseteq D_{Sub}(t_0)$

DÉFINITION 7.34 (*Dead-closed system*) *Soit $\langle N, M_0 \rangle$ un réseau de Petri. Si pour toute transition t de N, $D_{Sub}(t) = T$ alors $\langle N, M_0 \rangle$ est appelé un réseau DCS (Dead-closed system).*

L'algorithme de calcul de $D_{Sub}(t)$ est décrit ci-dessous (figure 3). Sa complexité est linéaire ($O(m)$ où m est le nombre d'arcs du graphe), équivalente à celle des algorithmes de parcours de graphes.

PROPOSITION 7.35 *Soit $\langle N, M_0 \rangle$ un DCS. $\langle N, M_0 \rangle$ est un K-Système et les trois énoncés suivants sont équivalents :*

1) $\langle N, M_0 \rangle$ est pseudo-vivant

2) $\langle N, M_0 \rangle$ satisfait la CS-propriété

3) $\langle N, M_0 \rangle$ est vivant

Considérons le réseau non ordonné (t_3) et conservatif de la figure 7.12. On peut vérifier par application de l'algorithme calculant $D(t)$ qu'il s'agit bien d'un DCS. Il comporte quatre siphons minimaux $S_1 = \{a, b, d\}$, $S_2 = \{e, c, f\}$, $S_3 = \{e, b, d\}$

1 Algorithme **CalculDsub(t)**
Données : Soient,
entrée : t une transition supposée morte
local : $D_t marked$: un ensemble de transitions
sortie : $D_{Sub}(t)$: un ensemble de transitions
2 **début**
3 $D_{Sub}(t)$:=$\{t\}$
4 $D_t marked$:=$\emptyset$
5 **tant que** $D_{Sub}(t) \setminus D_t marked \neq \emptyset$ **faire**
6 Soit $t \in D_{Sub}(t) \setminus D_t marked$
7 **si** r *est une place racine* **alors**
8 $D_{Sub}(t)$:=$D_{Sub}(t) \cup r^\bullet$ **%%**Application de R1
9 **pour tous les** $p \in t^\bullet$ **faire**
10 **si** ${}^\bullet p \subseteq D_{Sub}(t)$ **alors**
11 $D_{Sub}(t)$:=$D_{Sub}(t) \cup p^\bullet$ **%%**Application de R2
12 **pour tous les** $p \in {}^\bullet t$ *tels que* p *est bornée* **faire**
13 **si** $p^\bullet \subseteq D_{Sub}(t)$ **alors**
14 $D_{Sub}(t)$:=$D_{Sub}(t) \cup {}^\bullet p$ **%%**Application de R3
15 $D_t marked := D_t marked \cup \{t\}$
16 **fin**

Algorithme 3 : Algorithme de calcul de $D_{Sub}(t)$

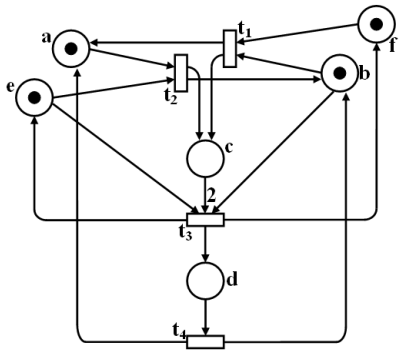


Figure 7.12. *Exemple de DCS*

et $S_4 = \{a, f, d\}$. Ce DCS satisfait la CS-propriété et est donc vivant, pour tout marquage initial vérifiant les quatre conditions suivantes :

$a+b+d>0,\ \ e+c+f>0,\ \ e+b+d-f>0,\ \ a+f+d-e>0.$

Un exemple d'un tel « bon » marquage est $M_0 = a+b+e+f$.

Les Root-Systèmes. Une caractérisation structurelle plus forte des K-Systèmes peut être obtenue à l'aide du sous-réseau induit par l'ensemble des places racines. On définit ainsi la sous-classe des DCS appelée Root-Système. Le graphe associé à un Root-Système est tel que chaque transition (non nécessairement ordonnée) admet une place racine [BAR 05].

DÉFINITION 7.36 *Soit $N = \langle P, T, F, V\rangle$ un réseau tel que $\forall t \in T, Root(t) \neq \emptyset$. La composante racine de N est le sous-réseau $N^* = \langle Root_N, T^*, F^*, V^*\rangle$ défini comme suit :*

– $Root_N$ est l'ensemble des places racines.

– $T^* = T$.

– $F^* \subseteq (F \cap ((Root_N \times T^*) \cup (T^* \times Root_N))) \mid (p,t) \in F^*$ ssi $(p,t) \in F$ et p est racine de t, et $(t,p) \in F^*$ ssi $(t,p) \in F$.

– V^* est la restriction de V sur F^*.

DÉFINITION 7.37 *Soit $\langle N, M_0\rangle$ un réseau de Petri. $\langle N, M_0\rangle$ est un Root-Système ssi :*

– $\forall t \in T, Root(t) \neq \emptyset$,

– et sa composante racine N^* est conservative fortement connexe.

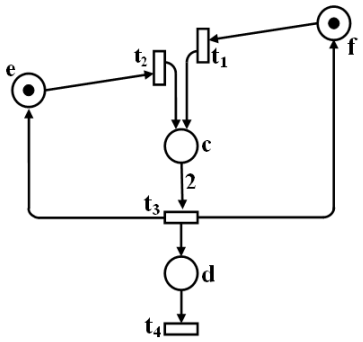


Figure 7.13. *La composante N^* du réseau racine de la figure 7.12*

Le réseau de la figure 7.12 n'est pas un Root-Système. Bien que $\forall t \in T, Root(t) \neq \emptyset$, sa composante racine N^* (figure 7.13) n'est pas fortement connexe.

PROPOSITION 7.38 *Soit $\langle N, M_0\rangle$ un Root-système, $\langle N, M_0\rangle$ est un K-Système et les trois énoncés suivants sont équivalents :*

1) $\langle N, M_0\rangle$ est pseudo-vivant

2) $\langle N, M_0\rangle$ satisfait la CS-propriété

3) $\langle N, M_0\rangle$ est vivant

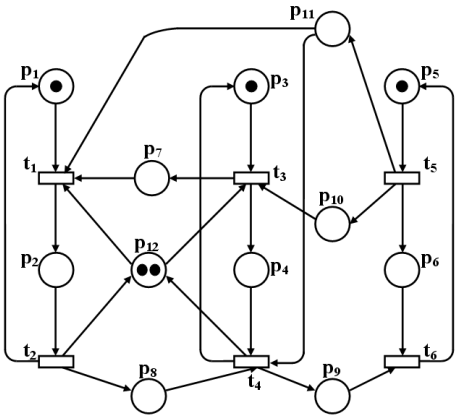


Figure 7.14. *Un exemple de Root Système*

Le réseau non ordonné de la figure 7.14 est un Root Système. Sa composante racine N^* est conservative et fortement connexe. Il comporte neuf siphons minimaux : $S_1 = \{p_5, p_6\}$, $S_2 = \{p_3, p_4\}$, $S_3 = \{p_1, p_2\}$, $S_4 = \{p_5, p_{10}, p_4, p_9\}$, $S_5 = \{p_5, p_{10}, p_7, p_2, p_8, p_9\}$, $S_6 = \{p_{12}, p_2, p_4\}$, $S_7 = \{p_3, p_7, p_2, p_8\}$, $S_8 = \{p_5, p_{11}, p_9\}$, $S_9 = \{p_{12}, p_2, p_8\}$.

Ce Root-Système satisfait la CS-propriété et est donc vivant pour tout marquage initial vérifiant les condition suivantes :
$p_5 + p_6 > 0,\ \ p_3 + p_4 > 0,\ \ p_1 + p_2 > 0,\ \ p_5 + p_{10} + p_4 + p_9 > 0,$
$p_5 + p_{10} + p_7 + p_2 + p_8 + p_9 > 0,$
$p_{12} + p_2 + p_4 > 0,\ \ p_3 + p_7 + p_2 + p_8 > 0,$
$p_{11} + 2 \cdot p_9 + 2 \cdot p_5 - p_3 - p_7 > 0,$
$p_{12} + p_2 - p_3 > 0$

Un exemple d'un tel « bon » marquage est : $M_0 = p_1 + p_3 + p_5 + 2 \cdot p_{12}$.

La classe des DCS synchronisés. La structure imposée à la composante racine N^* est une condition suffisante, mais non nécessaire pour assurer la K-propriété et

l'appartenance à la classe des K-Systèmes. On montre dans [BAR 05] comment on peut construire un « bon » K-Système par composition judicieuse de Root-Systèmes (ou plus généralement de DCS) pris comme modules communiquant via des canaux représentés par des places.

La classe des réseaux obtenue via cette composition est appelée la classe des DCS synchronisés. Cette classe est une extension ainsi qu'une généralisation des *multi-level Deterministically Synchronized Sequential Processes* ((DS)*SP) de [TER 01] qui sont eux mêmes une généralisation des graphes déterministes de processus séquentiels (DSSP) [REI 82, SOU 93].

DÉFINITION 7.39 *Un réseau de Petri $\langle N, M_0\rangle$ où $N = \langle P, T, F, V\rangle$ est un SDCS (Synchronized dead closed system) ssi P est l'union disjointe de $P_1, ..., P_n$ et B*
T est l'union disjointe de $T_1, ..., T_n$
et où les conditions suivantes sont satisfaites :

1) pour tout $i \in \{1, ...n\}$, soit $N_i = \langle P_i, T_i, F_{\lfloor((P_i\times T_i)\cup(T_i\times P_i)}, V_{\lfloor((P_i\times T_i)\cup(T_i\times P_i)}\rangle$, alors $\langle N_i, M_{0_{\lfloor P_i}}\rangle$ est un DCS vivant.

2) pour tout $i, j \in \{1, ...n\}$, si $i \neq j$ alors $V_{\lfloor((P_i\times T_i)\cup(T_i\times P_i)} = 0$.

3) pour tout module $N_i, i \in \{1, ...n\}$:

- $\exists$ (un tampon) $b \in B \mid b^\bullet \subseteq T_i$ (un tampon á destination privée)

- $\forall b \in B$, b préserve les places racines de N_i (i.e. pour tout $t \in T_i, Root(t)_{N_i} \subseteq Root(t)_N$)

4) soit $B' \subseteq B$ l'ensemble des tampons à destination privée dans N alors il existe $B'' \subseteq B'$ tel que le sous-réseau induit par l'ensemble des $N_i, i \in \{1, ...n\}$ et B'' est conservatif et fortement connexe.

PROPOSITION 7.40 *Soit $\langle N, M_0\rangle$ un SDCS. $\langle N, M_0\rangle$ est un K-Système et les trois énoncés suivants sont équivalents :*

1) $\langle N, M_0\rangle$ est pseudo-vivant

2) $\langle N, M_0\rangle$ satisfait la CS-propriété

3) $\langle N, M_0\rangle$ est vivant

Le réseau de la figure 7.15 est composé de deux modules N_1 et N_2 communicants à l'aide de trois tampons (trois canaux) b_1, b_2 et b_3. Chacun de ces modules est un Root-Système. On peut constater que le buffer b_1 ne preserve pas le conflit entre les transitions t_1 et t_3 mais preserve leurs racines respectives (p_1 pour t_1 et p_3 pour t_3). Ce réseau global n'est pas un Root système (sa composante racine N^* induite par N_1 et N_2 et les buffers racine b_2 et b_3 (b_1 n'est pas une place racine) est fortement connexe mais non conservative, car le buffer b_3 n'est pas structurellement borné) mais c'est un SDCS. Les conditions énoncées dans la definition 7.4.4 sont satisfaites ($B'' = \{b_1, b_2\}$). Pour tout marquage initial vérifiant la CS-propriété, ce réseau est vivant. Un tel « bon » marquage est : $M_0 = p_1 + p_3 + p_5 + 2 \cdot b_{13}$.

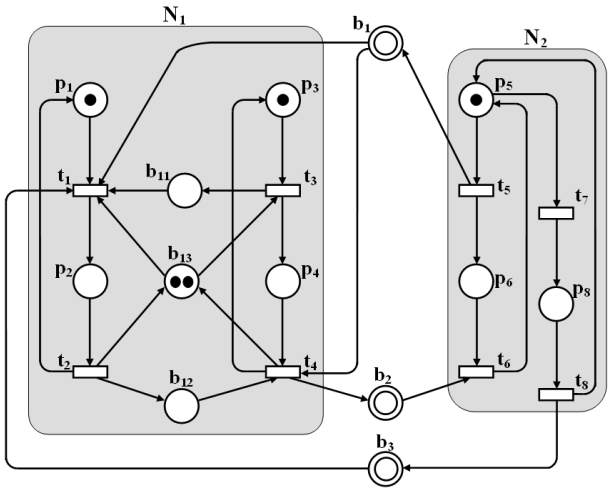


Figure 7.15. *Un exemple de SDCS*

7.5. Conclusion

L'analyse structurelle est un préalable indispensable à la vérification d'un modèle. En effet, la complexité des méthodes structurelles est généralement réduite et certainement négligeable devant celle des méthodes basées sur la génération de tout ou partie du graphe d'état.

Dans le cas le plus favorable de classes de modèles concurrents dont les synchronisations sont restreintes, les techniques structurelles conduisent à des procédures de décision. Dans le cas le plus général, elles fournissent des algorithmes de semi-décision à exécuter avant toute autre méthode de vérification.

De plus, elles présentent deux avantages déterminants. D'une part, elles sont souvent la seule alternative pour étudier les systèmes paramétrés (voir le chapitre 9, page 213). D'autre part, elles se combinent avec les méthodes « comportementales » afin de réduire la complexité empirique de ces méthodes (voir le chapitre 8, page 171).

7.6. Bibliographie

[BAR 89] BARKAOUI K., LEMAIRE B., « An effective characterization of minimal deadlocks and traps based on graph theory », *10 ème ICATPN*, Springer-Verlag, 1989.

[BAR 90] BARKAOUI K., MINOUX M., « Deadlocks and Traps in Petri Nets as Horn Satisfiability Solutions and Some Related Polynomially Solvable Problems », *Discrete Applied Mathematics*, vol. No. 29, 1990.

[BAR 92] BARKAOUI K., MINOUX M., « A Polynomial Time Graph Algorithm to Decide Liveness of some basic classes of bounded Petri nets », *LNCS*, vol. No. 616, p. 62-75, Springer-Verlag, 1992.

[BAR 96] BARKAOUI K., PRADAT-PEYRE J., « On liveness and Controlled Siphons in Petri nets », REISIG, Ed., *Petri Nets, Theory and Application*, n° 1091LNCS, Springer-Verlag, 1996.

[BAR 05] BARKAOUI K., COUVREUR J. M., KLAI K., « On the equivalence between liveness and dead-lock freeness in Petri nets », CIARDO, DARONDEAU, Eds., *Petri Nets, Theory and Application*, n° 3536LNCS, Springer-Verlag, 2005.

[BER 80] BERTHELOT G., ROUCAIROL G., VALK R., « Reduction of Nets and Parallel Programs. », BRAUER W., Ed., *LNCS : Net Theory and Applications*, vol. 84, Berlin, Heidelberg, New York, Springer-Verlag, p. 277–290, 1980.

[BER 83] BERTHELOT G., Transformation et analyse de réseaux de Petri, applications aux protocoles., Thèse d'état, Université Pierre et Marie Curie, Paris, 1983.

[BER 86] BERTHELOT G., « Transformations and decompositions of nets », *Advances in Petri Nets*, n° 254LNCS, Springer-Verlag, p. 359-376, 1986.

[COL 91] COLOM J. M., SILVA M., « Convex Geometry and Semiflows in P/T Nets. A Comparative Study of Algorithms for Computation of Minimal P-Semiflows. », *Lecture Notes in Computer Science ; Advances in Petri Nets 1990*, vol. 483, p. 79–112, Springer-Verlag, 1991.

[DES 92] DESEL J., « A proof of the rank theorem for Extended Free choice nets », *LNCS*, vol. No. 616, p. 134-153, Springer-Verlag, 1992.

[DES 95] DESEL J., ESPARZA J., *Free Choice Petri nets.*, N° 40, Cambridge University Press Cambridge Tracts in Theoretical Computer Science, 1995.

[LAU 94] LAUTENBACH K., RIDDER H., « Liveness in Bounded Petri Nets Which Are Covered by T-Invariants. », *Application and Theory of Petri Nets*, p. 358-375, 1994.

[POI 00] POITRENAUD D., PRADAT-PEYRE J., « Pre and Post-Agglomerations for LTL Model Checking », NIELSEN M., SIMPSON D., Eds., *High-level Petri Nets, Theory and Application*, n° 1825LNCS, Springer-Verlag, p. 387-408, 2000.

[REI 82] REISIG W., « Deterministic buffer synchronisation of sequential processes », *Acta Informatica*, n° 18, p. 117-134, 1982.

[SOU 93] SOUISSI Y., « Deterministic systems of sequential processes : a class of structred Petri nets », *Petri Nets, Theory and Application*, n° 674LNCS, Springer-Verlag, p. 62–81, 1993.

[TER 96] TERUEL E., SILVA M., « Structure Theory of Equal Conflict systems. », *Theoretical Computer Science*, vol. 153, n°1–2, p. 271–300, 1996.

[TER 01] TERUEL E., RECALDE L., SILVA M., « Structure Theory of multi-level deterministically synchronized sequential processes. », *Theoretical Computer Science*, vol. 254, n°1–2, p. 1–33, 2001.

Chapitre 8

Vérification efficace de systèmes finis

Ce chapitre présente certaines approches adaptées à la vérification comportementale de systèmes finis. On se situe ici dans le cadre du *model checking*, qui procède par exploration exhaustive de l'espace d'états. Cette approche se heurte au problème d'explosion combinatoire de l'espace d'états, et nécessite des techniques particulières pour la génération et le stockage de l'espace d'états.

Nous commençons par introduire la notion de graphe d'accessibilité et son intérêt pour la vérification de propriétés comportementales, notamment exprimées à l'aide de logique temporelle. Nous présentons ensuite une classification des plus importantes familles d'approches dédiées au *model checking*. Enfin nous verrons plus en détail trois de ces approches, basées respectivement sur les diagrammes de décision réduits, les notions d'ordre partiel, et l'exploitation des symétries d'un système.

8.1. Vérification formelle par *model checking*

Nous présentons ici la problématique de génération d'un espace d'états, et la façon d'exploiter un graphe d'accessibilité pour la vérification formelle de propriétés comportementales du système.

8.1.1. *Graphe d'accessibilité*

Le *model checking* consiste à explorer toutes les configurations ou états accessibles d'un système. On parle de graphe d'accessibilité, dont les nœuds représentent

Chapitre rédigé par Jean-Michel ILIÉ et Yann THIERRY-MIEG et Soheib BAARIR.

les états du système et les arcs les occurrences d'évènements produisant les changements d'états. Le principe est simple, on procède par induction en considérant un système muni d'un état initial s_0 et d'une relation de transition T décrivant l'ensemble des évènements qui peuvent se produire dans le système.

```
1  Algorithme GrapheAccessibilité(M)
   Données : Soient,
   entrée : M : ⟨s0, T⟩ un système et son état initial
   local : todo : initialement vide, ensemble des états connus restants à explorer
   sortie : graph : le graphe d'accessibilité à construire
2  début
3      todo.push(s0)
4      tant que todo ≠ ∅ faire
5          s := todo.pop()
6          pour tous les t ∈ T faire
7              si t franchissable depuis s alors
8                  s' :=succ(s, t)
9                  si s' ∉ graph alors
10                     graph.addEtat(s')
11                     todo.push(s')
12                 graph.addArc(s,s')
13     Retourner graph
14 fin
```

Algorithme 4 : Construction du graphe d'accessibilité.

L'algorithme *GrapheAccessibilité* présente un algorithme simple de construction du graphe d'accessibilité en largeur d'abord quand $todo$ est une file, c'est-à-dire où tous les successeurs directs d'un nœud sont étudiés avant d'étudier leurs propres descendants (*Breadth-First Search*). Si $todo$ est une pile, on obtient une variante de parcours en profondeur d'abord (*Depth-First Search*), c'est-à-dire en explorant les descendants plutôt que les successeurs directs. L'exploration va donc suivre un chemin jusqu'à retomber sur un état déjà atteint, puis explorer le chemin suivant. Ce type d'algorithme DFS s'implémente également fréquemment à l'aide d'une pile d'appels récursifs. Quel que soit le sens du parcours, l'algorithme s'arrête lorsque tous les états ont été explorés. On peut examiner le graphe construit pour trouver des comportements souhaités ou au contraire gênants. La vérification de propriétés se ramène donc à une simple exploration de graphe.

Cette approche énumérative a comme avantages sa simplicité et l'absence d'intervention manuelle dans le processus de vérification. Il faut comparer ces aspects aux approches par preuve assistée à l'aide de « *theorem prover* », outils d'inférence qui

Processus	Etats	Processus	Etats
2	10	8	6562
4	82	10	59050
6	730	20	3486784402

Tableau 8.1. *Progression exponentielle du nombre d'états du système*

facilitent la démarche. Le système est alors vu comme des axiomes, et les propriétés à vérifier sont vues comme des théorèmes à démontrer. La preuve nécessite l'intervention d'un expert, alors qu'en *model checking* la preuve est entièrement automatisable. En revanche, le *model checking* requiert une représentation finie des comportements système pour tenir en mémoire, et passe donc mal à l'échelle quand la taille du système augmente.

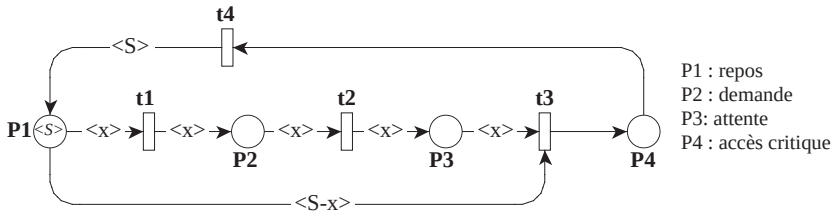


Figure 8.1. *Un réseau de Petri coloré représentant un système de contrôle de processus*

L'aspect critique pour que le *model checking* aboutisse est la taille de l'espace d'états, qui peut croître exponentiellement avec la taille du système. Le test de la ligne (9) est particulièrement critique, il nécessite habituellement un stockage des nœuds du graphe dans une table de hachage, qui doit tenir en mémoire vive.

Par exemple, la figure 8.1 présente un réseau de Petri coloré décrivant un système de contrôle de processus et leur synchronisation pour la réalisation d'une tâche en section critique. Le modèle est paramétré grâce à une variable <x> sur un domaine de couleurs distinguant les processus, $Proc = \{pr1, pr2, \ldots\}$. Les processus sont tous dans le même état initialement, ce qui est représenté par le fait qu'un jeton de chaque couleur du domaine des processus marque initialement la place $P1$ (de façon, générique, le symbole S dans $P1$ symbolise cette situation). Les processus peuvent indépendamment émettre une demande (place $P2$), qui une fois connue du contrôleur permettront aux processus d'être mis en attente active (place $P3$). Enfin, un processus en attente peut avoir accès à la section critique (place $P4$) quand tous les autres processus sont (re-)venus dans leur état initial (Ceci est symbolisé par l'expression $S - x$ sur l'arc $\langle P1, t3 \rangle$, sachant que x représente le processus qui requiert la section critique). Il est aisé de connaître le nombre d'états du système puisque chaque processus peut passer indépendamment par 3 états, puis atteindre de façon exclusive dans un

ultime état. Nous obtenons la formule : $3^n + 1$ pour un nombre n de processus. Le tableau de la figure 8.1 permet d'apprécier la progression exponentielle de la taille de l'espace d'états du système quand on augmente n.

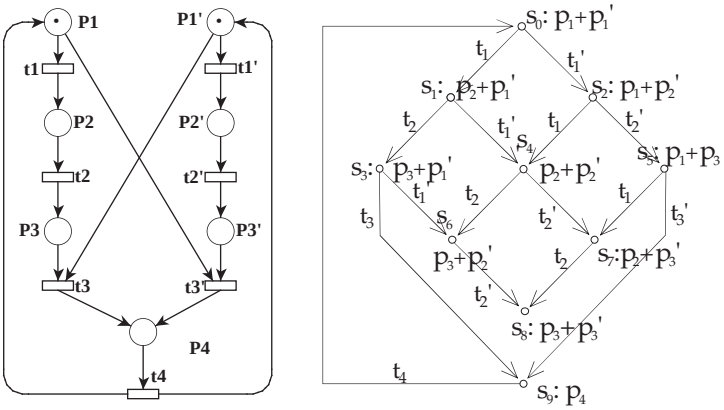


Figure 8.2. *réseau de Petri equivalent pour 2 processus et son graphe d'accessibilité.*

Les protocoles répartis contiennent souvent des erreurs subtiles, difficiles à reproduire, mais de nature à bloquer un système. L'indéterminisme de l'exécution autorise de nombreux entrelacements qu'il est difficile de cerner en raison de leur explosion combinatoire. Par exemple, le protocole de la figure 8.1 présente un problème d'interblocage. Il apparaît clairement dans l'espace d'état du RDP de la figure 8.2, qui reprend le même système mais en se limitant au cas simple de 2 processus concurrents. L'état $s_4 : [p3 + p3']$ est un état de blocage accessible, alors qu'il existe une infinité d'exécutions où le système boucle sur l'état $s_3 : [p4]$ et ne se bloque jamais : c'est un problème difficile à capturer par des approches basées sur le test.

Le calcul du graphe d'accessibilité est particulièrement intéressant si la vérification de la propriété est basée sur des études locales aux états et aux arcs du graphe. C'est le cas des propriétés d'interblocage et plus généralement des propriétés de sûreté. La vérification de propriétés comportementales plus poussées touchant à la notion de chemins ou de circuits peut aussi être envisagée, au prix d'algorithmes ad hoc. Ainsi, on peut tester si l'observation d'un évènement est attestée de façon récurrente. La recherche d'états d'accueil est aussi souvent utile pour confirmer la bonne tenue du système.

Afin d'éviter une trop forte multiplicité de tels algorithmes, un cadre général a été proposé : l'approche par automates. Dans les sections suivantes, nous introduirons

cette approche qui est adaptée la vérification des propriétés exprimées en logique temporelle, puis nous dresserons un panorama des grandes familles de techniques améliorant l'efficacité de la vérification (en section 8.2). Certaines de ces techniques jugées particulièrement représentatives seront ensuite détaillées. Afin de permettre des comparaisons, nous nous référerons à l'espace d'états de la figure 8.2.

8.1.2. *Approche automate du model checking*

Pour la vérification de propriétés d'accessibilité simples, comme la présence d'états bloquants (sans successeurs) ou le respect d'invariants, un simple parcours des nœuds du graphe d'accessibilité suffit. Pour des propriétés comportementales plus évoluées portant sur les enchaînements d'états du système, on utilise en général une variante de logique temporelle, dont les plus communes sont la logique temporelle linéaire LTL, et la logique arborescente CTL. La vérification de propriétés CTL est un domaine bien étudié [VAR 01], mais ne sera pas développé ici. Ce problème se ramène à une recherche d'accessibilité, pour laquelle plusieurs solutions sont déjà abordées dans ce chapitre. Nous présentons ici un cadre pour la vérification (plus complexe) de propriétés LTL.

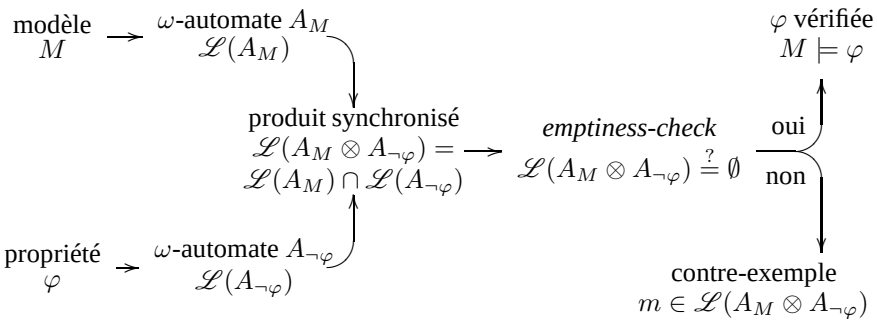


Figure 8.3. *Approche automate du model checking.*

Dans l'approche automate du *model checking* (Fig. 8.3), le modèle M d'un système à vérifier est vu comme un automate A_M, muni d'états et de transitions, reconnaissant des mots de longueur infinie (on parle d'ω-mots (omega-mots) et d'ω-automates). Les lettres de ces ω-mots correspondent chacune à une configuration ou état du système. Un ω-mot représente alors une séquence d'exécution du système qui traverse chacune de ces configurations. Le langage de l'automate (noté $\mathscr{L}(A_M)$) est l'ensemble des ω-mots qu'il reconnaît ; il représente l'ensemble des comportements possibles du système.

Par ailleurs, la propriété comportementale φ que l'on veut vérifier sur M est elle-même exprimée par un ω-automate $A_{\neg\varphi}$ dont le langage est l'ensemble des comportements qui *invalident* la propriété φ.

Un enchaînement de deux opérations permet de déterminer si $\mathscr{L}(A_M)$ et $\mathscr{L}(A_{\neg\varphi})$ partagent un ω-mot, c'est-à-dire s'il existe une exécution de M qui ne vérifie pas φ. D'abord le produit synchronisé des deux automates est construit, il s'agit d'un ω-automate reconnaissant l'intersection des deux langages. Ensuite un *emptiness-check* de ce produit est effectué, cette opération détermine si le langage reconnu par un automate est vide.

Le langage du produit est vide lorsqu'il n'existe aucune séquence d'exécution de M qui invalide φ. Dans le cas contraire un contre-exemple, c'est-à-dire un ω-mot représentant une exécution du système interdite par φ, peut être retourné.

Cette approche automate du *model checking* présente de nombreux avantages, en particulier, elle est très générale et s'applique à tout modèle fini. De plus elle permet d'englober une large classe de propriétés, et s'étend facilement à des modèles définis de façon compositionnelle, qui seront vus comme un produit synchronisé d'automates communiquants.

8.1.3. *Automates et logique temporelle*

Cette approche permet de manipuler des ensembles infinis (les langages) en les représentant par des structures finies (les ω-automates). Plusieurs types d'ω-automates existent qui se distinguent par leur forme ou leur sémantique. Les plus couramment utilisés, et qui nous intéresseront ici, sont appelés automates de Büchi.

Ainsi, la satisfaction d'une propriété exprimable par un automate de Büchi pourra être testée suivant l'approche de la section 8.1.2, par confrontation avec le modèle du système. Cependant, spécifier des propriétés directement sous cette forme n'est pas toujours aisé. On a recours à des logiques comme la *logique temporelle* (LTL) [VAR 96] dont la sémantique est simple et dont on sait transformer les énoncés en automates.

Les formules LTL sont construites à partir de variables propositionnelles appelées *propositions atomiques*, d'opérateurs booléens ($\neg$, $\vee$, $\wedge$, $\Rightarrow$, ...), et d'*opérateurs temporels* (F, G, U, X, ...).

Les propriétés atomiques permettent de caractériser les états du système. Il s'agit de variables booléennes dont la valuation est connue *de façon non-ambiguë* dans chaque état : un état s est dit étiqueté par une proposition atomique ϕ quand ϕ est vraie dans s. Les opérateurs temporels permettent de relier des états du système au sein d'une séquence d'exécution. Gf signifie que la sous-formule LTL f doit toujours

(G*enerally*) être vérifiée à partir de cet instant. $\mathsf{F}f$ indique que f doit être vérifiée à un instant ultérieur (F*uture*). $\mathsf{X}f$ impose que f soit vérifiée à l'instant immédiatement suivant (*ne*X*t*). Enfin $f \mathsf{U} g$ est vraie si f est vérifiée jusqu'à ce que g le soit (U*ntil*).

Par exemple, pour la figure 8.1, la proposition atomique $[P1.pr_1 + P2.pr_2]$ sera vraie chaque fois que la demande du processus pr_2 sera présente (place $P2$) et que simultanément pr_1 sera au repos (place $P1$). La formule $\mathsf{G}([P1.pr_1 + P2.pr_2] \Rightarrow \mathsf{F}[P4])$ spécifie que chaque fois que cette situation se présente, il se produira nécessairement plus tard un accès en section critique ($P4$). Sur le réseau de Petri à deux processus de la figure 8.2, $G([p_1 + p_2'] \Rightarrow \mathsf{F}[p_4])$ exprime la même propriété.

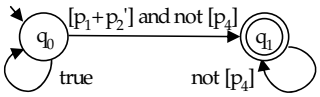


Figure 8.4. *Automate* $A_{\neg G([p_1+p_2'] \Longrightarrow [p_4])}$

L'automate de Büchi représenté figure 8.4 accepte toutes les séquences d'exécution qui invalident la propriété $G([p_1 + p_2'] \Rightarrow \mathsf{F}[p_4])$. Tout ω-mot accepté par cet automate correspond à une suite infinie d'états, démarrant de l'*état initial* de l'automate (noté par un arc sans source), et traversant une infinité de fois un *état d'acceptation* (état représenté par un double cercle). Le mot lui même est alors défini par la succession des étiquettes des arcs empruntés pour définir la séquence d'états. Par exemple, $(true)(true)([p_1 + p_2'] \wedge \neg[p_4])(\neg[p_4])...(\neg[p_4])$ est un mot valide. La proposition atomique true est considérée vraie dans tout état. Plus synthétiquement, les mots valides de notre automate sont définis par $(true)^*([p_1 + p_2'] \wedge \neg[p_4])^1(\neg[p_4])^\infty$, où $*$, 1 et ∞ représentent respectivement un nombre quelconque fini, une seule et une infinité d'occurrences d'états étiquetés par la proposition atomique. On remarquera que les séquences infinies de $true$ qui restent sur l'état initial ne sont pas acceptées (mots invalides).

8.1.4. *Automates et produit synchronisé*

Suivant l'approche illustrée en Figure 8.3, l'automate $A_{\neg\varphi}$ correspondant à la négation de la propriété devra être synchronisé avec l'automate A_M représentant tous les comportements du système, pour déterminer ensuite s'il existe un comportement invalidant la formule. La structure du produit synchronisé est un automate dont les nœuds sont des paires $\langle s, q\rangle$ où s est un état du système et q un état de l'automate de Büchi. Son état initial est $\langle s_0, q_0\rangle$ où s_0 est l'état initial du système et q_0 l'état initial de l'automate de Büchi. Pour tout nœud $\langle s_1, q_1\rangle$ de cette structure, les successeurs seront des nœuds $\langle s_2, q_2\rangle$ tels que :

– s_2 est un successeur de s_1 du point de vue du système,
– q_2 est un successeur de q_1 dans l'automate de Büchi et
– s_1 satisfait les propositions atomiques de l'arc $q_1 \rightarrow q_2$.

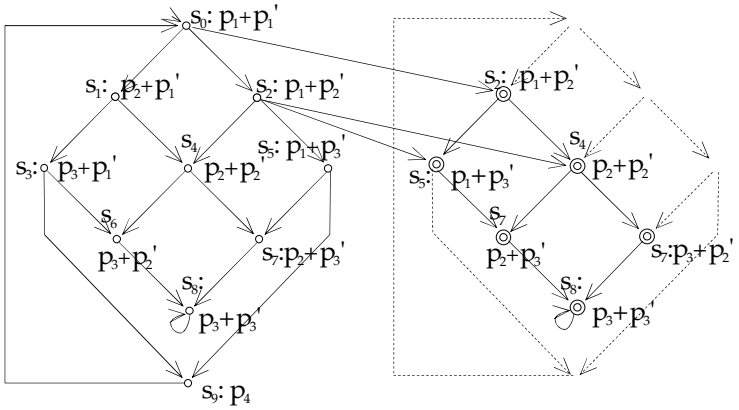


Figure 8.5. *Produit synchronisé avec l'automate* $A_{\neg \mathsf{G}([p_1+p_2'] \Rightarrow \mathsf{F}[p_4])}$

Toute séquence acceptée peut être exhibée comme un contre-exemple de la propriété désirée $\mathsf{G}([p_1 + p_2'] \Rightarrow \mathsf{F}[p_4])$, et le système ne satisfera la propriété que si le langage du produit synchronisé est vide (*emptiness-check*). Notons que pour qu'une séquence constitue un ω-mot elle doit être infinie ; on ajoute donc un arc qui boucle sur l'état bloquant s_3 pour représenter que le système n'évolue plus.

La figure 8.5 exhibe par ses arcs pleins le produit synchronisé du i de la figure 8.4. La partie gauche correspond à une synchronisation avec l'état q_0 de l'automate de Büchi et la partie droite avec q_1. Les pointillés représentent des transitions du système qui ne mènent pas vers des états étiquetés par les bonnes propositions atomiques : ils ne font pas partie du produit synchronisé.

A titre d'exemple, on peut déduire que la valeur de vérité de la propriété $\varphi : \mathsf{G}([p_1 + p_2'] \Rightarrow \mathsf{F}[p_4])$ est fausse puisque le ω-mot $\langle q_0, s_0\rangle\langle q_0, s_1\rangle\langle q_1, s_2\rangle\langle q_1, s_4\rangle(\langle q_1, s_5\rangle)^\infty$ est accepté, donc le langage de $A_M \otimes A_{\neg\varphi}$ n'est pas vide. Par un raisonnement similaire en construisant un produit synchronisé adapté, on déduira que $G([p_2 + p_2'] \Rightarrow \neg F[p_4])$ est vraie.

L'*emptiness-check* se réalise en temps et espace linéaire par rapport à la taille du produit synchronisé puisqu'il suffit de suivre les chemins acceptants de ce produit synchronisé. Cependant, la taille du produit synchronisé est au pire des cas le produit des tailles des deux automates.

La taille de l'automate $A_{\neg\varphi}$ croît de façon exponentielle en fonction du nombre d'objets de la formule (propositions atomiques et opérateurs). Fort heureusement, les formules LTL à vérifier s'avèrent petites et les automates correspondants n'ont en fait que quelques états. Il existe de plus des variantes plus compactes d'automates de Büchi, qui portent par exemple [GIA 02, COU 05] les conditions d'acceptation sur les transitions au lieu d'avoir des états d'acceptation.

La taille de l'automate A_M (espace d'états) croît de façon exponentielle avec la complexité du système. Cette explosion du nombre d'états est l'obstacle principal de l'approche automate, et est à l'origine de nombreuses techniques de réduction relatées maintenant.

8.2. Classification des techniques pour le *model checking*

Les techniques mises en place pour réaliser les outils de *model checking* sont nombreuses. Cependant nous pouvons les classer en deux grandes catégories : les approches qui opèrent par compression de l'espace mémoire utilisée et celles qui opèrent par abstraction, d'états ou de comportement, de façon à se concentrer sur les informations essentielles pour la vérification.

8.2.1. *Les approches par compression*

Pour faire tenir la représentation de l'espace d'états en mémoire, le premier type d'approches consiste simplement à compresser la représentation de l'espace d'états. Il est clair que réduire la taille de la signature d'un état permet d'en loger plus. Dans cette optique, on utilise des vecteurs de bits au lieu d'entiers, ou l'on compresse les états avec un algorithme classique (zip).

Une deuxième approche plus fine consiste à utiliser des tables annexes pour décoder et recoder l'état. On voit alors le système comme composé de parties ou composants C_i, chacun ayant un état propre. Les états locaux à chaque composant C_i sont logés dans des tables annexe H_i à mesure de leur découverte, ou suite à un pré-calcul. L'état global du système est représenté par un vecteur d'entiers interprétés comme des indices dans les tables d'états locaux des composants. Par exemple, l'état global $[3, 22, 7]$ se lit : C_0 est dans l'état $H_0[3]$, C_1 est dans l'état $H_1[22]$ ... Quand des composants partagent la même structure, ou qu'on a plusieurs instances du même composant dans le système, ils peuvent même partager leurs tables d'états locaux. Cette approche semble nécessaire dans le cas de modèles complexes, où la description de l'état local est volumineuse, donc l'utilisation d'indices fournit un bon taux de compression. Cette approche a été utilisée notamment dans [HOL 97] pour l'outil SPIN, et depuis reprise largement.

Une autre approche consiste à ne pas stocker les états, mais simplement leur présence. C'est le cas du *bit-state hash coding* qui n'utilise qu'un bit pour représenter la présence de l'état [STE 96], et permet donc d'arrête la construction (ligne (9) de l'algorithme présenté en tête de ce chapitre) sans stocker les états. La recherche d'états remplissant une certaine condition doit alors être réalisée à la volée, l'algorithme ne fournissant pas le graphe en sortie. Le risque encouru est la présence de collisions dans la table de hachage, qu'on ne pourra pas départager. Pour minimiser ce risque on peut utiliser des fonctions de hachage particulières, en suivant une approche probabiliste. L'idée est de faire plusieurs fois le calcul en utilisant à chaque fois des fonctions de hachage statistiquement indépendantes. La probabilité d'occurrence d'une collision peut être bornée à une valeur δ en fonction de la taille de la table et de son remplissage. Chaque répétition du calcul qui fournit un résultat d'accord avec les premiers réduit la probabilité d'occurrence d'une erreur [COU 92]. Par exemple, si la probabilité de collision est de 1%, et que trois calculs statistiquement indépendants fournissent le même résultat, il n'y a que $0,0001\%$ de chance que trois collisions aient eu lieu (et en plus au même endroit !).

Enfin, une structure de données arborescente partagée très dense a été proposée pour la représentation d'un ensemble d'états : les diagrammes de décision réduits. On manipule alors directement des ensembles d'états à chaque pas de l'algorithme, de façon dite « symbolique ». Les diagrammes de décision sont une méthode de codage efficace, qui exploite l'existence de propositions identiques entre états afin d'en partager la représentation. Cette catégorie de techniques sera détaillée en section 8.3.

8.2.2. *Les approches par classes d'équivalence*

Un deuxième type d'approches consiste à limiter la représentation de l'espace d'état aux informations pertinentes pour la vérification. On construit alors des graphes représentatifs de l'espace d'états complet, mais de taille réduite. Le plus souvent, ces techniques exploitent une relation d'équivalence particulière, afin d'assurer que la structure (réduite) qui est construite préserve les comportements pertinents vis-à-vis d'une propriété ou classe de propriétés. Par exemple, diverses relations d'équivalence préservant la présence d'interblocages, les propriétés d'accessibilité, ou la préservation de logique temporelle type LTL ont été proposées, et permettent souvent des réductions exponentielles de la taille de la représentation.

Comportements équivalents. Une classe de techniques ayant un grand succès est basée sur la notion d'ordre partiel d'occurrence des évènements au cours d'une séquence d'exécution. En effet, certaines séquences peuvent être considérées équivalentes, si l'on fait abstraction de l'ordre de certains évènements indépendants. On parle de *traces* représentatives d'un ensemble de séquences. Par exemple la trace $\{a, b\}.c$ représente les séquences $a.b.c$ et $b.a.c$, où l'on sait que a et b se produisent avant c, mais l'on ne fixe pas d'ordre d'occurrence entre a et b (donc on a un ordre partiel).

Ces techniques préservent globalement la logique dite LTL-X. Elles seront détaillées dans la section 8.4.

Ces relations d'équivalence basées sur les évènements consistent à distinguer les évènements dits « observables » et les évènements « invisibles ». Ces derniers ne modifient pas de façon visible l'état du système et peuvent donc se produire sans qu'on les observe. La notion de visibilité est fortement dépendante de la propriété que l'on veut tester, mais bien souvent pour une propriété fixée, une grosse partie du comportement peut être abstraite de cette manière. Par exemple, pour les propriétés globales d'absence d'interblocage, les évènements locaux aux composants (qui n'échangent pas de données avec les autres composants) peuvent être considérés invisibles. Cette approche par observation s'insère dans la logique proposée par l'approche automate du *model checking* , et s'applique donc très largement. Cette idée peut également être utilisée dans un contexte symbolique pour accélérer considérablement les procédures à base de diagrammes de décision.

Il y a un intérêt particulier à exploiter la notion d'observation dans le cadre des systèmes distribués car on peut chercher à vérifier des propriétés locales à certains composantes. Le paradigme de composants synchronisés ou automates communiquants pour représenter un système est également à l'origine de méthodes compositionnelles, où l'on cherche à prouver des propriétés en exploitant cette structure du système. L'approche par graphe de synchronisation [LAK 04] est un exemple mettant en œuvre ce principe. On construit un graphe dont les nœuds correspondent à des branchements ou synchronisations entre processus pertinents du point de vue de l'état global du système. Les graphe d'états internes aux composants sont stockés dans une structure de représentation particulière, qui permet d'accélérer la recherche des points de synchronisation.

États équivalents. Une première approche consiste à faire abstraction de parties de l'information concernant l'état du système. Pour les données on peut parfois simplifier la représentation [KES 98a], par exemple en assimilant la valeur d'un réel à l'un des trois symboles $\{+, -, 0\}$, signifiant qu'il a une valeur positive, négative ou nulle. Cette information est suffisante pour travailler, si l'on ne fait que des multiplications de réels, et qu'on ne s'intéresse pas à la valeur mais à son signe. Cette approche permet de limiter le nombre d'états distincts et aussi la taille de leur signature. Elle est supportée partiellement par des techniques informatiques de vérification informatiquement mais nécessite encore l'expertise humaine importante.

Un second type d'approches largement automatisé consiste à exploiter les symétries du système pour reconnaître des états comme équivalents du point de vue de son fonctionnement. On construit alors un graphe dont les nœuds sont des classes d'équivalence d'états. Selon la relation d'équivalence choisie et la présence de symétries dans le système, les gains en taille peuvent être exponentiels. Ces approches seront détaillées en section 8.5, et permet de préserver LTL.

Afin d'illustrer comment ces techniques agissent pour réduire l'espace d'états parcouru lors de la vérification d'une propriété, nous allons maintenant détailler les concepts de 3 d'entre elles en les confrontant toutes à notre exemple de référence donnée en figures 8.1 et 8.2. Il s'agit des
indexsymbolique !technique techniques symboliques basées sur les diagrammes de décision, des techniques d'ordre partiel, et des techniques qui exploitent les symétries.

8.3. Approches basées sur les diagrammes de décision

Cette section détaille les approches basées sur les diagrammes de décision. Les diagrammes de décision ont été introduits initialement pour la représentation compacte d'expressions booléennes [AKE 78]. Leur première application dans le cadre du *model checking* est proposée par Bryant dans le cadre de sa thèse [BRY 86], largement repris ensuite notamment dans le papier fondateur [BUR 92] où la puissance de cette technique est exhibée, permettant de gérer un nombre d'états supérieur de plusieurs ordres de grandeur (10^{20} états !) aux techniques explicites classiques.

8.3.1. *Binary Decision Diagram : BDD*

Diagramme de Décision Réduit. Un diagramme de décision binaire permet la représentation d'une fonction booléenne de façon compacte. Considérons par exemple la fonction booléenne $f = (a \vee b) \wedge c$ de trois variables a, b et c. Un arbre de décision représentant cette fonction est présenté en figure 8.6 (a). Cet arbre se lit depuis la racine, chaque chemin dans la structure donnant la valeur de vérité de f en fonction des valeurs des variables a, b et c.

Cette structure peut être représentée de façon compacte, en partageant des sous-chemins. Une première étape consiste à ne représenter que deux nœuds terminaux 0 et 1 (figure 8.6 (b)). Cette démarche peut être appliquée récursivement à tous les nœuds de l'arbre, ce qui donne la structure représentée en figure 8.6 (c). Cette structure s'appelle un ROBDD pour *Reduced Ordered Binary Decision Diagram* ou simplement BDD dans la suite : il est réduit car les nœuds sont partagés entre les chemins et ordonné car les variables sont toujours rencontrées dans le même ordre (a puis b puis c) de la racine aux feuilles.

Une autre optimisation est parfois encore appliquée afin de réduire la taille de la structure : l'élimination des tests redondants, présenté figure 8.6 (d). Les variables sont toujours rencontrées dans le même ordre, mais on permet aux arcs de « sauter » des niveaux, si la variable sautée n'influe pas sur la valeur de vérité de la fonction représentée. En pratique, cela signifie que tous ses arcs pointent vers le même nœud (comparez les figures (c) et (d)). Sur cette dernière figure il apparaît clairement que la fonction $f = (a \vee b) \wedge c$.

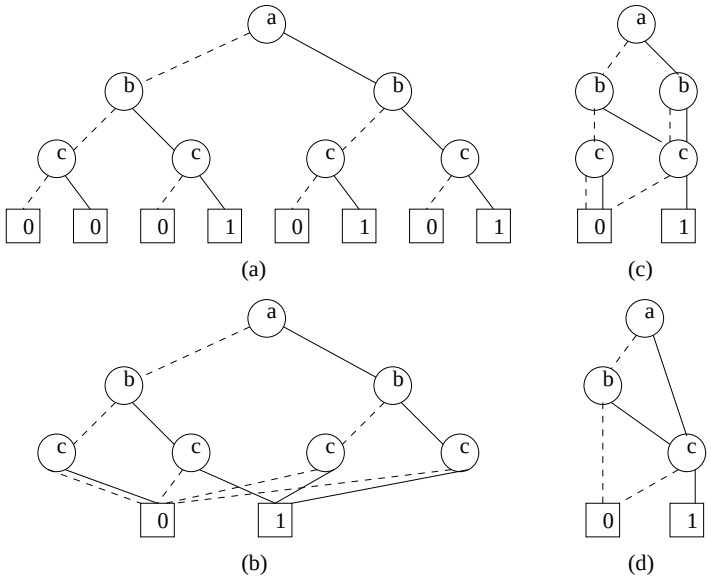


Figure 8.6. *Diagramme de décision binaire et réductions. Les traits pleins sont correspondent à vrai et les pointillés à faux. (a) diagramme complet sans réductions. (b) sans redondance des nœuds terminaux. (c) sans redondance des nœuds internes (d) sans tests redondants*

Ces techniques manipulent symboliquement des ensembles de valeurs (chaque nœud représente l'ensemble des chemins dont il est la racine). Ainsi, on utilise le terme « représentation symbolique » pour référer à ce type de techniques, par opposition aux représentations dites « explicites » ou chaque chemin est représenté séparément (comme dans une table de vérité).

8.3.2. *BDD pour la vérification de systèmes finis*

Dans le cadre de la vérification par *model checking*, on utilisera un BDD pour représenter un espace d'états d'un système. Les variables du BDD représentent alors les variables composant l'état du système, chaque chemin dans la structure menant au nœud terminal 1 définissant un état accessible.

Considérons par exemple le système exemple à deux processus décrit figure 8.2. La figure 8.7 présente son espace d'états, représenté sous la forme d'un BDD. Les variables du BDD sont nommées d'après les places du réseau, les valeurs portées sur l'arc indiquent le marquage de la place dans cet état. La figure se lit de gauche à droite, chaque état étant représenté par un chemin partant de la racine (P_0) et allant au terminal 1. Par exemple le chemin le plus haut de la figure est l'état p_7.

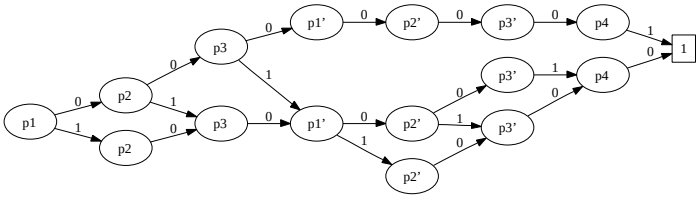


Figure 8.7. *Espace d'états de l'exemple de la figure 8.2 sous forme de BDD.*

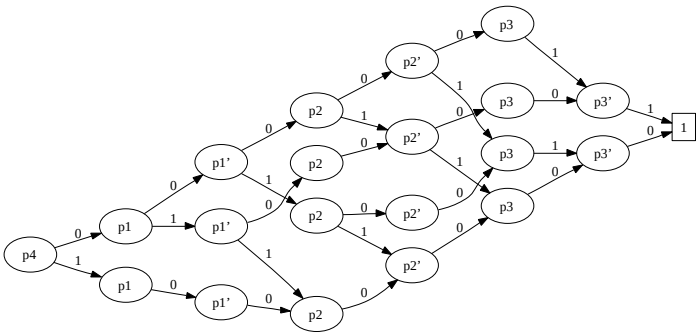


Figure 8.8. *Un ordre moins favorable sur les variables.*

Le tableau 8.2 permet d'apprécier sur notre exemple la progression linéaire de la taille du codage avec un ordre approprié, alors que le nombre d'états explose exponentiellement avec $3^n + 1$. En d'autres termes, pour 20 processus 3 milliards d'états sont représentés sur 160 nœuds BDD ! Le temps de construction (non représenté) est très inférieur à la seconde. On a affaire ici à une méthode de compression très efficace, qui travaille réellement sur les états et non sur des abstractions par classes d'équivalence comme les approches par symétrie (cf. tableau 8.3).

Cependant la **taille de la représentation est fortement liée à l'ordre choisi** pour mentionner les variables. La figure 8.8 représente le même espace d'états, mais avec

Processus	Etats	nœuds BDD (Ordre contigu)	nœuds BDD (Ordre entrelacé)
2	10	16	21
5	244	40	260
10	59050	80	13321
15	1.43489e+07	120	589838
20	3.48678e+09	160	-

Tableau 8.2. *Nombre de nœuds BDD en fonction du nombre de processus*

un codage où les variables correspondant aux places d'un processus sont entrelacées. Au lieu de $p_1 > p_2 > p_3 > p'_1 > p'_2 > p'_3$ on a choisi $p_1 > p'_1 > p_2 > p'_2 > p_3 > p'_3$. Dans le premier cas, l'ordre respecte la structure du système : les places liées à chaque processus (qui font partie d'un flot) sont contiguës dans la représentation, dans le deuxième au contraire, les variables de chaque processus sont entrelacées, menant à une représentation dont la taille explose. La dernière colonne du tableau exhibe cette explosion en taille pour un ordre défavorable, le calcul ne s'étant pas terminé pour 20 processus, sur une machine ayant 3 Go de RAM.

On peut donc facilement exhiber des exemples où un ordre peut fournir une taille linéaire au nombre de variables et d'autres une taille exponentielle. De plus la recherche d'un ordre optimal est un problème de classe de complexité NP-Complet [TAN 93], ce qui rend l'usage de ces méthodes capricieux. Des ordres raisonnables peuvent le plus souvent être trouvés en pratique, soit heuristiquement en examinant la structure du système (déjà évoqué dans [BRY 86] et reste la solution majoritairement retenue), soit à travers un réordonnancement dynamique des variables [RUD 93], lancé quand la taille de la représentation excède un certain seuil.

8.3.3. *Construction et manipulation*

Canonicité et cache. L'élaboration d'un BDD s'appuie sur une table d'unicité des nœuds déjà existants : comme la construction de l'arbre procède depuis les feuilles jusqu'à la racine, on référencera directement le nœud pré-existant plutôt que de créer un nouveau nœud qui aurait les mêmes successeurs. L'opération d'union définie récursivement (cf. algorithme 5) permet d'assurer par construction l'obtention de BDD canoniques. Elle s'appuie sur un cache d'opération afin que sa complexité soit liée au nombre de nœuds BDD dans la structure et non au nombre de chemins. La ligne (9) est critique, elle correspond à un appel récursif à l'union : l'union est construite comme un nouveau nœud avec un fils 0 calculé comme l'union du fils 0 de a et de b, et un fils 1 calculé comme l'union des fils 1 de a et b.

Les BDD permettent grâce à un cache d'opérations d'implémenter des algorithmes efficaces manipulant des ensembles de valeurs, dont la complexité est liée au nombre de nœuds du BDD et non au nombre de chemins. Par exemple les opérations ensemblistes d'union, intersection ou différence ensembliste entre deux BDD ont une complexité linéaire au produit du nombre de nœuds dans les deux structures. La création d'un BDD se fait toujours en contrôlant que le nœud n'existe pas déjà dans une table d'unicité (géré par $CreerBDD$ ligne (9) de l'union). L'unicité de représentation des nœuds permet la mise en place d'un cache efficace, sans lequel les opérations auraient une complexité liée au nombre de chemins dans les structures et non au nombre de nœuds. Or le nombre de chemins dans la structure peut être exponentiel par rapport aux nombre de nœuds utilisés pour les représenter, comme on l'a vu sur l'exemple.

```
1  bdd Union(bdd a,bdd b) :
2  début
3  |  si a = 0 ∨ b = 1 alors
4  |  |  retourner b
5  |  si b = 0 ∨ a = 1 alors
6  |  |  retourner a
7  |  si ⟨{a, b}, r⟩ ∈ Cache alors
8  |  |  retourner r
9  |  bdd r := CreerBDD(0 → Union(a[0], b[0]), 1 → Union(a[1], b[1]))
10 |  Cache.add(⟨{a, b}, r⟩)
11 |  retourner r
12 fin
```

Algorithme 5 : Opération d'union de deux BDD.

Relation de transition. Pour le *model checking*, la relation de transition est également le plus souvent représentée sous la forme d'un BDD, comprenant $2k$ niveaux si le BDD représentant l'espace d'états en compte k. La variable d'indice $2i, i \in 0..k$ du BDD de la transition représente la valeur avant franchissement, et la variable d'indice $2i + 1$ donne sa (ses) nouvelle(s) valeur(s). L'application du BDD T représentant une transition à un espace d'états S s'effectue comme un parcours attelé des deux BDD, contrôlé par un cache d'opération.

Ce codage des transitions se prête aux opérations d'union, permettant de manipuler la relation de transition du système T de façon symbolique, sans référer directement aux évènements t qui la composent. L'opération d'application d'un BDD T représentant la relation de transition à un BDD S représentant des états atteints permet d'obtenir l'ensemble des successeurs accessibles en un pas depuis les états de S en franchissant un des évènements de T. Cette opération reste en complexité facteur des tailles en nœuds des BDD T et S, mais peut produire un grand nombre d'états en une seule application. Ce codage symbolique de la relation de transition est lui-même sujet des problèmes de taille selon l'ordre des variables choisi ; beaucoup d'approches proposent de décomposer la relation de transitions en parties [J.R 91, MIN 99].

Deux variantes de l'algorithme de production de l'espace d'états sont présentés ici. Dans la version de gauche, la ligne (5) permet d'obtenir les successeurs des états $todo$ en cours d'exploration par une application de T, et la ligne (6) ne conserve que les états nouveaux. Le parcours est donc un parcours strictement en largeur de l'espace d'états.

De façon contre-intuitive, il est souvent plus efficace de procéder en suivant l'algorithme de droite qui réalise un point fixe. En effet, le parcours est encore en largeur,

Données : // Soit M : $\langle s_0, T \rangle$ un système, son état initial et son ensemble de transitions
$bdd\ todo$: les états nouveaux à explorer
$bdd\ graph$: les états accessibles déjà construits

```
début
    todo := {s0 }
    graph := {s0 }
    tant que todo ≠ ∅ faire
        bdd tmp := T(todo)
        todo := tmp \ graph
        graph := graph ∪ tmp
fin
```

```
début
    todo := { s0 }
    graph := { }
    tant que todo ≠ graph faire
        graph := todo
        todo := todo ∪ T(graph)
fin
```

Algorithme 6 : Deux variantes BDD pour produire l'espace d'états

mais les états sont progressivement ajoutés dans la structure, ce qui évite de construire les tmp intermédiaires du premier algorithme, dont les nœuds n'apparaîtront vraisemblablement pas dans le résultat final (donc encombrent les caches et tables d'unicité), et dont la taille de représentation peut être parfois être plus grande que $todo$ du deuxième l'algorithme. En effet, rappelons que le nombre de nœuds BDD dans la structure n'est pas directement lié au nombre d'états représenté, donc on n'est pas assuré que le $todo$ du premier algorithme ait une représentation petite, même s'il contient moins d'états. De plus, en raison d'un bon fonctionnement du cache, l'application de T dans la version de droite n'est pas nécessairement très coûteux.

En fait c'est là un des plus gros problème de ces méthodes symboliques, parfois, même si la représentation de l'espace d'états final est petite (car très dense), les BDD intermédiaires produits peuvent être très gros. C'est l'*effet de pic* selon lequel la taille de la représentation en BDD croît au cours de la construction jusqu'à un pic très élevé, parfois plusieurs ordres de grandeur supérieur à la représentation finale, avant de devenir dense. Cet effet de pic est attribué au sens du parcours en largeur d'abord, qui produit des états selon un ordre significatif du point de vue du graphe d'états, mais non pertinent vis-à-vis du BDD qui le représente.

Pour lutter contre cet effet de pic, des propositions récentes [CIA 03] exploitent la localité des actions pour réaliser un point fixe sur des parties de la relation de transition. Le principe est de remonter des feuilles : un nœud de niveau k est dit saturé si tous les évènements qui touchent les niveaux k à n (dessous dans l'arbre) ont été franchis sur les états dont il est la racine jusqu'à obtention d'un point fixe. Quand un nœud est saturé, on remonte au niveau $k-1$, on franchit les évènements qui touchent ce niveau $k-1$ et les niveaux inférieurs. A chaque fois qu'un de ces évènement modifie un nœud de niveau inférieur à $k-1$ (donc qui était saturé par construction), le nœud sera re-saturé avant de poursuivre la construction. Quand l'algorithme atteint la racine

au niveau 1, tous les évènements ont été franchis. Cet algorithme dit « par saturation » permet dans bien des cas d'éviter l'effet de pic, permettant la vérification de systèmes de très grande taille (à l'extrême 10^{626} états pour le diner de 1000 philosophes, calculé en quelques secondes).

Méthodes symboliques et logique temporelle. Les méthodes symboliques permettent naturellement de trouver des états satisfaisant une propriété donnée dans l'espace d'états accessibles. Ceci correspond à une simple opération de sélection dans la structure. La vérification d'invariants ou de propriétés de sureté est donc aisée, et de complexité faible (temps et espace constant) une fois l'espace d'états calculé.

Pour la vérification de propriétés exprimées à l'aide de logique temporelle arborescente (CTL), on peut constater que la vérification se ramène à une succession de points fixes imbriqués [BUR 92]. Or des algorithmes de point fixe efficace doivent déjà être développés pour la construction de l'espace d'états complet. La majorité des outils symboliques supportent donc la vérification de propriétés CTL.

Pour des propriétés LTL, il a été démontré qu'on pouvait se ramener à la vérification CTL en ajoutant des contraintes d'équité [E.M 94]. Cette approche a depuis été étendue par la définition d'un algorithme symbolique [KES 98b] permettant directement la vérification de propriétés LTL, prenant en compte des contraintes d'équité et de compassion. Pour la vérification de propriétés LTL-X, on peut s'appuyer sur une représentation hybride appelée graphe d'observation symbolique [HAD 04] dans laquelle on génère un graphe (représenté de façon explicite) dont les nœuds représentent des classes d'équivalence d'états, stockés sous forme symbolique. Cette méthode s'appuie sur la notion d'évènements observables, afin de limiter le nombre des classes d'équivalence, et donne de bons résultats en pratique. Par rapport aux techniques de vérification CTL, cette approche peut exploiter l'approche automate du *model checking* LTL et bénéficier aussi des techniques compatibles de réduction de la taille d'un espace d'état (par exemple, symétries, modularité).

8.3.4. *Extensions des BDD*

Pour appliquer les techniques symboliques à un problème, il faut tout d'abord trouver un codage pertinent des états du système. Avec les BDD classiques, cela signifie un codage sous la forme de vecteur booléens ; ce codage peut être mal approprié selon les modèles ce qui conduit à la définition d'extensions des BDD présentés ici. Les BDD sont vus en général comme des fonctions :$\mathbb{B}^n \longmapsto \mathbb{B}$.

Quand la fonction est creuse, c'est-à-dire majoritairement fausse, il peut être intéressant de supprimer les chemins menant au terminal 0. Ce sont les ZBDD [MIN 93] pour *Zero-suppressed BDD*, qui offrent une vision cohérente avec l'utilisation de BDD pour représenter un ensemble de valeurs plutôt qu'une fonction. Les seuls chemins

représentés sont ceux qui appartiennent à l'ensemble. Selon les problèmes, cette technique peut être plus ou moins efficace que les BDD. Les implémentations des BDD proposent souvent la suppression des chemins « faux » sous la forme d'une option à activer.

Une extension majeure des BDD est la définition d'une forme canonique pour les fonctions $\mathbb{B}^n \longmapsto \mathbb{R}$. Ce sont les BDD multi-terminaux (MTBDD [FUJ 97]) ou diagrammes de décision algébriques (ADD [BAH 93]). Le principe est que le nombre de valeurs de sorties est fini (et borné par 2^n), même si la fonction est à valeur dans $\mathbb{R}$. On peut donc facilement étendre les règles de construction afin de créer un nœud terminal pour chaque valeur effectivement atteinte, le reste de la construction n'étant pas modifié. En *model checking* ces structures ont notamment été utilisées pour traiter des problèmes stochastiques [HER 03], où chaque état est atteint avec un certain taux. Elles permettent en effet une représentation efficace des matrices de franchissement et des états.

Une autre extension consiste à considérer des fonctions $\mathbb{N}^n \longmapsto \mathbb{B}$. Ce sont les diagrammes de décision multi-valués ou MDD, définis dans [SRI 90] à travers une équivalence avec des BDD, et implantés en tant que tel avec des nœuds à plusieurs fils dans [MIN 99]. Si le domaine des variables du MDD est borné *a priori*, on peut noter qu'un encodage binaire des variables est toujours possible, en représentant une variable MDD bornée par n par $\log_2(n)$ variables BDD. Les MDD offrent parfois une représentation beaucoup plus efficace, car les chemins comptent moins de variables, et leur interprétation est plus aisée (une variable MDD par variable du système). En revanche, pour certains énoncés, si le nombre de fils par nœud croît trop fortement, les algorithmes perdent de leur efficacité, et le coût de stockage des nœuds peut devenir important. En pratique, ces arbres sont *zero-suppressed*, les chemins menant au terminal 0 ne sont pas représentés. Cette technique peut être conjointement utilisée avec des terminaux multiples.

Un problème commun à toutes ces représentations est la contrainte liée à l'ordre d'occurrence des variables, et au nombre de variables intervenant dans chaque chemin, qui doivent être constants le long de tous les chemins pour assurer la canonicité de la représentation. Les diagrammes de décision de donnée (DDD [COU 02]) lèvent partiellement cette contrainte dans le cadre d'arbres de décision multi-valués. Cette structure plus dynamique permet de capturer l'état d'objets de taille variable (files, listes...) grâce à la définition d'opérateurs de concaténation. Une autre originalité des DDD est la définition des relations de transition par des *homomorphismes* au lieu d'une représentation symbolique à $2k$ niveaux. Les homomorphismes sont un mécanisme puissant et flexible qui permet la définition aisée de manipulations complexes, en assurant la transparence des mécanisme de cache et d'évaluation symbolique. Les algorithmes « par saturation » par exemple s'expriment facilement en termes d'homomorphismes.

8.4. Approches par ordre partiel

Les méthodes à base d'ordre partiel attaquent un des facteurs majeurs d'explosion de l'espace d'états : l'entrelacement des exécutions augmente le nombre d'états, même lorsque ces évènements sont en réalité indépendants. Ceci peut provoquer une explosion combinatoire de l'espace d'états. Par exemple, pour un système comportant n transitions qui peuvent être exécutées de façon indépendante à partir d'un état donné, il existe $n!$ séquences de franchissements à considérer, si l'on prend en compte les entrelacements. Étant donné que la seule différence entre ces séquences provient de leur ordre, la représentation de l'espace d'états peut être réduite en définissant un sous-ensemble de séquences représentatives de l'entrelacement. Mazurkiewicz dans [MAZ 86]a posé les fondations de cette théorie, en définissant la notion de *trace*. Une trace représente une classe d'équivalence de séquences de franchissements, tel que deux séquences dans une classe puissent être obtenus en permutant des franchissements d'évènements adjacents dans la trace. En pratique, la relation d'indépendance de deux évènements est définie par opposition aux relations de dépendances : deux évènements sont en conflit si le déclenchement de l'un désactive l'autre, ou sont en relation causale si le déclenchement de l'un active l'autre.

On appelle ce type de techniques « réductions d'ordre partiel » car les graphes de traces qu'ils construisent représentent en fait un ordre partiel dans l'ordre d'occurrence des évènements.

Graphes de traces. Une première exploitation de cette technique consiste à réduire la représentation du graphe d'états accessibles en retirant les séquences qui sont redondantes vis-à-vis de la représentation par traces. Les graphes basés sur les traces ont notamment été étudiés par Valmari [VAL 90, VAL 93], Godefroid et Wolper [GOD 92, GOD 96] et Peled [PEL 94c, PEL 94b].

Covering Step Graph. On peut citer également l'approche dite des « pas couvrants » ou *Covering Step Graph* (CSG). La construction d'un CSG est une proposition récente [VER 96, VER 97] visant à réduire les entrelacements dans les séquences de franchissements. Le principe consiste à tirer profit de l'indépendance des évènements pour tirer *simultanément* des ensembles de transitions en un seul pas. On obtient ainsi une nouvelle structure, le CSG, dans lequel les états sont des états du graphe d'accessibilité, mais les arcs représentent des *pas*, ou tir simultané de plusieurs transitions. Le CSG permet des réductions importantes quand des processus évoluent de façon indépendante, mais atteignent périodiquement la même configuration, ou des configurations équivalentes.

Des exemples présentés dans [VER 96] montrent que le CSG peut parfois être linéairement meilleur en complexité qu'un graphe de traces. La propriété agréable que toutes les séquences sont couvertes dans un CSG induit que plusieurs propriétés de

chemin puissent être vérifiées. En particulier, la détection d'états bloquants ou la vivacité des transitions (au sens réseau de Petri, décrivant une résistance dans la durée du système aux perturbations). A la différence des méthodes d'ensembles persistants, aucun proviso n'est nécessaire. De plus, l'algorithme est aisément adaptable pour prendre en compte les transitions observables. En particulier, la vérification de propriétés de logique temporelle est possible si chaque pas est contraint de manière à ne contenir qu'au plus une seule transition observable. Cette condition permet d'assurer la préservation de tous les entrelacements observables d'évènements. Cette méthode s'applique également bien pour la construction de graphes d'états temporisés, et peut se combiner avec l'utilisation d'ensembles persistants [RIB 02].

Dépliages. Une troisième approche vise à directement représenter l'ordre partiel des franchissements, en réutilisant les notions de concurrence et de conflit des réseaux Place/Transition. Cette approche a été introduite par [NIE 81, WIN 87] en introduisant une traduction d'un réseau de Petri général dans une classe particulière de réseau de Petri étiqueté, appelée processus arborescent (*Branching process*). Une telle traduction est appelée un dépliage, vu qu'une transition (et par extension, une place) peut être représentée plusieurs fois dans les processus, selon l'ordre de son occurrence. Dans cette représentation, les évènements indépendants sont modélisés par des transitions indépendantes, exhibant ainsi leurs entrelacements possibles sans les représenter explicitement. Ces techniques par dépliage exploitent la notion de localité d'un évènement, c'est-à-dire le fait qu'une transition d'un réseau de Petri ne touche qu'un nombre limité de places du système. Plusieurs travaux ont suivi cette approche, introduisant des algorithmes efficaces pour construire un tel dépliage, et l'utiliser pour vérifier des propriétés [MCM 95, ESP 92, ESP 93, KON 96, COU 96].

Ces différentes méthodes à base d'ordre partiel sont utilisées pour vérifier une classe très large de propriétés de logique temporelle. La section qui suit introduit de façon plus détaillée les méthodes basées sur les graphes de traces : ensembles persistants et ensembles en sommeil ; les techniques basées sur le dépliage et la technique CSG ne seront pas développées dans ce livre. La figure 8.2 d'exemple à deux processus sera utilisé dans la présentation pour illustrer les réductions des différentes techniques.

8.4.1. *Traces et verification*

Dans un espace d'états, l'ordre partiel est lié à une certaine relation d'indépendance entre des franchissements de transitions, appelée propriété « diamant », c'est-à-dire que si une séquence $t_1.t_2$ est valide depuis un état, alors $t2.t1$ l'est aussi. De plus ces deux séquences doivent atteindre le même état. Selon cette relation d'indépendance, les scénarios d'exécution d'un système peuvent être regroupés en classes d'équivalence appelées traces. Deux séquences appartiennent à la même trace (ou sont considérées équivalentes) si l'on peut obtenir l'une à partir de l'autre en permutant

successivement des occurrences d'évènements adjacents dans la séquence et indépendants.

Dans l'espace d'états de la figure 8.2, on peut noter que (t_1, t'_1) depuis l'état $p_1 + p'_1$, (t_2, t'_1) depuis l'état $p_2 + p'_1$, (t_1, t'_2) depuis l'état $p_1 + p'_2$ et (t_2, t'_2) depuis l'état $p_2 + p'_2$ sont des paires de franchissements indépendants. De ce fait, les séquences depuis l'état $p_2 + p'_1$ comme $t_2.t'_1.t'_2$, $t'_1.t_2.t'_2$, $t'_1.t'_2.t_2$ peuvent être considérées équivalentes et représentées par une unique trace, par exemple $[t_2.t'_1.t'_2]$.

Notion de trace. Dans un graphe d'accessibilité, le nombre d'entrelacements peut être réduit en se focalisant sur des séquences représentatives des traces du système. Plus précisément, une trace est représentée dans un graphe G si toutes les séquences qu'elle contient sont des préfixes des séquences de G, ou des permutations valides (du point de vue de l'indépendance) de ces préfixes. Cette relation de préfixe est introduite pour obtenir un graphe plus compact.

Par exemple la séquence $t'_1.t_2$ est un préfixe de $t'_1.t_2.t'_2$, donc elle peut être représentée par la trace $[t_2.t'_1.t'_2]$. Les figures 8.9, 8.10(a) et 8.10(b) exhibent trois graphes réduits grâce à cette définition. Dans ces figures, les arcs en gras représentent les franchissements considérés représentatifs, et les arcs grisés des arcs qui ne sont pas construits (pour permettre de comparer à la figure 8.2). Ces graphes diffèrent par les représentants choisis pour les traces, mais ils couvrent tous l'ensemble des traces franchissables depuis l'état initial du système, c'est-à-dire les trois traces : $[t_1.t_2.t'_1.t'_2]$, $[[t_1.t_2.t_3 + t'_1.t'_2.t'_3]^*.t_4.t_1.t_2.t'_1.t'_2]$ et $[[t_1.t_2.t_3 + t'_1.t'_2.t'_3].t_4]^\infty$. La meilleure réduction pour cet exemple est obtenue sur la figure 8.10(b), vu qu'on ne peut réduire plus le graphe sans perdre de traces.

Il est essentiel que la technique n'introduise pas des comportements incorrects vis-à-vis de la propriété à vérifier. De ce fait, le graphe réduit se doit d'être un sous-ensemble du graphe complet pour ne pas introduire de nouveaux comportements. De plus, l'utilisation de sous-traces (au sens de l'inclusion de traces) est dangereuse car elles peuvent être interprétées à tort comme des traces complètes, modifiant ainsi le comportement. Ainsi, sur la figure 8.10(a), en l'absence d'autres informations, on pourrait croire que la sous-trace $t_1.t'_1.t'_2$ mène à un état bloquant, ce qui est faux sur le graphe complet de la figure 8.2.

En utilisant un graphe de traces valide, de nombreuses propriétés peuvent être vérifiées. Une représentation partielle des traces suffit même parfois. Par exemple, la propriété de quasi-vivacité (pour toute transition t du système, il existe une séquence pouvant être tirée depuis l'état initial qui inclut t) peut être vérifiée en ne conservant qu'une seule trace par transition. Les invariants d'état peuvent également être traités, en introduisant une transition qui sélectionne les états de faute, et en étudiant sa quasi-vivacité. Par exemple un invariant exprimant « p_2 et p'_2 ne sont jamais simultanément

marquées », s'étudie en introduisant une transition franchissable quand p_2 et p'_2 sont marquées.

Les propriétés plus générales de sûreté et de vivacité requièrent plus d'efforts car il faut distinguer les transitions *observables* et *invisibles* vis-à-vis de la propriété. Une transition est dite observable si son franchissement peut affecter la valeur de vérité d'une proposition atomique intervenant dans la propriété, et invisible si au contraire son franchissement n'a pas directement d'effet sur la propriété. Dans le contexte simple des réseaux de Petri, une transition peut être définie comme observable si l'une des places connectées est concernée par une proposition atomique.

Graphe de traces *valide*. De fait, trois points clés doivent être correctement pris en compte lors de la construction, étant donné que les formules de logique temporelle peuvent être sensibles à l'ordre d'occurrence des évènements.

1) Le jeu de traces construites doit couvrir tous les entrelacements possibles des transitions observables, afin de permettre la comparaison (*emptiness-check*) avec l'automate représentant la formule.

2) En conséquence, les réductions ne peuvent concerner que les transitions invisibles. Ceci permet de préserver des propriétés de sûreté et de vivacité concernant des séquences infinies de transitions observables. Cependant, une séquence contenant une boucle d'évènements invisibles risque d'être perdue. Une telle séquence, qualifiée de divergente, doit être capturée par le graphe réduit afin d'assurer dans le cas général la préservation de propriétés. Cependant la représentation d'un sous-ensemble de ces séquences divergentes peut parfois suffire.

3) Le troisième point à considérer est la classe de formules qui peuvent bénéficier de la construction par graphe de traces. De fait, pour satisfaire le point numéro (1), il faudrait construire le graphe complet de façon habituelle si tous les évènements sont observables, ce qui élimine l'intérêt de l'approche. C'est le cas par exemple de propriétés d'état (par opposition aux propriétés évènementielles) intervenant dans une formule temporelle contenant l'opérateur de successeur immédiat (neXt) ; plusieurs travaux proposent donc de limiter la logique utilisée à LTL-X (où X signifie l'opérateur neXt) [WIL 96]. Ces formules font partie d'une large famille de logique dite insensible au bégaiement (*stuttering-invariant*) : c'est-à-dire que la valeur de vérité de la formule ne sera pas affecté par une répétition finie de la valeur de vérité d'une proposition atomique. Si la séquence $a.b.c$ est acceptée, les séquences $a^+.b^+.c^+$ sont aussi acceptées, où a^+ désigne une séquence finie non-vide de répétitions de a. Si l'on considère des propriétés qui ne sont pas insensibles au bégaiement, toutes les transitions doivent être observées, car par définition les transitions invisibles causent cette répétition des valeurs des propriétés atomiques. Dans [PEL 96], un algorithme élégant permettant de déterminer si une propriété respecte la condition de *stuttering-invariance* est présenté.

Bien sûr, il serait contre-productif de détecter les (in)dépendances des évènements en construisant le graphe d'accessibilité. En pratique, les graphes de traces sont construits à la volée, l'(in)dépendance des évènements franchissables depuis un état nouvellement atteint étant le plus souvent calculable. Pour les réseaux de Petri, ce calcul s'appuie sur la notion de conflit de transitions introduit plus haut, s'appliquant uniquement aux transitions franchissables. On peut choisir un sous-ensemble de transitions à franchir, les transitions franchissables mais non choisies restant franchissables dans l'état successeur. Dans le cadre de programmes multi-taches, les actions d'un processus peuvent être considérées indépendantes de celles des autres processus si elles ne manipulent que des variables « locales » au processus. Le graphe obtenu ainsi est une représentation réduite du graphe d'accessibilité, cependant il faut encore s'assurer que cette représentation préserve les propriétés souhaitées.

Interprétation des graphes de traces. Deux problèmes peuvent survenir quand les transitions franchissables ne sont pas sélectionnées :

– le premier est le problème des « transitions ignorées », selon lequel certains évènements peuvent être perpétuellement franchissables, sans jamais être sélectionnés. Ce problème apparaît dès qu'un jeu de transitions est isolé, ou indépendant du reste du système. Par exemple, avec un programme comportant deux processus indépendants et un graphe de traces construit en sélectionnant systématiquement les actions du premier processus quand les deux processus peuvent progresser, notre construction « ignorerait » les actions du deuxième processus.

– le second est le problème de « confusion », selon lequel certains états peuvent ne pas être visités alors qu'ils débutent des séquences qui ne sont pas représentées par ailleurs. Sur l'exemple de la figure 8.2 avec la construction de graphe de la figure 8.9, bien que t_1 et t'_1 ne soient pas en conflit dans l'état initial, il faut pourtant franchir ces deux évènements. De fait, si l'on ne choisit qu'un seul de ces évènements, disons t_1, alors toutes les séquences contenant t'_3 seront perdues vu que l'état $p_1 + p'_3$ ne sera jamais rencontré. Ceci est dû à un conflit entre t_1 et t'_3 qui n'est pas visible dans l'état initial (car t'_3 n'est pas encore franchissable), mais qui devient problématique après la séquence $t'_1.t'_2$. Ainsi une relation de dépendance peut être induite par des dépendances causales entre évènements.

Nous allons ici détailler trois algorithmes permettant la construction complète d'un graphe de traces valide. La première, basée sur les *ensembles persistants*, est une application directe du discours ci-dessus. L'approche par *ensembles en sommeil* (*sleep set*) est complémentaire de la première approche. Enfin, la technique du *graphe de pas couvrant* (*covering step graph*) est une construction particulière qui exploite ces mêmes notions de traces.

8.4.2. *Recherches d'ensembles persistants*

Un ensemble de transitions T franchissables depuis un état donné s est dit ensemble persistant si pour toute transition $t \in T$, et pour toute séquence $t.w$ franchissable depuis s, où w est une séquence de transitions choisies hors de T, les séquences construites en permutant t avec un des éléments de w sont franchissables depuis s, et mènent toutes au même état. Les transitions de l'ensemble persistant sont donc indépendantes des transitions qui n'en font pas partie. Dans le cas limite où toutes les transitions sont dépendantes les unes des autres, l'ensemble persistant se ramène à l'ensemble des transitions franchissables.

A partir de cette notion d'ensemble persistant, on construit un graphe de trace de la manière suivante :

– quand un nouvel état est atteint, l'algorithme calcule un ensemble persistant contenant au moins une transition franchissable (à condition que l'état ne soit pas bloquant)

– les successeurs de l'état qui sont effectivement construits sont les successeurs par les transitions de l'ensemble persistant.

Cette technique peut être vue comme une linéarisation du franchissement de transitions indépendantes. De fait, les transitions franchissables qui ne sont pas sélectionnées restent franchissables dans l'état successeur. Cette construction réduit donc la taille de la représentation tout en préservant les interblocages et l'existence de séquences de franchissements infinies. Cette construction peut présenter le problème des transitions ignorées, selon le choix fait pour constituer l'ensemble persistant dans chaque état. Cependant, pour certaines classes de modèle (par exemple les réseaux de Petri borné et fortement connexe [VAL 89]), on peut montrer l'absence de transitions non-franchies.

Proviso. Dans d'autres cas, on peut contourner le problème en s'assurant dans chaque état que l'on respecte des contraintes supplémentaires, appelées *provisos* (littéralement, une condition « pourvu que » ou condition suffisante). Dans [VAL 89], un premier proviso est proposé, en s'appuyant sur la détection durant la construction de toutes les composantes fortement connexes non-triviales du graphe d'accessibilité. Si dans une de ces composantes une transition franchissable n'est pas jamais sélectionnée, on étend l'ensemble persistant pour les couvrir également. Un autre proviso plus simple est proposé dans [HOL 92]. Il consiste à détecter les états depuis lesquels une transition franchissable ferme un circuit. Dans un tel état, toutes les transitions franchissables seront effectivement franchies. Le fait qu'il n'y ait pas de de transitions franchissables mais ignorées assure la construction d'un graphe de traces valide préservant les interblocages ainsi que les invariants d'état. D'autres provisos ont été

proposés, notamment dans [VAL 90], ou plus spécifiquement pour assurer un bon déroulement de la phase de synchronisation d'automates (voir section 8.1.3) à la volée [VAL 93, PEL 94a].

Calcul d'ensembles persistants. En se basant sur l'analyse structurelle d'un réseau de Petri, différents algorithmes ont été proposés pour calculer efficacement des ensembles persistants dans chaque état visité, des plus simples utilisant une simple recherche de transitions en conflit [GOD 94, BAU 97] aux plus avancés comme la technique des ensemble

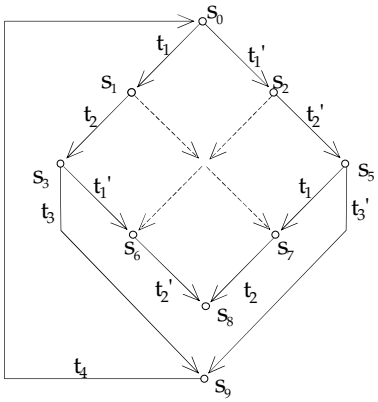


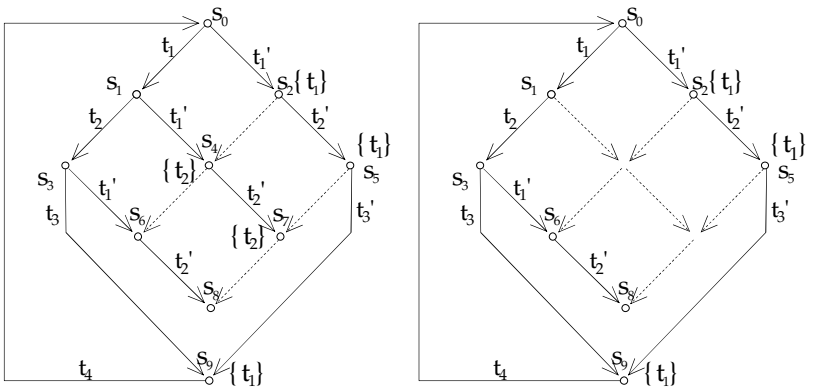
Figure 8.9. *La méthode d'ensembles persistants.*

La méthode la plus simple, dite par conflits, pour calculer un ensemble persistant est proposée dans [GOD 91]. Elle se base sur les constats suivants :

1) l'ensemble obtenu en partant d'une transition franchissable, et en ajoutant récursivement les transitions franchissables et en conflit avec une de l'ensemble est un cas particulier d'ensemble persistant. En cas de transitions non franchissables mais en conflit, on applique le point (2).

2) pour les transitions non-franchissables qui présentent un conflit, on peut toujours utiliser par défaut l'ensemble des transitions franchissables.

Ainsi, on est assuré que toutes les transitions de l'ensemble persistant retenu sont franchissables et n'ont pas de conflit celles qui ne sont pas retenues. Souvent, cette technique produit des ensembles persistants plus gros que la méthode des *stubborn set*, car l'application du cas (2) induit l'absence de réductions pour cet état. Elle peut cependant dans le meilleur des cas être aussi efficace, comme pour l'exemple de la figure 8.9. Ici, l'application du point (1) ne force jamais à appliquer le point (2), et les ensembles persistants construits sont les mêmes qu'avec la méthode des *stubborn set*. L'approche par *stubborn set* propose dans le cas (2) pour les transitions



(a) Méthode de base

(b) Combinaison avec la technique des ensembles rétifs

Figure 8.10. *Espace réduit par les ensembles en sommeil (sleep sets)*

non-franchissables, de trouver une des places dont le marquage est déficitaire (insuffisant pour satisfaire les préconditions de la transition) et aussi sélectionner les transitions qui peuvent y verser des jetons. Des approches complémentaires pour limiter le nombres de transitions de l'ensemble persistant ont également été proposées dans [BAU 97], à l'aide de priorités.

8.4.3. *Recherche de Sleep set*

La méthode des ensembles « en sommeil », ou *sleep set*, introduite dans [GOD 94], exploite l'absence de conflit effectif entre franchissements d'évènements. Elle vise à éviter d'atteindre certains états plusieurs fois par des chemins différents, en prévoyant l'effet diamant dû à l'entrelacement d'évènements indépendants. Au contraire d'un *stubborn set*, le *sleep set* S associé à un état s (noté (s, S)) représente un ensemble de transitions franchissables qu'il est inutile de franchir, car leur état cible est déjà atteint par un autre chemin. Cette technique réduit donc le nombre d'arcs à explorer du graphe d'accessibilité, mais pas le nombre d'états.

Les *sleep set* sont construits à la volée durant la construction du graphe, en initialisant le *sleep set* S_0 de l'état initial à l'ensemble vide. Dans la construction, on considère l'ensemble ordonné T des transitions franchissables depuis un état (s, S). Pour chaque transition t de T, on construit le *sleep set* de l'état atteint comme le *sleep set* de s, augmenté des transitions qui précèdent t dans T, et purgé des transitions en conflit effectif avec t dans l'état source s.

Le graphe de la figure 8.10(a) présente les réductions obtenues à l'aide de cette méthode. Comme le *sleep set* de l'état initial est vide par définition, t_1 et t'_1 sont franchies comme d'habitude. Si l'on tire t_1 en premier, l'état $p_2 + p'_1$ est obtenu d'abord, puis $p_1 + p'_2$ par franchissement de t'_1. Le *sleep set* de $p_2 + p'_1$ reste vide, mais celui de $p_1 + p'_2$ est $\{t_1\}$ car t_1 et t'_1 ne sont pas en conflit. Depuis $p_2 + p'_1$, toutes les transitions franchissables sont donc tirées, atteignant entre autre l'état $p_2 + p'_2$. Depuis $p_1 + p'_2$ au contraire, vu son *sleep set* $\{t_1\}$, seule t'_2 sera franchie. Ainsi l'état $p_2 + p'_2$ ne sera atteint qu'une fois depuis l'état initial par la séquence $t_1.t'_1$. De façon générale, à part p_7 atteint deux fois, les états ne sont atteints que par un chemin depuis l'état initial.

L'intérêt de cette méthode vient du fait qu'elle réduit considérablement le nombre d'arcs du graphe d'accessibilité explorés, accélerant ainsi la procédure. Or le calcul du franchissement peut être une opération coûteuse selon les modèles et les autres techniques de *model checking* employées, et la recherche d'un nouvel état dans l'ensemble des états déjà atteints est toujours un point critique pour les performances du *model checking* . Du point de vue espace, le fait que tous les états soient explorés induit le plus souvent un graphe plus volumineux que l'approche par *stubborn set*. C'est le cas sur notre exemple, voir figures 8.9 et 8.10(a). L'inverse est également possible, des ensembles en sommeil pouvant exister grâce à l'absence de conflit *effectif* entre transitions, alors que l'approche par ensembles persistants fournit dans chaque état l'ensemble des transitions franchissables.

La préservation des états du graphe d'accessibilité permet de vérifier l'ensemble des propriétés d'accessibilité, dont les invariants d'états. Toutes les traces sont également représentées, vu que les arcs retirés sont en réalité inutiles, mais certaines traces ne sont pas complètement représentées, comme $[t_1.t'_1.t'_2]$ pour la trace complète $[t_1.t'_1.t'_2.t_2]$. Ainsi, certaines propriétés comme la présence d'états bloquants ne peuvent être vérifiées que si on rappelle que les transitions du *sleep set* pourraient également être tirées. De façon générale, il faut adapter les algorithmes de vérification pour prendre en compte les *sleep set* associés aux états au cours de la procédure de *model checking* .

8.4.4. *Combinaison des approches Sleep et Stubborn Set*

La méthode des *sleep set* peut également être combinée avec l'approche par *stubborn set* : on retire du *stubborn set* un *sleep set* calculé à l'aide de l'algorithme précédent. On réduit ainsi le graphe calculé par *stubborn set*. La figure 8.10(b) montre l'application combinée des deux méthodes. On constate que l'état $p_2 + p'_3$ n'est plus atteint du tout. En effet, depuis l'état initial, l'état $p_1 + p'_3$ est atteint en franchissant $t'_1.t'_2$ et $\{t_1\}$ est *le sleep* set de cet état. Comme son *stubborn set* est limité à $\{t_1, t'_3\}$, seule t'_3 sera effectivement franchie.

Comme pour les ensembles persistants, des propriétés générales peuvent être ainsi vérifiées, en réutilisant le proviso de Godefroid pour résoudre le problème des transitions ignorées. En fait on peut même améliorer l'algorithme en constatant que par définition une transition d'un *sleep set* n'est pas ignorée.

8.5. Approche par symétries

Pour lutter contre l'effet d'explosion de l'espace d'états, l'approche que nous présentons ici consiste à exploiter des symétries du système [CHI 91] : de nombreux systèmes répartis peuvent être vus comme la composition parallèle de n processus au comportement analogue. Par exemple, les processus du réseau représenté en figure 8.1 se comportent de façon identique. Il est alors possible de construire un graphe d'accessibilité réduit, dit *graphe* ou *automate quotient*, dont les nœuds sont des *classes d'équivalence d'états*.

8.5.1. *Graphe quotient*

Le principe de ce graphe consiste à ne pas distinguer les processus les uns des autres. De ce fait, au lieu de construire un nœud pour l'état s_1 où le processus 1 vient d'exécuter sa demande (franchissement de t_1) et, séparément, un pour l'état s_8 où le processus 2 vient d'exécuter sa demande (franchissement de $t_1\prime$) , on construira un seul nœud $\widehat{s}_1$ représentant la classe d'équivalence $\{s_1, s_8\}$ des états où l'*un des processus* vient d'exécuter sa demande.

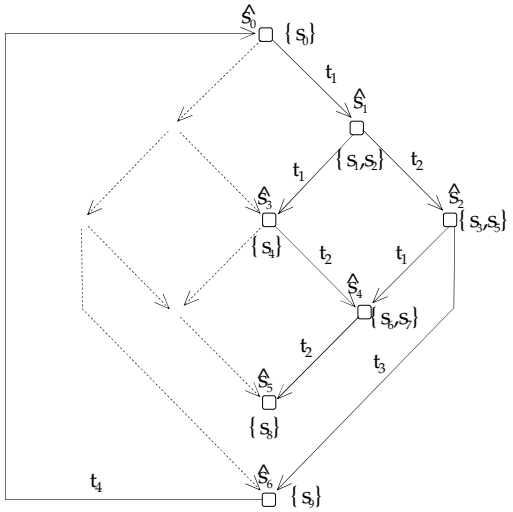


Figure 8.11. *Graphe quotient quand tous les processus sont permutables.*

Processus	États	Classes
2	10	7
4	82	16
6	730	29
8	6562	46
10	59050	67
20	3486784402	232

Tableau 8.3. *Progression comparée de classes d'équivalence et des états*

La figure 8.11 illustre l'effet de cette réduction. Le tableau 8.3 quant à lui montre que l'accroissement du nombre de classes d'équivalence en fonction du nombre de processus est linéaire, ce qui permet d'affirmer que la réduction obtenue est d'ordre exponentiel. Dans chaque nœud du graphe quotient, les états des processus 1 et 2 peuvent être permutés. On place un arc entre deux nœuds $\widehat{s}_i$ et $\widehat{s}_j$ si au moins un des états représentés par $\widehat{s}_i$ permet d'accéder à un des états représentés par $\widehat{s}_j$. Pour formaliser ce raisonnement, nous dirons qu'un graphe quotient est produit *sous* une relation d'équivalence $\mathcal{R}$ qui précise quelles sont les processus permutables dans chaque état.

Le calcul de $\mathcal{R}$ peut être fait directement à partir du graphe d'accessibilité ou bien sur un réseau de Petri vu comme un graphe (voir notamment [SCH 00]), en testant toutes les permutations qui laissent le graphe considéré invariant (on parle d'isomorphisme de graphes), cependant ce travail est d'ordre exponentiel ce qui en limite l'intérêt pour les structures aussi importantes que les graphes d'accessibilité. En revanche, en partant d'une description de haut niveau comme les réseaux de Petri colorés. il est aisé de reconnaître les objets manipulés et de savoir ceux dont les actions les rendent permutables. Dans le réseau de la figure 8.1, l'ensemble des processus est clairement identifié comme un domaine de couleurs permettant de distinguer les jetons dans les places. Comme les actions du réseau (franchissement de transitions) ne nécessite que d'instancier le paramètre x sans contrainte *a priori* sur la couleur utilisée, nous pouvons établir que la relation d'équivalence est $\mathcal{R} : Proc = \{pr_1, pr_2, ...\}$, signifiant que toutes les couleurs sont permutables.

Si par contre, il existe des processus pouvant avoir des comportements singuliers (même un seul), alors ils ne seront pas permutables avec les autres et il faut donc les isoler sous peine d'induire des erreurs dans la vérification. La formalisation correspond à exprimer $\mathcal{R}$ sous la forme d'une partition. Par exemple, si certaines actions du réseau imposaient de distinguer pr_1 et pr_2 (par exemple, en plaçant une priorité de pr_1 sur pr_2 pour l'accès à la section critique au niveau de la transition t_3), la relation d'équivalence deviendrait $\mathcal{R} : Proc = \{pr_1\} \cup \{pr_2\}$. Il s'agit d'une *relation identité*, car aucune permutation d'éléments n'est admise par $\mathcal{R}$. Ceci correspond au pire des cas, où le graphe quotient produit sous $\mathcal{R}$ est égal au graphe d'accessibilité classique :

les classes d'équivalence du graphe quotient ne représentent chacune qu'un unique état.

Il est primordial de réussir à définir une relation d'équivalence aussi permissive que possible (en termes de permutations). Nous utilisons à cette fin un module d'analyse structurelle du modèle [THI 03], qui examine une à une chaque opération du modèle pour connaître les éléments distingués les uns des autres : par exemple une action de type `if (x > 3) doA(); else doB();` sépare déjà l'ensemble des entiers en $N = \{0, 1, 2\} \cup \{3, \ldots n\}$.

Cette technique aurait cependant peu d'intérêt sans une représentation compacte des classes d'équivalence associées aux nœuds du graphe. L'une des approches proposées a été de choisir un représentant pour chaque classe d'équivalence du graphe. En effet, la connaissance de la relation $\mathcal{R}$ permet de reconstruire totalement les classes à partir de leur représentant. Comme pour le graphe d'accessibilité, on peut construire un graphe quotient par induction à partir de l'état initial en utilisant la relation de successeurs. La différence apparaît en ligne (9) de l'algorithme du graphe d'accessibilité : il faut recalculer les classes d'équivalence de tous les états déjà visités afin de vérifier pour un nouvel état calculé s'il est déjà représenté.

Une approche plus directe consiste à définir une représentation canonique pour chaque classe d'équivalence à partir de laquelle on puisse directement calculer les représentations des successeurs [CHI 91]. L'algorithme de construction du graphe quotient mime alors totalement celui du graphe d'accessibilité. Cette représentation canonique est appelée *état symbolique* parce qu'elle introduit des variables pour représenter des groupes d'objets du système pouvant développer les mêmes comportements (en référence à un état du système). Pour un réseau de Petri coloré, cela correspond à avoir certaines couleurs d'une même partie de la partition de $\mathcal{R}$, présentes dans les mêmes places et avec le même nombre d'occurences. En introduisant une série de variables $\widehat{pr}_i$ pour représenter un groupe d'objets de cardinalité $\mid \widehat{pr}_i \mid$, nous pouvons définir une partition des couleurs de $Proc = \{pr_1, pr_2, ...\}$. Un état symbolique correspond à un marquage des places à l'aide de (certaines de) ces variables. La représentation canonique est obtenue simplement en imposant un ordre sur les places, afin de déduire une façon non-ambiguë de nommer les variables symboliques.

Par exemple, la classe d'équivalence $\{s_6, s_7\}$, représentant les états $[P3.pr_1 + P2.pr_2]$ et $[P3.pr_2 + P2.pr_1]$ dans le réseau de Petri symétrique de la figure 8.1, peut être représentée symboliquement en introduisant 2 variables $\widehat{pr}_1$ et $\widehat{pr}_2$ pour distinguer l'existence d'un processus en $P3$ et d'un processus en $P2$. Ces deux variables représentent des ensembles qui prennent leurs valeurs dans $Proc = \{pr_2, pr_1\}$ et qui sont de cardinalité 1 chacune. Ce qui donne l'état $\widehat{s}_4 = [P3.\widehat{pr}_1 + P2.\widehat{pr}_2]$.

La règle de franchissement est aussi dite symbolique. Elle prend en compte un état symbolique pour calculer un successeur symbolique sans faire référence aux classes d'équivalence induites. Elle consiste en 3 étapes :

1) raffiner l'état symbolique considéré pour isoler symboliquement des objets qui vont servir au franchissement (quand cela est possible),

2) opérer le franchissement en exploitant ces objets,

3) agréger les objets qui sont dans le même état (ils marquent les places de façon identique), et obtenir finalement la représentation canonique de la classe d'équivalence atteinte.

Par exemple, à partir de l'état $\widehat{s}_4 = [P_3.\widehat{pr}_1 + P_2.\widehat{pr}_2]$, la transition t_2 est franchissable en utilisant l'objet de $\widehat{pr}_2$ qui marque la place P_2. Du fait du paramètre x en entrée et en sortie de transition, l'objet est juste déplacé vers la place de sortie P_3, ce qui donne le nouveau marquage $[P_3.(\widehat{pr}_1 + \widehat{pr}_2)]$. Comme $\widehat{pr}_1$ et $\widehat{pr}_2$ marquent la même place avec la même occurrence, ils peuvent être regroupés (sous l'appellation $\widehat{pr}_1$ par exemple). Il en résulte que l'état symbolique atteint par le franchissement est $\widehat{s}_5 = [P_3.\widehat{pr}_1]$, où $\widehat{pr}_1$ représente les 2 objets de $Proc$.

Le graphe quotient obtenu à partir d'une relation d'équivalence $\mathcal{R}$ permet de vérifier facilement des propriétés d'accessibilité, en effet tout les états représentés par une classe d'équivalence sont accessibles sur le graphe initial. Cependant, les séquences représentées sur le graphe réduit sont un sur-ensemble des séquences du graphe initial. Par exemple, il existe un chemin de $\widehat{s}_1$ vers $\widehat{s}_2$ en figure 8.11, mais il n'existe pas de chemin de $s_1 \in \widehat{s}_1$ vers $s_5 \in \widehat{s}_2$ en figure 8.2.

8.5.2. *Automates quotient et produit synchronisé symbolique*

La relation d'équivalence utilisée doit être adaptée à la propriété à vérifier, de façon à s'assurer que les ω-mots acceptés par le graphe réduit et par l'automate de la formule, donc potentiellement contre-exemples, existent réellement sur le graphe initial.

Une première approche consiste à raffiner la relation d'équivalence déduite du modèle par une relation d'équivalence déduite de la formule. L'analyse de relation d'équivalence admise par la formule est similaire à celle du modèle : chaque proposition atomique peut donner lieu à la séparation d'éléments *a priori* semblables du modèle. En référence au réseau coloré de la figure 8.1, la relation d'équivalence déduite de la propriété $\varphi : G([P1.pr_1 + P2.pr_2] \implies F[P4])$, est $\mathcal{R}_\varphi : Proc = \{pr_1\} \cup \{pr_2\}$. En effet, les couleurs pr_1 et pr_2 doivent être distinguer entre elles (et optionnellement avec le reste des couleurs de $Proc$ car il faut pouvoir tester sans ambiguïté la véracité des propositions atomiques $P1.pr_1$ et $P2.pr_2$.

Le principal défaut de cette approche est le caractère absolu de la relation d'équivalence construite. En effet, par rapport à la vue d'ensemble rappelée en figure 8.3, nous avons construit séparément des relations d'équivalence $\mathcal{R}_M$ et $\mathcal{R}_{\neg\varphi}$ en fonction du modèle et de la formule. Cependant la construction du produit synchronisé nécessite que ces deux relations soient compatibles. En restant à ce niveau, nous sommes donc contraints d'utiliser l'intersection des deux relations d'équivalence (leur PGCD), au risque de n'avoir plus à exploiter que la permutation identité.

Avec la méthode SSP (*Symbolic Synchronized Product* [AJA 98, BAA 04]), le problème est repoussé car au lieu de chercher à réduire l'espace d'états (l'automate A_M de la figure 8.3), c'est un *produit synchronisé symbolique* qui est calculé directement (à la volée). Dans cette méthode, les arcs de l'automate sont considérés comme autant de contraintes à appliquer sur la relation d'équivalence limitant ainsi les permutations d'objets, mais les contraintes d'un arc de l'automate ne sont prises en compte que si une synchronisation avec lui est entammée. De plus, elles sont relachées quand elles deviennent inutiles pour la validité de la vérification.

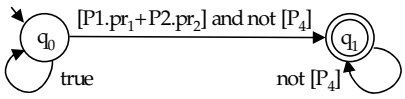


Figure 8.12. *Automate* $A_{\neg G([P1.pr_1+P2.pr_2] \implies [P4])}$

Pour l'automate de la figure 8.12 codant la propriété $\neg G([P1.pr_1 + P2.pr2] \implies [P4])$, les relations d'équivalence pour chacun des arcs sont (de gauche à droite) : $\mathcal{R}a_{0,0} = \mathcal{R}_M : Proc = \{pr_1, pr_2\}$; $\mathcal{R}a_{0,1} : Proc = \{pr_1\} \cup \{pr_2\}$; $\mathcal{R}a_{1,1} = \mathcal{R}_M$.

Le produit synchronisé symbolique est défini comme un graphe orienté, dont chaque nœud est un triplet $\langle \mathcal{R}_l, \widehat{s}, q \rangle$ constitué d'un état q de l'automate de la formule à vérifier, d'une représentation $\widehat{s}$ d'une classe d'états fondée sur une relation d'équivalence $\mathcal{R}_l$ attribuée contextuellement à ce triplet. De plus, chaque état de la classe satisfait les propositions atomiques d'un arc de sortie de q. La constitution des états initiaux de ce produit n'apporte aucune difficulté puisqu'il s'agit de former des triplets $\langle \mathcal{R}_M, \widehat{s}_0, q_0 \rangle$, où l'état $\widehat{s}_0$ représente le singleton $\{s_0\}$, q_0 est un état initial de l'automate. Le calcul des successeurs nécessite un peu plus d'attention. On construira des successeurs à partir d'un nœud $\langle \mathcal{R}_{l1}, \widehat{s}_1, q_1 \rangle$ du produit synchronisé symbolique et d'un arc $q_1 \rightarrow q_2$ de l'automate de Büchi, en suivant les étapes suivantes :

1) prendre en compte les contraintes de permutations imposées par la synchronisation avec l'arc $q_1 \rightarrow q_2$ en calculant la relation d'équivalence restreinte $\mathcal{R}_l = \mathcal{R}_{l1} \cap \mathcal{R}a_{12}$.

2) partitionner la classe d'équivalence de l'état symbolique $\widehat{s}_1$ du fait de cette relation ; en pratique, nous obtenons directement un ensemble d'états symboliques à partir desquels sera testée la satisfaction des propositions atomiques de l'arc $q_1 \rightarrow q_2$ et la franchissabilité des transitions.

3) construire un nœud successeur $\langle \mathcal{R}_{l2}, \widehat{s}_2, q_2 \rangle$, pour chaque état symbolique $\widehat{s}_2$ atteint, en notant que $\mathcal{R}_{l2}$ peut être une relation au moins aussi permissive (en termes de permutations) que $\mathcal{R}_l$; autrement dit $\mathcal{R}_M \subseteq \mathcal{R}_{l2} \subseteq \mathcal{R}_l$. Ce dernier point est fondamental, il consiste à autoriser pour les calculs de successeurs suivants toutes les nouvelles permutations rendues possibles par l'état symbolique atteint. En effet, des objets jusqu'à lors non permutables peuvent par leur nouvel état être identiques à tout point de vue et peuvent donc être regroupés dans la même classe d'équivalence sans perte d'information.

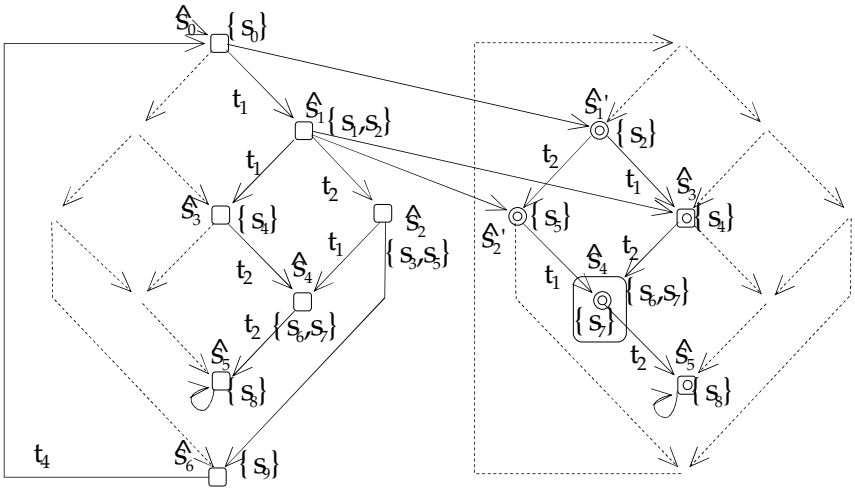


Figure 8.13. *Produit synchronisé symbolique avec l'automate* $A_{\neg G([P1.pr_1 + P2.pr2] \implies [P4])}$

La figure 8.13 exhibe de façon synthétique le produit synchronisé symbolique obtenu à partir de l'automate invalidant la propriété φ : $G([P1.pr_1 + P2.pr2] \implies [P4])$. Là encore, les nœuds de la partie gauche sont synchronisés avec l'état q_0 et ceux de la partie droite avec l'état q_1. De plus, les nœuds à représentation carrée correspondent à la relation d'équivalence R_M : $Proc = \{pr_1, pr2\}$, tandis que ceux à représentation ronde correspondent à une relation d'équivalence restreinte, ici la relation identité.

Cette figure appelle plusieurs remarques : - L'arc de l'automate bouclant sur q_0 n'impose pas de contrainte sur les permutations autorisées, c'est pourquoi la transition $\langle \mathcal{R}_M, \widehat{s}_0, q_0\rangle \rightarrow \langle \mathcal{R}_M, \widehat{s}_1, q_0\rangle$ est définie par un simple franchissement symbolique suivant la relation $\mathcal{R}_M$. - Les successeurs de $\langle \mathcal{R}_M, \widehat{s}_1, q_0\rangle$ sont définis sur une relation restreinte à l'identité du fait de la synchronisation avec l'arc $q_0 \rightarrow q_1$. La classe de l'état symbolique $\widehat{s}_1$ est partitionnée en deux classes singletons, $\widehat{s}'_1 = \{s_1 : [P1.pr_2 + P2.pr_1]\}$ et $\widehat{s}''_1 = \{s_2 : [P1.pr_1 + P2.pr_2]\}$. Seule la seconde dont l'état satisfait les propositions de l'arc aura ses deux successeurs calculés, atteignant ainsi les classes singletons $\widehat{s}'_2 = \{s_5 : [P1.pr_1 + P3.pr_2]\}$ et $\widehat{s}_3 = \{s_4 : [P2.(pr_1 + pr_2)]\}$. - A partir du nœud $\langle \mathcal{R}_I, \widehat{s}'_2, q_1\rangle$, les chemins jusqu'à s_8 (singleton représenté par $\widehat{s}_5$), peuvent être construits dans le produit synchronisé symbolique en adoptant la relation d'équivalence $\mathcal{R}_I$ pour la construction des nœuds. On observera toutefois qu'à partir du nœud $\langle \mathcal{R}_M, \widehat{s}_3, q_1\rangle$, les chemins jusqu'à s_8 sont construits avec la relation d'équivalence $\mathcal{R}_M$. En effet, dans l'état $s_4 = [P2.(pr_1 + pr_2)]$, singleton représenté par $\widehat{s}_3$, les couleurs pr_1 et pr_2 marquent la même place $P2$. La permutation des ces couleurs ne modifie pas le marquage du réseau, c'est pourquoi, la relation d'équivalence $\mathcal{R}_I$ peut être élargie à $\mathcal{R}_M$ à partir de ce moment. - Il y a toutefois une contrepartie puisque le même état pourrait être défini dans plusieurs nœuds du produit synchronisé symbolique, munis de relations d'équivalence différentes. C'est le cas de l'état s_7 représenté par les successeurs respectifs de $\langle \mathcal{R}_I, \widehat{s}'_2, q_1\rangle$ et $\langle \mathcal{R}_M, \widehat{s}_3, q_1\rangle$. Fort heureusement l'inclusion d'une classe d'équivalence dans une autre peut être testé symboliquement, notamment pour éviter les redondances de parcours dans le travail de vérification.

Chaque séquence du produit synchronisé classique est bien représentée par un chemin du produit synchronisé symbolique. Ceci inclut *a fortiori* les séquences acceptées et donc invalidantes. Cela permet d'exploiter directement le produit synchronisé symbolique pour la phase de vérification (*emptiness-check*). Ainsi, la séquence acceptée $\langle \mathcal{R}_M, \widehat{s}_0, q_0\rangle\langle \mathcal{R}_M, \widehat{s}_1, q_0\rangle\langle \mathcal{R}_I, \widehat{s}_3, q_1\rangle\langle \mathcal{R}_M, \widehat{s}_4, q_1\rangle(\langle \mathcal{R}_M, \widehat{s}_5, q_1\rangle)^\infty$ représente une ensemble de séquences invalidantes dont $s_0 s_1 s_4 s_6 (s_8)^\infty$.

On remarquera que la taille du produit synchronisé symbolique de l'exemple est proche de celle du produit synchronisé avec le graphe quotient (construit en toute abstraction de la propriété). La méthode SSP est efficace tout en étant juste. De plus, la construction qui procéde par induction sur un calcul de successeurs, peut s'arrêter au plus vite, dès qu'une séquence invalidant la propriété a été exhibée.

8.6. Conclusion

Le *model checking* est une méthode formelle automatisable qui permet d'assurer la préservation de propriétés comportementales. Les moteurs de vérification proposés sont dès lors suffisamment génériques pour être adaptés à de nombreux langages de spécification de systèmes (par exemple *SPOT* pour LTL, http ://spot.lip6.fr/). Il existe

deux obstacles principaux à la mise en application de telles techniques dans le cadre de méthodologies de développement industrielles :

1) le passage à l'échelle sur des spécifications de taille industrielle.

2) l'intégration des formalismes utilisé pour le *model checking* dans les cycles de développement existants.

Pour le premier point comme on l'a vu, l'explosion combinatoire de l'espace d'états est un facteur limitant pour ces méthodes, même pour des modèles de taille limitée. Les techniques présentées dans ce chapitre offrent des solutions partielles à ce problème et permettent déjà de traiter des spécifications de tailles importantes. On peut en pratique combiner ces techniques en vue d'une meilleure efficacité car elles attaquent chacune l'espace d'états sur un point de vue différent (voir *SPIN*, `http://spinroot.com/spin/whatispin.html` ou *PROD* et MARIA, `http://www.tcs.hut.fi/Software/prod/`). Certaines techniques comme les symétries se prêtent de plus aisément à l'analyse de problèmes plus larges concernant à la fois la vérification et les études de performances quantitatives (voir notamment *GreatSPN*, `http://www.di.unito.it/~greatspn`).

Ces approches nécessitent toutefois des pré-processeurs d'analyse qui déterminent si le modèle est de taille finie, ce qui est déjà tout à fait éprouvé sur des modèles adaptés comme les réseaux de Petri (voir le chapitre 7) ou des langages de bas niveau (hardware) mais qui s'avère encore difficile à réaliser à partir de langages de programmation ou de spécification (trop) expressifs. Nous verrons dans le chapitre suivant comment ramener la description de certains systèmes infinis à des modèles à structures finies, de façon à réaliser leur vérification (voir le chapitre 9).

Pour ce qui est du deuxième point, le problème est que le coût de formation des ingénieurs à ces méthodes et l'investissement nécessaire pour développer des modèles de qualité est jugé trop élevé par rapport au gain de qualité qui résulte de l'utilisation de méthodes formelles dans la majorité des domaines applicatifs.

De ce constat, la vérification de systèmes logiciels tend actuellement à mettre en place des passerelles depuis des langages de spécification informels (UML en particulier) vers des modèles formels (réseaux de Petri...) sur lesquels les techniques de vérification introduites dans ce chapitre puissent être exploitées, même partiellement. Une telle approche est en cohérence avec la proposition tout à fait récente de l'OMG, dite *Model Driven Architecture* (MDA) de l'OMG.

Une autre solution consiste à approcher la vérification directement du produit, c'est-à-dire du programme source lui-même. Plusieurs travaux suivent cette direction, soit pour proposer un langage de programmation restreint mais traduisible dans un langage de spécification vérifiable (par exemple *VERISOFT* pour un C réparti, *QASAR* pour ADA) soit pour proposer un langage complet pour lequel un outillage

d'aide à l'abstraction est proposé (par exemple *SPIN* pour le C de la norme ANSI, ou *JPF* pour Java). Si ces outils passent mal à l'échelle aujourd'hui, nous voyons de façon évidente l'intérêt d'obtenir un compilateur assurant non-seulement la cohérence syntaxique mais aussi l'assurance d'un « bon comportement » [HOA 03].

8.7. Bibliographie

[AJA 98] AJAMI K., HADDAD S., ILIÉ J.-M., « Exploiting Symmetry in Linear Temporal Model Checking : one Step Beyond », *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), part of Theory and practice of Software (ETAPS'98)*, vol. 1384 de *LNCS*, Lisbon, Portugal, Springer-Verlag, p. 52–67, avril 1998.

[AKE 78] AKERS B., « Binary decision diagrams », vol. 27 de *IEEE Transaction on Computers*, p. 509-516, 1978.

[BAA 04] BAARIR S., HADDAD S., ILIÉ J.-M., « Exploiting Partial Symmetries in Well-formed nets for the Reachability and the linear Time Model Checking Problems », *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, p. 223-228, septembre 2004.

[BAH 93] BAHAR R. I., FROHM E. A., GAONA C. M., HACHTEL G. D., MACII E., PARDO A., SOMENZI F., « Algebraic decision diagrams and their applications », *ICCAD '93 : Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, Los Alamitos, CA, USA, IEEE Computer Society Press, p. 188–191, 1993.

[BAU 97] BAUSE F., « Analysis of Petri Nets with a Dynamic Priority Method », *Proc. Int. Conf. on Applications and Theory of Petri nets*, vol. 1248 de *LNCS*, Toulouse, France, Springer Verlag, p. 215–234, juin 1997.

[BRY 86] BRYANT R., « Graph-Based Algorithms for Boolean Function Manipulation », *IEEE Transactions on Computers*, vol. 35, n°8, p. 677–691, August 1986.

[BUR 92] BURCH J., CLARKE E., MCMILLAN K., « Symbolic Model Checking : 10^{20} States and Beyond », *Information and Computation (Special issue for best papers from LICS90)*, vol. 98, n°2, p. 153–181, 1992.

[CHI 91] CHIOLA G., DUTHEILLET C., FRANCESCHINIS G., HADDAD S., « On well-formed coloured nets and their symbolic reachability graph », JENSEN K., ROZENBERG G., Eds., *Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application.*, Springer-Verlag, 1991.

[CIA 03] CIARDO G., MARMORSTEIN R., SIMINICEANU. R., « Saturation unbound », GARAVEL H., HATCLIFF J., Eds., *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, Warsaw, Poland, Springer-Verlag LNCS 2619, p. 379–393, April 2003.

[COU 92] COURCOUBETIS C., VARDI M., WOLPER P., YANNAKAKIS M., « Memory-Efficient Algorithms for the Verification of Temporal Properties », *Formal Methods in System Design*, vol. 1, p. 275–288, 1992.

[COU 96] COUVREUR J., POITRENAUD D., « Model Checking Based on Occurrence Net Graph », GOTZHEIN R., BREDEREKE J., Eds., *Proc. Formal Description Techniques IX, Theory, Application and Tools, FORTE/PSTV'96*, Kaiserslautern, Germany, Chapman and Hall, p. 380–395, octobre 1996.

[COU 02] COUVREUR J.-M., ENCRENAZ E., PAVIOT-ADET E., POITRENAUD D., WACRENIER P.-A., « Data Decision Diagrams for Petri Net Analysis. », *Proc. of ICATPN'2002*, vol. 2360 de *LNCS*, Springer Verlag, p. 101–120, juin 2002.

[COU 05] COUVREUR J.-M., DURET-LUTZ A., POITRENAUD D., « On-the-Fly Emptiness Checks for Generalized Büchi Automata », GODEFROID P., Ed., *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, vol. 3639 de *LNCS*, Springer-Verlag, p. 143–158, août 2005.

[E.M 94] E.M. CLARKE, O. GRUMBERG, K. HAMAGUCHI, « Another Look at LTL Model Checking », DAVID L. DILL, Ed., *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, vol. 818, Standford, California, USA, Springer-Verlag, p. 415–427, 1994.

[ESP 92] ESPARZA J., Model Checking using Net Unfoldings, Rapport n˚14/92, Hildesheimer Informatikfachbericht, 1992.

[ESP 93] ESPARZA J., « Model Checking using Net Unfoldings », GAUDEL M., JOUANNAUD J., Eds., *Proc. TAPSOFT'93*, vol. 668 de *LNCS*, Springer Verlag, p. 613–628, 1993.

[FUJ 97] FUJITA M., MCGEER P. C., YANG J. C.-Y., « Multi-Terminal Binary Decision Diagrams : An Efficient Data Structure for Matrix Representation. », *Formal Methods in System Design*, vol. 10, n˚2/3, p. 149-169, 1997.

[GIA 02] GIANNAKOPOULOU D., LERDA F., « From States to Transitions : Improving Translation of LTL Formulae to Büchi Automata », PELED D., VARDI M., Eds., *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, vol. 2529 de *LNCS*, Houston, Texas, Springer-Verlag, p. 308–326, novembre 2002.

[GOD 91] GODEFROID P., WOLPER P., « A Partial Approach to Model Checking », *Proc. 6th IEEE Symposium on Logic in Computer Science*, Amsterdam, Holland, p. 406–415, juillet 1991.

[GOD 92] GODEFROID P., WOLPER P., « Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties », *Proceedings of the 3rd International Workshop on Computer Aided Verification*, Springer-Verlag, p. 332–342, 1992.

[GOD 94] GODEFROID P., Partial-Order Methods for the Verification of Concurrent Systems, an Approach to the State-Explosion Problem, Thèse de Doctorat, Université de Liège, 1994, also in volume 1032 of LNCS, Springer Verlag, 1996.

[GOD 96] GODEFROID P., *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., 1996.

[HAD 04] HADDAD S., ILIÉ J.-M., KLAI K., « Design and Evaluation of a Symbolic and Abstraction-based Model checker », *2nd International Symposium on Automated Technology for Verification and Analysis ATVA'04*, Springer Verlag, December 2004.

[HER 03] HERMANNS H., KWIATKOWSKA M., NORMAN G., PARKER D., SIEGLE M., « On the use of MTBDDs for Performability Analysis and Verification of Stochastic Systems », *Journal of Logic and Algebraic Programming : Special Issue on Probabilistic Techniques for the Design and Analysis of Systems*, vol. 56, n°1-2, p. 23–67, 2003.

[HOA 03] HOARE T., « The verifying compiler : A grand challenge for computing research », *J. ACM*, vol. 50, n°1, p. 63–69, ACM Press, 2003.

[HOL 92] HOLZMANN G., GODEFROID P., PIROTIN D., « Coverage Preserving Reduction Strategies for the Reachability Analysis », LINN R., UYAR M., Eds., *Proc. Protocol Specification, Testing and Verification XII*, Florida, USA, IFIP, North-Holland, p. 349–364, juin 1992.

[HOL 97] HOLZMANN G., « State Compression in Spin », *Proc. Third Spin Workshop*, Twente University, The Netherlands, April 1997.

[J.R 91] J.R. BURCH, E.M. CLARKE, D.E. LONG, « Symbolic Model Checking with Partitioned Transition Relations », A. HALAAS, P.B. DENYER, Eds., *International Conference on Very Large Scale Integration*, Edinburgh, Scotland, North-Holland, p. 49–58, 1991.

[KES 98a] KESTEN Y., PNUELI A., « Modularization and Abstraction : The Keys to Practical Formal Verification », *LNCS*, vol. 1450, p. 54–70, 1998.

[KES 98b] KESTEN Y., PNUELI A., ON RAVIV L., « Algorithmic Verification of Linear Temporal Logic Specifications », *LNCS*, vol. 1443, 1998.

[KON 96] KONDRATYEV A., KISHINEVSKY M., TAUBIN A., TEN S., « A Structural Approach for the Analysis of Petri nets by Reduced Unfoldings », BILLINGTON J., REISIG W., Eds., *Proc. 17th Int. Conf. on Applications and Theory of Petri nets*, vol. 1091 de *LNCS*, Osaka, Japan, Springer Verlag, p. 346–365, juin 1996.

[LAK 04] LAKOS C., PETRUCCI L., « Modular analysis of systems composed of semiautonomous subsystems », p. 185-194, IEEE Comp. Soc. Press, juin 2004.

[MAZ 86] MAZURKIEWICZ A., « Trace Theory », BRAUER W., REISIG W., ROZENBERG G., Eds., *Petri Nets : Applications and Relationships to other Models of Concurrency, Advances in Petri Nets, Part II*, vol. 255 de *LNCS*, Bad Honnef, Germany, Springer Verlag, p. 279–324, septembre 1986.

[MCM 95] MCMILLAN K., « Trace Theoretic Verification of Asynchronous circuits using Unfoldings », WOLPER P., Ed., *Proc. 7th Int. Conf. on Computer-Aided Verification*, vol. 939 de *LNCS*, Liège, Belgium, Springer Verlag, p. 180–195, juin 1995.

[MIN 93] ICHI MINATO S., « Zero-suppressed BDDs for set manipulation in combinatorial problems », *DAC '93 : Proceedings of the 30th international conference on Design automation*, New York, NY, USA, ACM Press, p. 272–277, 1993.

[MIN 99] MINER A., CIARDO G., « Efficient Reachability Set Generation and Storage Using Decision Diagrams », *Proc. of ICATPN'99*, vol. 1639 de *LNCS*, Springer Verlag, p. 6–25, 1999.

[NIE 81] NIELSEN M., PLOTKIN G., WINSKEL G., « Petri Nets, Events Structures and Domains, Part I », *Theoretical Computer Science*, vol. 13, n°1, p. 85–108, 1981.

[PEL 94a] PELED D., « Combining Partial Order Reductions with On-the-fly Model-checking », *Proc. 6th Int. Conf. on Computer-Aided Verification*, vol. 818 de *LNCS*, Stanford, USA, Springer Verlag, p. 377–390, juin 1994.

[PEL 94b] PELED D., PNUELI A., « Proving Partial-order Properties », *Theoretical Computer Science*, vol. 126, p. 143–182, 1994.

[PEL 94c] PELED D., « Combining Partial Order Reductions with On-the-fly Model-Checking », *Proceedings of the 6th International Conference on Computer Aided Verification*, Springer-Verlag, p. 377–390, 1994.

[PEL 96] PELED D., WILKE T., WOLPER P., « An Algorithmic Approach for Checking Closure Properties of ω-regular Languages », MONTANARI U., SASSONE V., Eds., *Proc. 7th Int. Conf. on Concurrency Theory*, vol. 1119 de *LNCS*, Pisa, Italy, Springer Verlag, p. 596–626, août 1996.

[RIB 02] RIBET P.-O., VERNADAT F., BERTHOMIEU B., « On Combining the Persistent Sets Method with the Covering Steps Graph Method », *FORTE '02 : Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, London, UK, Springer-Verlag, p. 344–359, 2002.

[RUD 93] RUDELL R., « Dynamic variable ordering for ordered binary decision diagrams », *ICCAD '93 : Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, Los Alamitos, CA, USA, IEEE Computer Society Press, p. 42–47, 1993.

[SCH 00] SCHMIDT K., « LoLA : A Low Level Analyser. », NIELSEN, M., SIMPSON, D., Eds., *LNCS : 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, vol. 1825 de *LNCS*, Springer-Verlag, p. 465–474, 2000.

[SRI 90] SRINIVASAN A., KAM T., MALIK S., BRAYTON R. K., « Algorithms for Discrete Function Manipulation. », *ICCAD*, p. 92-95, 1990.

[STE 96] STERN U., DILL D., « A New Scheme for Memory-efficient Probabilistic Verification », GOTZHEIN R., BREDEREKE R., Eds., *Proc. Formal Description Techniques IX, Theory, Application and Tools*, Kaiserslautern, Germany, Chapman Hall, p. 333–348, octobre 1996.

[TAN 93] TANI S., HAMAGUCHI K., YAJIMA S., « The Complexity of the Optimal Variable Ordering Problems of Shared Binary Decision Diagrams », *ISAAC : 4th International Symposium on Algorithms And Computation (formerly SIGAL International Symposium on Algorithms)*, ACM/IPSJ/IEICE, 1993.

[THI 03] THIERRY-MIEG Y., DUTHEILLET C., MOUNIER I., « Automatic Symmetry Detection in Well-Formed Nets », VAN DER AALST W., BEST E., Eds., *Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN'03)*, vol. 2679 de *LNCS*, Eindhoven, The Netherlands, Springer Verlag, p. 82–101, juin 2003.

[VAL 89] VALMARI A., « Stubborn sets for reduced state space generation », *Proceedings of ATPN'89*, Springer Verlag, LNCS 483, 1989.

[VAL 90] VALMARI A., « A Stubborn Attack on State Explosion », CLARKE E., KURSHAN R., Eds., *Proc. 2th Int. Conf. on Computer-Aided Verification*, vol. 531 de *LNCS*, New

Brunswick, NJ, USA, Springer Verlag, p. 156–165, juin 1990.

[VAL 93] VALMARI A., « On-the-fly Verification with Stubborn Sets », COURCOUBETIS C., Ed., *Proc. 5th Int. Conf. on Computer-Aided Verification*, vol. 697 de *LNCS*, Elounda, Greece, Springer Verlag, p. 397–408, juin 1993.

[VAR 96] VARDI M. Y., « An Automata-Theoretic Approach to Linear Temporal Logic », MOLLER F., BIRTWISTLE G. M., Eds., *Proceedings of the 8th Banff Higher Order Workshop*, vol. 1043 de *LNCS*, Springer-Verlag, p. 238–266, 1996.

[VAR 01] VARDI M. Y., « Branching vs. Linear Time : Final Showdown », *LNCS*, vol. 2031, p. 1–22, 2001.

[VER 96] VERNADAT F., AZÉMA P., MICHEL F., « Covering Step Graphs », BILLINGTON J., REISIG W., Eds., *Proc. 17th Int. Conf. on Applications and Theory of Petri nets*, vol. 1091 de *LNCS*, Osaka, Japan, Springer Verlag, p. 516–535, juin 1996.

[VER 97] VERNADAT F., MICHEL F., « Covering Step Graphs Preserving Failure Semantics », *Proc. 18th Int. Conf. on Applications and Theory of Petri nets*, vol. 1248 de *LNCS*, Toulouse, France, Springer Verlag, p. 253–270, juin 1997.

[WIL 96] WILLEMS B., WOLPER P., « Partial-order Methods for Model Checking : From Linear Time to Branching Time », *Proc. 11th Annual Symposium on Logic in Computer Science*, New Brunswick, NJ, USA, p. 294–303, juillet 1996.

[WIN 87] WINSKEL G., « Event Structures », *Advances in Petri Nets 1986, Part II*, vol. 255 de *LNCS*, Springer Verlag, p. 325–392, 1987.

Chapitre 9

Vérification de systèmes infinis

Ce chapitre est dédié à la vérification de systèmes pouvant atteindre un nombre infini d'états. Dans la section 9.2, nous détaillons les patrons de modélisation conduisant à de tels systèmes. Puis la section 9.3 dresse un panorama (non exhaustif) des techniques de vérification employées. Les deux sections suivantes sont consacrées à une étude plus poussée de deux formalismes particuliers. Dans la section 9.4, une extension stricte des réseaux de Petri – les réseaux de Petri récursifs – ainsi que leurs techniques de vérification sont présentées. Enfin, une algèbre de processus très expressive – le π-calcul – et ses méthodes d'analyse sont étudiées dans la section 9.5.

9.1. Introduction

La méthode de vérification la plus naïve consiste à appliquer la sémantique opérationnelle du formalisme afin de construire un graphe d'accessibilité sur lequel l'analyse sera conduite. Cependant, cette technique est inapplicable dans deux cas de figure. D'une part, lorsque le modèle considéré a potentiellement un nombre infini d'états, cette technique devient une procédure de semi-décision. D'autre part, lorsque le concepteur souhaite effectuer une analyse paramétrée de son modèle (*e.g.* la recherche de la taille minimale d'un tampon pour éviter les interblocages), ceci conduit aussi à une procédure de semi-décision obtenue par énumération potentiellement infinie des instances du système sur lesquelles on applique l'algorithme de vérification (dans notre exemple, en incrémentant la taille du tampon). Ce chapitre a pour objectif de décrire des techniques applicables à ce type de problème par exploitation des spécificités des formalismes considérés.

Chapitre rédigé par Frédéric PESCHANSKI et Denis POITRENAUD.

En pratique, on rencontre fréquemment des systèmes informatiques dont le nombre d'états est potentiellement infini. Afin d'illustrer notre propos, nous introduisons un cas typique de système infini. Il s'agit du protocole de fenêtre « glissante » présent dans la couche TCP [STE 76]. Ce protocole a fait l'objet de différentes modélisations formelles afin de vérifier ses propriétés. La figure 9.1, extraite de [ROC 99], montre son architecture générale.

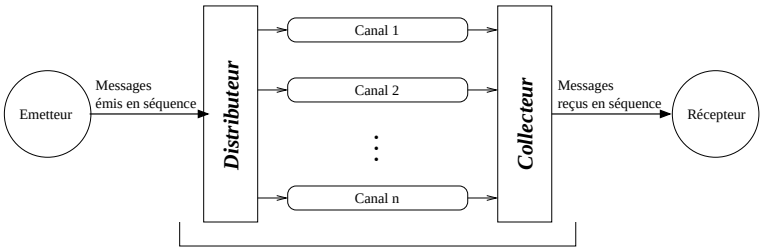


Figure 9.1. *Le protocole de transfert à fenêtres glissantes*

Le but de ce protocole est de transporter des messages en séquence depuis un émetteur vers un récepteur distant sur des canaux non bornés, non ordonnés et non fiables. La préservation de l'ordre entre les messages est garantie par des intervalles de bornes flottantes : *les fenêtres glissantes*. Dans le modèle spécifié dans [ROC 99], du côté de l'émetteur, les messages sont reçus par un *distributeur* qui dispose d'un certain nombre de canaux de communication pour acheminer les messages vers un *collecteur* responsable du réassemblage des séquences en vue de leur livraison au récepteur. On trouve au sein de ce protocole les deux facteurs qui conduisent au caractère infini de la vérification. Tout d'abord, les canaux de communication ne sont pas bornés et peuvent donc recevoir un nombre quelconque de messages. De plus, le système est paramétré par la taille des fenêtres et le nombre de canaux de communication. La propriété que l'on souhaite vérifier est la suivante : la séquence des messages émis et celle des messages reçus sont identiques.

Afin de définir ce problème de vérification de façon plus formelle, il est nécessaire de fixer le formalisme de description du système et la propriété à vérifier ou plus généralement le langage de description de propriétés. Pour chacun de ces problèmes particuliers, différents cas de figure sont possibles.

– Des algorithmes de décision sont établis. Fréquemment, cet algorithme est obtenu au prix d'une restriction du formalisme ou de la nature des propriétés. De plus, la complexité des algorithmes est élevée. Enfin, la preuve de leur correction est élaborée et requiert des raisonnements mathématiques non triviaux.

– Des algorithmes de semi-décision existent. Une procédure de semi-décision se termine nécessairement si la propriété à vérifier est satisfaite mais peut ne pas se terminer dans le cas contraire. Dans la pratique, un test d'arrêt avec réponse incertaine doit

être mis en œuvre. Ils sont plus simples à concevoir que les procédures de décision mais soulèvent le problème de leur *précision* : le ratio de réponses incertaines.

– Les algorithmes de semi-décision conduisent naturellement à une étude théorique. La preuve de l'indécidabilité du problème dans le cas général justifie leur introduction. L'identification de cas particuliers où la procédure se termine fournit des algorithmes de décision pour des sous-classes du formalisme.

– Une approche semi-automatique est aussi possible. À l'aide d'un assistant de preuves, le modélisateur établit une démonstration de sa propriété. D'un façon générale, il doit exprimer des propriétés intermédiaires alors que l'assistant prouve les implications d'une propriété intermédiaire par les précédentes.

Dans ce chapitre, nous traiterons plus spécifiquement d'une extension des réseaux de Petri, les *réseaux de Petri récursifs*, permettant entre autres la création dynamique de processus. Nous illustrerons ainsi une démarche visant à obtenir des algorithmes de décision pour un modèle étendu en se reposant sur l'algorithme correspondant du modèle original. Notre deuxième illustration concernera une variante du π-calcul. Ce formalisme est particulièrement adapté à la modélisation de la mobilité. Il s'agit d'un formalisme basé sur les algèbres de processus dont les constructeurs le dotent d'un pouvoir d'expression équivalent à celui des machines de Turing. Par conséquent, la vérification de tels modèles conduit soit à des procédures de semi-décision, soit à des vérifications semi-automatiques.

9.2. Sources de l'infini dans les formalismes de description

Dans cette section, nous détaillons les *patrons* de systèmes à évènements discrets qui entrainent leur caractère infini. Suivant la démarche adoptée lors de l'introduction, nous distinguons les systèmes à nombre d'états infini et les systèmes paramétrés.

9.2.1. *Systèmes à nombre d'états infini*

Récursion et parallélisme. Dans les algèbres de processus, la *récursion* seule ne conduit pas à des systèmes infinis. Ainsi une algébre dotée uniquement des opérateurs préfixe, choix et récursivité est équivalente au formalisme des automates finis. Par contre, l'adjonction de l'opérateur parallèle conduit à une situation radicalement différente. Ainsi spécifier une création dynamique d'objets par ce type d'algèbre s'effectue de façon naturelle. Par voie de conséquence, ces systèmes sont potentiellement infinis. Nous illustrons différentes sortes d'objets pouvant être créés dynamiquement.

– **Envoi de messages.** Tout canal non borné reçoit sans contrôle d'admission tout message produit.

– **Création dynamique de processus.** Par exemple, un serveur réentrant accepte à tout moment une requête de service et crée un processus dédié au traitement de la requête.

– **Évolution dynamique des systèmes.** Par exemple, un message reçu représente un processus en cours de fonctionnement. Dans un réseau mobile, les interconnexions entre processus sont modifiées dynamiquement.

9.2.2. *Systèmes paramétrés*

Dimensionnement du système. Lors de la phase de spécification d'un système informatique, le concepteur ne souhaite pas fixer les quantités de ressources qui lui seront allouées mais désire les déterminer lors de la phase d'évaluation de performances. Ceci implique que les propriétés fonctionnelles du système doivent être étudiées indépendamment de la valeur de ces quantités.

Systèmes ouverts. La conception objet d'une application conduit naturellement à un système ouvert : les méthodes d'un objet sont invoquées par son environnement qui n'est pas connu lors de la conception de la classe associée. On cherche alors à prouver que, sous certaines hypothèses du comportement de l'environnement, les propriétés attendues de l'objet sont satisfaites. Le domaine du paramètre est ici l'ensemble des environnements en nombre potentiellement infini.

9.3. Quelques schémas usuels de vérification

Devant la diversité des techniques employées pour la vérification de systèmes infinis, il est impossible d'en dresser un panorama exhaustif. Cette section a donc pour objectif de mettre l'accent sur des techniques génériques et de les illustrer par un exemple significatif de vérification. Les références indiquées dans cette section devraient permettre au lecteur d'approfondir sa compréhension des techniques.

Techniques structurelles. Les techniques structurelles présentées dans le chapitre 7 ont été essentiellement développées dans le cadre des réseaux de Petri et s'appliquent donc aux systèmes infinis. Par exemple, les invariants linéaires (ou flots) positifs peuvent être employés pour prouver que le système est fini. En effet, si chaque place d'un réseau de Petri intervient dans un des flots du réseau, cela nous assure que son graphe d'accessibilité est fini. Nous présentons dans la section 9.4.2 une adaptation du calcul d'invariants linéaires pour les réseaux de Petri récursifs.

Représentation symbolique d'ensembles infinis d'états. La construction du graphe de couverture [FIN 93] d'un réseau de Petri permet entre autres de décider le caractère borné de chaque place et le problème de couverture (*i.e.* l'accessiblité d'un marquage

supérieur ou égal à un marquage donné). Cette construction est similaire à celle du graphe d'accessibilité mais dans certaines conditions substitue à un marquage atteint un ensemble infini de marquages. La représentation symbolique consiste à désigner les places pouvant contenir un nombre quelconque de jetons. La construction garantit qu'un sous-ensemble de cet ensemble de marquages est effectivement accessible et que ce sous-ensemble contient des marquages arbitrairement grands sur les places désignées.

La correction de cette construction est principalement due à un argument d'accélération, *i.e.* l'existence de sous-séquences itérées partiellement croissantes. La preuve de terminaison s'établit en exhibant un ordre bien fondé associé aux nœuds d'un chemin infini du graphe de couverture. Ces deux types d'arguments ont été depuis appliqués dans de nombreux contextes (voir [LER 05] pour un cadre d'application assez général englobant plusieurs sous-classes de réseaux de Petri). Lorsque seul le premier argument est applicable, cela conduit à des procédures de semi-décision. L'outil FAST présenté dans [BAR 03] implémente une telle procédure pour la vérification de propriétés d'accessibilité dans des automates augmentés de variables entières (non bornées).

Exploration partielle. Les réseaux de Petri n'autorisent que la vérification de formule LTL évènementielle (des formules dans lesquelles les propriétés atomiques ne font référence qu'aux évènements et non pas aux états visités). Le problème de la vérification de formules CTL ou LTL basée sur les états par un réseau de Petri est indécidable. La procédure de décision associée à la vérification de formules LTL évènementielles est basée sur la détermination de séquences répétitives particulières (d'où le terme d'exploration partielle). Cette détermination est rendue possible parce que l'accessibilité d'un état ou le problème plus général de couverture est un problème décidable pour les réseaux de Petri (voir [ESP 94] pour une synthèse des résultats de décidabilité de différents problèmes de vérification dans le cadre des réseaux de Petri). Il est à noter que les procédures de décision de tels problèmes sont extrêmement complexes (non primitives récursives) et qu'en conséquence, elles ne peuvent être mises directement en pratique.

Les techniques efficaces de *model checking* étudiées au chapitre 8 font toutes l'hypothèse que le système est composé d'un nombre fini d'états. Toutefois, les techniques telles que l'approche par automate pour la vérification de formule LTL ne nécessitent pas de construire au préalable le graphe d'accessibilité. Ce dernier est construit au fur et à mesure de la progression de l'algorithme de vérification et la construction est stoppée dès qu'un contre-exemple peut être exhibé. Cela conduit à une procédure de semi-décision pouvant être appliquée aux systèmes infinis. Pour assurer sa terminaison, la longueur des séquences visitées par l'algorithme est généralement bornée arbitrairement. On parle alors de « *bounded model checking* » (voir [HEL 05] pour une application de ce type de technique à la vérification de formules de logique temporelle à temps linéaire avec opérateurs du passé).

Simulation. Les techniques de simulation sont essentiellement employées dans une démarche de conception par raffinements successifs dans laquelle une modélisation abstraite du système est enrichie pour s'approcher du système concret. Le problème consiste alors à s'assurer que le modélisation enrichie simule bien ce qui était spécifié dans le modèle abstrait. Ici encore, le pouvoir d'expression des modèles est un frein à l'application de ces techniques. Par exemple, il est impossible de décider si un réseau de Petri est simulable par un autre réseau de Petri dans le cas général alors que cela est possible si au moins l'un des deux est fini. La bissimulation (simulation simultanée dans les deux directions) peut servir de base à des vérifications par équivalences comportementales. Dans la section 9.5.3, des procédures de décision de bisimulation seront présentées dans le cadre d'algèbres de processus.

Abstraction. Les techniques d'abstraction permettent de faire porter la vérification sur des modèles simplifiés, mais connectés, du système de départ. Dans [LOI 95], les auteurs donnent les conditions suffisantes sous lesquelles un système infini peut être automatiquement abstrait en un système à nombre fini d'états tout en préservant la satisfaction de formules de logique temporelle à temps arborescent. Du point de vue des systèmes paramétrés, on peut s'intéresser à la spécialisation des données. Le model checker SMV [CIM 02] utilise par exemple une technique de réduction des types de données (*datatype reduction*) permettant de considérer uniquement des cas représentatifs finis pour des types de données infinis. Un découpage binaire simple est de considérer un élément arbitraire i du domaine D ainsi que l'abstraction $D \setminus \{i\}$ (tout sauf i). Une autre technique employée dans SMV est celle de la décomposition temporelle par cas (*temporal case splitting*) qui consiste à considérer, au moment de la vérification, uniquement les éléments effectivement utilisés dans les spécifications. D'autres approche courantes comme le raffinement d'action et l'interprétation abstraite s'apparentent également à des techniques d'abstraction. Dans la sous-section 9.5.4, nous illustrons une autre technique courante, qui consiste en l'abstraction du comportement sous la forme d'un type dit comportemental inféré à partir du système source, et de faire porter la vérification sur le type obtenu.

Composition. Les techniques compositionelles permettent de réduire la compléxité de vérification en déduisant les propriétés globales d'un système par l'analyse et la combinaison de propriétés locales. Une approche largement étudiée, se rapprochant d'une technique d'abstraction, est celle du paradigme prérequis/garantie (*assume/-guarantee*) [ABA 93] dans laquelle chaque composant du système étudié localement est contractualisé par un couple (R, G) où R représente ce que requiert le système (prérequis) et G ce qu'il nécessite (garantie) d'un point de vue global. Un composant C valide (R, G) si l'environnement de C invalide R (ou cesse de satisfaire les prérequis) *avant que* C n'invalide G (ou cesse de fournir les garanties). Une autre technique concerne la composition dite paresseuse [SHA 98] qui propose d'enrichir l'environnement global des propriétés locales au fur et à mesure de la vérification.

Ainsi, le premier composant est vérifié de façon purement locale, le second composant est vérifié localement et vis-à-vis des propriétés obtenues sur le premier et qui possèdent un impact global, et ainsi de suite. Dans toutes ces approches, la problématique de caractérisation précise de l'environnement occupe une place centrale. Nous décrivons une telle caractérisation par des systèmes de transitions étiquetées (LTS) dans la sous-section 9.5.3.

Preuve assistée. Les techniques semi-automatiques de vérification sont actuellement en plein essor. Elles reposent généralement sur des assistants de preuve combinés à des procédures de décision. On peut schématiser le cœur d'un assistant de preuve comme l'opérationalisation d'une logique plus ou moins expressive (souvent des logiques d'ordre supérieur) associée à des principes d'extension conservatifs de la logique (types inductifs, définitions de fonctions, etc.). On encode dans la logique non seulement la sémantique du système considéré mais également les propriétés que l'on souhaite démontrer sur ce système. Une propriété est un but associé à un certain nombre d'hypothèses, l'assistant fournissant alors des tactiques plus ou moins évoluées et automatiques pour atteindre le but à partir des hypothèses et des lemmes et théorèmes précédemment certifiés. Dans [ROC 99] par exemple, l'assistant de preuve Isabelle est utilisé pour vérifier formellement des protocoles de communication à espace d'état infini. Les systèmes sont encodés sous la forme d'automates de transitions étiquetées (cf. sous-section 9.5.3) inférés à partir de la sémantique opérationnelle structurée du formalisme (CCS dans [ROC 99]). Certaines étapes sont opérées de façon automatique, par diverses procédures de décision (e.g. déduction à la Prolog) et les cas difficiles se font de manière assistée, par exemple par l'assertion de lemmes intermédiaires, de raisonnements inductifs ou co-inductifs, etc. Des travaux récents introduisent des tactiques automatiques de type équivalence comportementale et/ou *model checking* directement dans l'assistant de preuve. Une telle approche est discutée dans la sous-section 9.5.4.

9.4. Réseaux de Petri récursifs

Le formalisme des réseaux de Petri récursifs (RdPR) a été introduit pour permettre la modélisation de systèmes dynamiques pour lesquels la création de processus est requise [SEG 95, SEG 96]. Dans ce modèle, un processus est caractérisé par un réseau dans lequel des transitions particulières (dites abstraites) permettent la création de sous-processus et des ensembles de marquages (dits de terminaison) indiquent les états pour lesquels le processus peut se terminer. Tous les processus partagent la même structure de contrôle (c'est-à-dire le même RdPR) mais l'état initial d'un processus dépend de la transition abstraite le créant. En conséquence, l'état d'un processus est simplement caractérisé par un marquage du RdPR. Le parallélisme (l'aspect fondamental capturé par les réseaux de Petri) est autorisé entre le processus appelant et appelé. La terminaison d'un processus fils est signifiée au processus père (par un changement de son marquage). Pour se faire, la relation de paternité (liant appelant et appelé) est

codée dans l'état d'un RdPR. En conséquence, un état d'un RdPR est un arbre de processus dans lequel chaque nœud porte le marquage du processus qu'il représente.

Il a été montré dans différents articles [HAD 99a, HAD 00, HAD 01] que le formalisme des RdPR est une réelle extension des réseaux de Petri en terme de pouvoir d'expression. D'autre part, il a aussi été montré que le problème de l'accessibilité reste décidable. Il est à noter que cette propriété est fondamentale pour pouvoir définir un outil de vérification. Toutefois, dans [HAD 99b], les auteurs ont démontré que la vérification de formules LTL n'est pas un problème décidable pour ce formalisme alors qu'il l'est dans le cadre des réseaux de Petri (à condition de se limiter à des formules évènementielles).

Dans cette section, nous présentons un sous-formalisme des RdPR appelé les réseaux de Petri récursifs séquentiels (RdPRS) [HAD 01]. Dans un RdPRS, le comportement d'un processus créant un processus fils est bloqué jusqu'à la terminaison de ce dernier. En conséquence, le seul parallélisme autorisé ne peut se produire qu'au sein d'un même processus. Ce formalisme, bien que moins expressif que les RdPR, reste une réelle extension des réseaux de Petri mais préserve la décidabilité de tous les problèmes cités ci-dessus.

9.4.1. *Pouvoir d'expression*

Les RdPRS sont introduits informellement au travers de deux exemples simples permettant de comprendre le formalisme et de juger de son pouvoir d'expression. Nous discutons ensuite de différentes voies pour la vérification (comportementale et structurelle).

Interruptions et exceptions. Nous allons illustrer le formalisme des RdPRS en intégrant successivement à un modèle existant un mécanisme d'interruption puis la prise en compte d'exception. Le RdPRS modélisant une abstraction du système original est présenté dans la figure 9.2. Ce RdPRS n'emploie que des éléments communs aux réseaux de Petri. Dans cette abstraction, le système peut réaliser un unique comportement infini où la transition $t_{correct}$ est systématiquement franchie. Il est à noter que tout réseau de Petri peut être interprété comme étant un RdPRS mettant en œuvre un unique processus.

Lors du déclenchement d'une interruption (que celle-ci soit logicielle ou matérielle), le contexte courant du système doit être sauvegardé et le contrôle doit ensuite être dérouté vers la procédure de traitement de l'interruption. Selon son niveau, l'interruption peut éventuellement masquer les autres interruptions durant son traitement. Enfin, une fois la procédure de traitement terminée, le contexte d'exécution du système doit être rechargé.

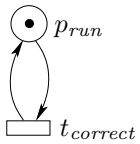


Figure 9.2. *Une abstraction d'un système*

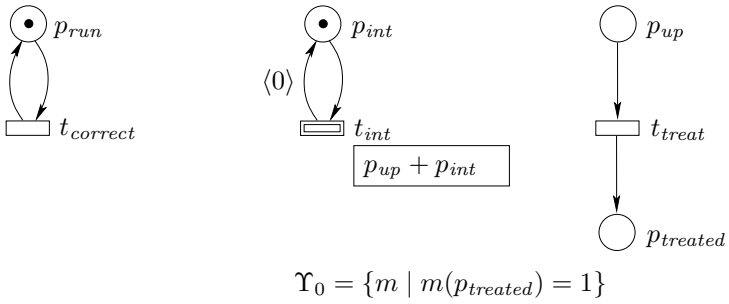


Figure 9.3. *Intégration d'un mécanisme d'interruption*

Le RdPRS présenté dans la figure 9.3 ajoute un mécanisme d'interruption au réseau de la figure 9.2. Les éléments nouveaux par rapport au formalisme des réseaux de Petri sont les suivants :

– La transition t_{int} : cette transition, dite *abstraite* et représentée par un rectangle à bord double, modélise le déclenchement de l'interruption. Ici, nous considérons que l'interruption peut toujours survenir et en conséquence, sa seule place d'entrée (p_{int}) est initialement marquée. Le franchissement de cette transition abstraite crée un nouveau processus (dont la structure de contrôle est le même RdPRS) et stoppe son exécution jusqu'à la terminaison de ce dernier. Le processus créé débute son exécution à partir du marquage initial ($p_{up} + p_{int}$) associé à la transition abstraite et représenté dans le cadre adjacent à t_{int}. Ce marquage est appelé le *marquage de départ* de la transition abstraite. Du point de vue du marquage courant du processus dans lequel la transition abstraite est tirée, ce franchissement consiste à consommer les jetons désignés par la pré condition (ici un jeton de la place p_{int}). Nous verrons ci-dessous que la production des jetons spécifiés par la post condition dépend de la terminaison du nouveau processus.

– L'ensemble Υ_0 : Un ensemble de marquages ordinaires définit des états dans lequel les processus peuvent se terminer. Un tel ensemble est appelé un *ensemble de terminaison* et il doit être spécifié sous la forme d'une représentation effective d'un ensemble semi-linéaire de marquages (dans la présentation originale des réseaux

récursif, la terminaison était spécifiée par des transitions particulières dites terminales – le r présentation donnée ici est plus expressive). Dans notre exemple, l'ensemble Υ_0 est constitué de tous les marquages pour lesquels la place $p_{treated}$ contient exactement un jeton (i.e. $\Upsilon_0 = \{m \mid m(p_{treated}) = 1\}$). Un RdPRS peut être muni de plusieurs ensembles de terminaison (non nécessairement distincts). L'indice associé à chacun de ces ensembles permet de distinguer les différents types de terminaison possibles. Dans notre exemple, un seul type de terminaison est supporté et il est caractérisé par l'indice 0.

– La valuation $\langle 0 \rangle$ portée par l'arc liant la transition t_{int} à la place p_{int} : Lorsqu'un processus se termine (et donc que son marquage courant appartient à un ensemble de terminaison donné), l'état du processus père est modifié selon le type de terminaison. Pour se faire, la post condition d'une transition abstraite est conditionnée par l'indice associé à l'ensemble de terminaison atteint par le fils. Ainsi, dans notre exemple, la terminaison de la procédure de traitement de l'interruption conduit à la production d'un jeton dans la place p_{int} au niveau du processus père. Ceci est spécifié dans le réseau par la valuation $\langle 0 \rangle$ portée par l'arc sortant de la transition abstraite.

En résumé, un pas d'exécution d'un RdPR correspond à un pas du dernier des processus ayant été créés et trois types de pas sont distingués :

– La création dynamique de processus est réalisée par les transitions *abstraites*. Leur franchissement provoque la consommation des jetons de la pré condition, bloque l'exécution de l'appelant et enfin débute l'exécution d'un processus fils à partir du marquage de départ associé à la transition abstraite.

– Les actions élémentaires sont modélisées par le biais de transitions *élémentaires*. Leur franchissement au sein d'un processus modifie le marquage courant de celui-ci de façon similaire au franchissement d'une transition dans un réseau de Petri ordinaire.

– La terminaison d'un processus peut survenir lorsque celui-ci atteint un état appartenant à un ensemble de terminaison. Ce type de pas est appelé une *coupure*. Un tel pas entraîne la terminaison du processus courant et conduit, dans le processus père (et à condition que celui-ci existe), à la production des jetons liés à la post condition de la transition abstraite ayant donné naissance au processus. Cette post condition est spécifiée en fonction de l'ensemble de terminaison atteint.

Un état d'un RdPRS mémorise le marquage de chaque processus. De plus, l'identité de la transition abstraite ayant conduit à la création de chacun (à l'exception du processus initial) est aussi mémorisé. Il est important de noter que le franchissement d'une coupure au sein du processus racine conduit à un état composé d'aucun processus. Cet état particulier est bloquant et il est noté $\perp$.

L'état initial du RdPRS de la figure 9.3 est composé d'un unique processus dans lequel les seules places p_{run} et p_{int} contiennent chacune un jeton. Il est clair que ce

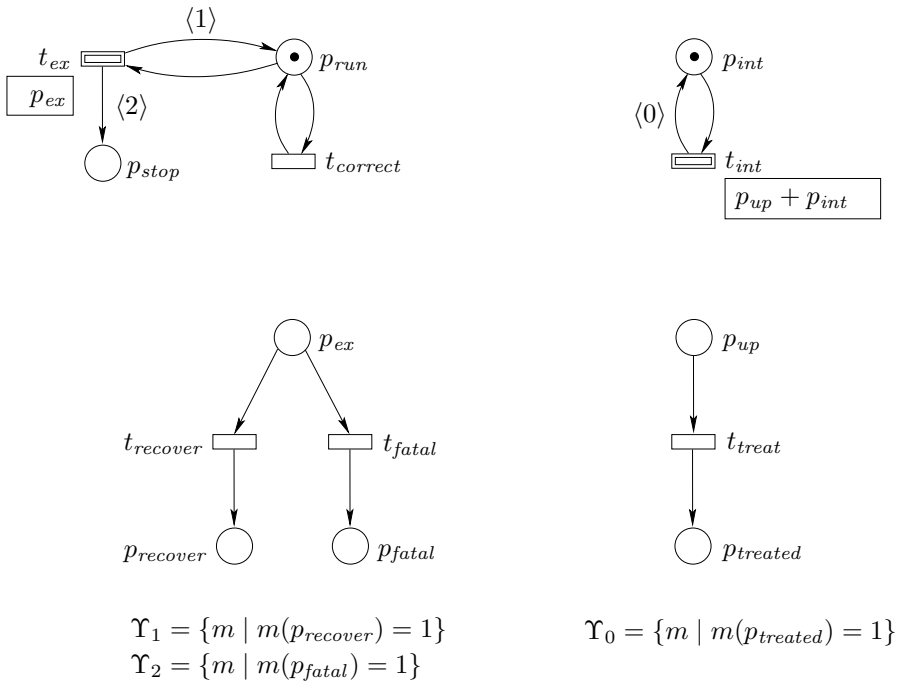


Figure 9.4. *Intégration d'un mécanisme d'exception*

réseau peut atteindre une infinité d'états. En effet, la prise en compte d'une interruption ne masque pas celles pouvant survenir durant la procédure de traitement (dans le marquage de départ associé à t_{int}, la place p_{int} contient un jeton). Nous pouvons remarquer que cette infinité d'états est possible alors que toutes les places sont bornées (elles peuvent contenir au maximum un unique jeton).

L'intégration d'un mécanisme d'exception est illustrée dans le RdPRS de la figure 9.4. Dans cet exemple, une exception ne peut survenir (transition abstraite t_{ex}) que si le processus est en cours d'exécution (i.e. la place p_{run} est marquée). Son traitement peut recouvrer l'erreur (place $p_{recover}$) ou conduire à une situation critique (place p_{fatal}). Les deux ensembles de terminaison Υ_1 et Υ_2 permettent de distinguer ces deux cas et les arcs sortants de la transition abstraite t_{ex} ont été étiquetés en conséquence.

La figure 9.5 présente un extrait du graphe d'accessibilité de ce réseau. Un état du réseau est représenté par la liste des processus le composant, chacun étant étiqueté par son marquage courant. Le nom de la transition abstraite ayant donné naissance à un

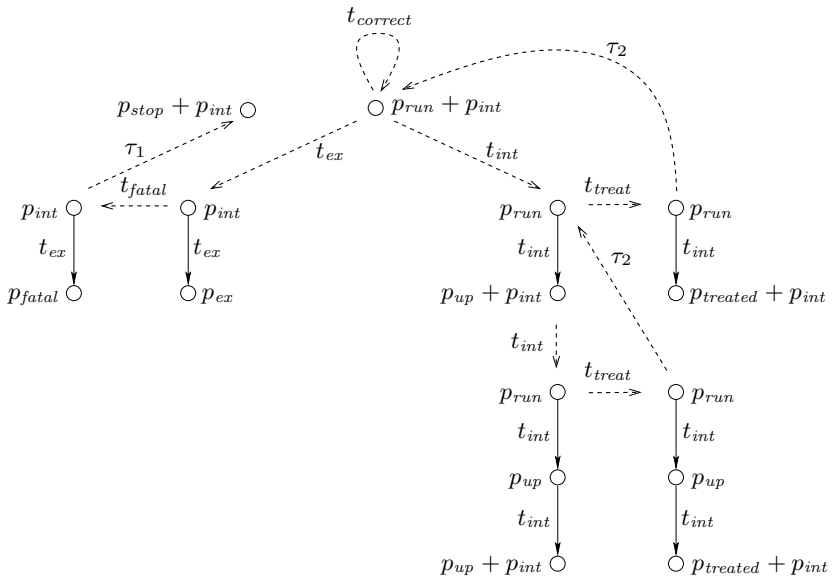
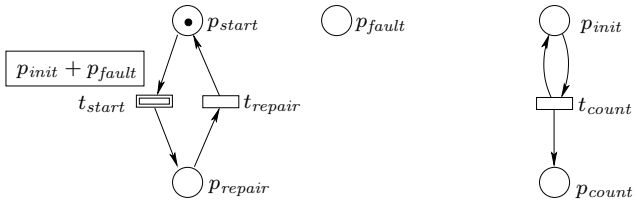


Figure 9.5. *Extrait du graphe d'accessibilité du RdPRS de la figure 9.4*

processus est noté sur l'arc reliant un processus père à son fils. Enfin, le franchissement d'une coupure correspondant à un ensemble de terminaison Υ_i est noté τ_i.

La partie droite de la figure illustre la prise en compte des interruptions et la partie gauche celle des exceptions. Il est à noter qu'une interruption ne peut pas survenir durant le traitement d'une exception. Par contre, si le traitement d'une exception conduit à une erreur fatale, une fois cette erreur répercutée au niveau du processus père, le comportement de celui-ci sera limité à la prise en compte d'interruption (la seule transition franchissable, mais qui n'est pas représentée dans cet extrait, à partir de l'état dans lequel le processus racine a atteint le marquage $p_{stop} + p_{int}$, est t_{int}).

L'exemple de la figure 9.4 montre la capacité des RdPRS à mémoriser implicitement le contexte des processus suspendus alors que ce type de modélisation dans le formalisme des réseaux de Petri requiert une représentation explicite de chacun des contextes. En employant cette capacité, nous avons montré dans [HAD 00] que les RdPRS incluent la famille des langages algébriques. D'autre part, il a été montré que le langage des palindromes (un langage algébrique particulier) ne peuvent pas être reconnu par un réseau de Petri (étiqueté) [JAN 79]. Ainsi, la famille de langages des RdPRS inclut strictement celle des réseaux ordinaires.



$$\Upsilon_0 = \{m \mid m(p_{fault}) > 0\}$$

Figure 9.6. *Modélisation d'un système tolérant au faute*

Reprise sur faute. Pour modéliser un système tolérant aux fautes, un ingénieur commence par modéliser le système nominal puis introduit les comportements fautifs et le mécanisme de réparation associé. Nous nous limitons, une fois encore, à une modélisation très abstraite du système nominal. Cette abstraction est donnée dans la partie droite de la figure 9.6, le système nominal exécute infiniment des instructions (la transition élémentaire t_{count}). Le marquage de la place p_{count} représente le nombre d'instructions ayant été exécutées depuis la création du processus.

Le RdPRS complet est obtenu en ajoutant la partie gauche de la figure 9.6. Son comportement peut être décrit comme suit. Il n'y a que deux états accessibles composés d'un unique processus : l'état initial (noté tr_{start} et où un jeton dans la place p_{start} indique que le système est prêt à être exécuté) et l'état de réparation (noté tr_{repair} et où un jeton dans la place p_{repair} indique que le système est en réparation). À partir de l'état initial, la transition abstraite t_{start} est franchie et l'exécution des instructions est "jouée" par le nouveau processus. Si ce processus meurt par le franchissement d'une coupure (indiquant l'occurrence d'une faute), l'état de réparation est atteint. La place p_{fault} représente la possibilité d'une faute. Comme cette place est toujours marquée lorsque les instructions sont exécutées et due à la définition adéquate de Υ_0, l'occurrence d'une faute est toujours possible. Nous faisons l'hypothèse qu'aucune faute ne peut survenir durant la phase de réparation. En introduisant des places additionnelles et en modifiant Υ_0, nous aurions pu modéliser des occurrences de fautes plus complexes (e.g. conditionnées par l'exécution du système).

Les états du RdPRS de la figure 9.6 sont constitués soit d'un unique processus soit d'un processus père et d'un fils. Toutefois le nombre de marquages pouvant être atteint par le processus fils est infini (la place p_{count} n'est pas bornée). En conséquence, l'état de réparation peut être atteint à partir d'une infinité d'états. Le graphe d'accessibilité de ce réseau est présenté dans la figure 9.7. Les sommets du graphe étant étiquetés tr_i (avec $i \geq 0$) correspondent aux états pour lesquels le système nominal a exécuté i fois la transition t_{count}. Il est clair que l'état tr_{repair} peut être atteint par chacun des états tr_i.

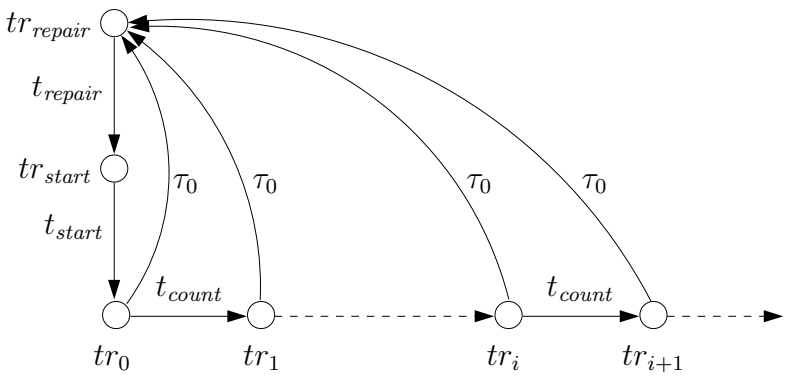


Figure 9.7. *Graphe d'accessibilité (infini) du RdPRS de la figure 9.6*

Ceci signifie que le graphe d'accessibilité d'un RdPRS peut avoir des états ayant un degré entrant infini. Cette capacité n'est partagée ni par les réseaux de Petri, ni par les algèbres de processus. En particulier, les états du graphe d'accessibilité d'un réseau de Petri ont un degré entrant borné par le nombre de transitions du réseau et le fait d'autoriser des transitions à être inobservées dans le graphe ne permet pas de lever cette limite.

Dit autrement, la modélisation d'un système nominal ayant un nombre infini d'états fautifs est impossible avec les réseaux de Petri. Dans le cas restreint où le nombre d'états du système nominal est fini, cette modélisation est théoriquement possible. Toutefois, la modélisation des fautes requiert un nombre proportionnel de transitions par rapport à celui des états accessibles. Ceci conduit à des réseaux complexes et intriqués. Le RdPRS présenté ici ne dépend pas du nombre d'états du système nominal conduisant ainsi à une représentation compacte et modulaire.

9.4.2. *Vérification*

Une caractéristique primordiale des réseaux de Petri est que l'accessibilité d'un état est un problème décidable (même si l'algorithme de décision n'est pas primitif récursif). En effet, beaucoup de propriétés de sûreté (exprimant qu'un mauvais comportement ne doit pas se produire) peuvent être réduite à un problème d'accessibilité. Une procédure de décision peut donc être à la base d'un outil important de vérification.

Il est possible de décider du caractère borné d'un réseau de Petri. Le graphe des marquages accessibles d'un réseau borné est fini et les techniques présentées dans le chapitre 8 peuvent alors être appliquées. D'autre part, l'appartenance d'un mot au langage d'un réseau de Petri étiqueté est aussi un problème décidable. Cette propriété,

appelée récursivité du langage, permet de vérifier qu'un comportement attendu peut effectivement être réalisé par le système. Enfin, la vérification de propriété temporelle est possible mais uniquement dans le cadre restreint de la logique temporelle évènementielle et à temps linéaire (des formules LTL ne faisant référence qu'au transition du système et non pas aux états visités). [ESP 94] présente une large synthèse des résultats de décidabilité pour les réseaux de Petri ordinaires.

L'ensemble de ces problèmes restent décidables dans le cadre des RdPRS. Ainsi même s'ils forment une extension stricte des réseaux de Petri ordinaires, la vérification de nombreuses propriétés comportementales reste possible. Dans cette section, nous nous limiterons à l'étude du problème d'accessibilité.

Un autre point fort des réseaux de Petri est les techniques de vérification structurelle telles que celles présentées dans le chapitre 7. Nous présentons ici une technique d'analyse structurelle visant à la détermination d'invariants linéaires de RdPRS.

Vérification comportementale. Étant donné un RdPRS et son état initial, il s'agit de décider si un état particulier est accessible par franchissements successifs. La procédure de décision présenté (informellement) ici est inspirée de celle (plus complexe) employée pour les RdPR. Le principe général consiste à ramener le problème original à une suite (potentiellement large) de problèmes d'accessibilité dans des réseaux de Petri ordinaires.

La procédure de décision est réalisée en deux étapes :

– La première est indépendante de l'état initial du RdPRS et de celui devant être atteint. Elle consiste à déterminer si un processus démarrant avec le marquage initial associé à une transition abstraite peut atteindre un marquage issu d'un ensemble de terminaison. Une telle transition abstraite est dite « *fermable* » par rapport à cet ensemble de terminaison. Cette détermination doit être réalisée pour chaque transition abstraite face à chaque ensemble de terminaison.

– La seconde étape vise à prédire le devenir des processus composant l'état initial du RdPRS et en conséquence la nature des processus composant l'état visé. En effet, un processus initialement présent peut soit être terminé durant la séquence de franchissement soit perdurer en faisant potentiellement évoluer son marquage. Les processus de l'état destination ne correspondant pas à des processus présents dans l'état initial doivent avoir été créé par la séquence de franchissement. Même si chaque prédiction doit satisfaire un certain nombre de contraintes (par exemple s'il est prédit qu'un processus de l'état destination doit être créé par la séquence alors il doit en être de même pour tous ces processus fils), plusieurs prédictions distinctes sont souvent possibles. La figure 9.8 présente un exemple de prédiction. Les deux processus de l'état initial portant les marquages m_0 et m_1 doivent évoluer vers les marquages respectifs m'_0 et m'_1. Par contre, le processus portant le marquage m_2 doit se terminer durant la séquence de franchissement et en conséquence il en est de même pour tous ses sous

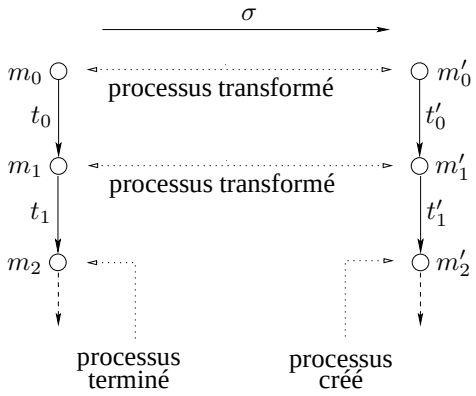


Figure 9.8. *Prédiction du devenir des processus*

processus. Enfin, il est prédit que le processus de l'état destination portant le marquage m'_2 sera créé par la séquence. Il est clair que le nombre de prédictions possibles est borné par le nombre de processus présents dans les deux états.

Toutes les étapes de la procédure de décision sont basées sur le même principe : la construction d'un réseau de Petri ordinaire et d'un problème d'accessibilité équivalent au problème élémentaire devant être décidé. Nous nous focalisons à présent sur la détermination des transitions abstraites « *fermables* ».

Il faut tout d'abord remarquer que décider si un réseau de Petri peut atteindre un marquage appartenant à un ensemble semi-linéaire peut être réduit à un simple problème d'accessibilité en transformant le réseau original.

Une transition abstraite est « *fermable* » si un processus créé par cette transition à la possibilité de réaliser une coupure et donc de se terminer. C'est donc un problème d'accessibilité et si une transition est « *fermable* », elle induit une séquence de franchissement associée à ce problème. Lorsqu'une transition abstraite est franchie au sein d'un processus, l'exécution de ce dernier est bloquée jusqu'à ce que le processus fils créé par la transition abstraite soit terminé. En conséquence, la séquence de franchissement associée ne peut faire intervenir au niveau du processus racine que des transitions abstraites « *fermables* ».

Cela conduit à un calcul itératif de l'ensemble F des transitions « *fermables* ». F_0 est le sous-ensemble d'éléments de F tels que la séquence associée ne contient aucun franchissement de transitions abstraites. Ainsi, lorsque de nouveaux éléments sont ajoutés à F, cela indique que de nouvelles transitions « *fermables* » peuvent être employées.

Les éléments de F_0 sont déterminés en testant si le réseau de Petri ordinaire obtenu en supprimant toutes les transitions abstraites du RdPRS peut atteindre un marquage appartenant à un ensemble de terminaison à partir du marquage de départ de la transition abstraite considérée. Pour le calcul de F_1, les transitions abstraites appartenant à F_0 sont à présent simulées par des transitions élémentaires ayant les mêmes pré et post conditions. Le rôle de ces transitions est de simuler la création d'un sous-processus suivi de sa terminaison.

Le fait qu'un ensemble F_i (avec $i \geq 0$) soit vide nous assure que tous les ensembles F_j (avec $j > i$) seront eux aussi vides. Sachant que l'ensemble des transitions abstraites est fini, ceci démontre la terminaison de la procédure.

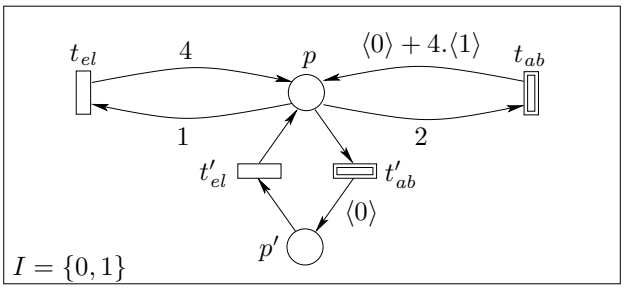
Nous sommes maintenant en position de décrire la procédure de décision du problème d'accessibilité. Soit src l'état initial et $dest$ l'état destination. Si $src = \perp$ (l'état composé d'aucun processus) alors il est suffisant de tester que $dest = \perp$. Faisons maintenant l'hypothèse que $src \neq \perp$ et $dest = \perp$. Ce problème est similaire à celui consistant à déterminer si une transition abstraite est « *fermable* » mis à part que l'état initial n'est plus nécessairement composé d'un seul processus. En conséquence les sous-processus de la racine doivent eux-mêmes être fermés successivement ce qui conduit à un ensemble de problèmes de terminaison.

Enfin, lorsque $src \neq \perp$ et $dest \neq \perp$, le principe consiste à vérifier une à une les différentes prédictions possibles du devenir des processus. La différence principale vient du fait que des processus de l'état destination peuvent être prédits comme étant créés. Ici encore, la décision peut être réduite à un problème d'accessibilité dans un réseau ordinaire.

Vérification structurelle. Parmi les méthodes de vérification structurelle présentées dans le chapitre 7, nous nous focalisons sur le calcul d'invariants linéaires que nous adaptons aux RdPRS. Le calcul présenté ici est adapté de la méthode générale applicable aux RdPR.

La matrice d'incidence W d'un RdPRS est définie comme suit. Les lignes de la matrice sont composées des places et des transitions abstraites. La signification intentionnelle d'une variable indicée par une place est son nombre de jetons alors que l'interprétation d'une variable indicée par une transition abstraite est le nombre de processus créés par son franchissement.

Les colonnes de cette matrice sont composées des transitions et de paires (t, i) où t est une transition abstraite et i est un indice désignant un ensemble de terminaison. Une colonne indicée par une transition représente son franchissement alors qu'une paire (t, i) représente le franchissement, dans un sous processus initié par la transition abstraite t, d'une coupure relative à l'ensemble de terminaison désigné par l'indice i.



(a) Un RdPRS (seules les données utiles sont représentées)

	t_{el}	t'_{el}	t_{ab}	t'_{ab}	$(t_{ab},0)$	$(t_{ab},1)$	$(t'_{ab},0)$	$(t'_{ab},1)$	
p	3	1	-2	-1	1	4	0	0	P
p'	0	-1	0	0	0	0	1	0	
t_{ab}	0	0	1	0	-1	-1	0	0	T_{ab}
t'_{ab}	0	0	0	1	0	0	-1	-1	
	T_{el}		T_{ab}		$T_{ab} \times I$				

(b) Sa matrice W

Figure 9.9. *Un réseau récursif et sa matrice d'incidence W*

Nous notons respectivement T_{el} et T_{ab} l'ensemble des transitions élémentaires et abstraites. La matrice d'incidence est définie par :

– $\forall p \in P, \forall t \in T_{el}, \forall t' \in T_{ab}$,
$W(p,t) = Post(p,t) - Pre(p,t)$ et $W(t',t) = 0$

– $\forall p \in P, \forall t, t' \in T_{ab}$ with $t' \neq t$,
$W(p,t) = -Pre(p,t)$ et $W(t,t) = 1$ et $W(t',t) = 0$

– $\forall p \in P, \forall t, t' \in T_{ab}, \forall i \in I$ with $t' \neq t$,
$W(p,(t,i)) = Post(p,t,i)$ et $W(t,(t,i)) = -1$ et
$W(t',(t,i)) = 0$

Nous rappelons que la post condition d'une transition abstraite est conditionné par l'ensemble de terminaison atteint par le processus fils (d'où la notation $Post(p,t,i)$).

La figure 9.9 illustre la définition de la matrice W. Elle est divisée en six blocs dépendant du type de lignes et de colonnes. Regardons quelques uns des éléments de la ligne correspondant à la place p : la transition élémentaire t_{el} consomme un jeton de p et en produit quatre, en conséquence, l'élément correspondant dans W est 3 ; le

franchissement de la transition abstraite t_{ab} consomme deux jetons et donc l'élément correspondant est -2. Une coupure associée à t_{ab} et l'indice 0 (resp. 1) produit un jeton (resp. quatre jetons). En conséquence, l'élément correspondant dans W est donc 1 (resp. 4). Observons à présent la ligne indicée par la transition abstraite t_{ab} : son franchissement crée un processus initié par t_{ab} et donc l'élément correspondant est 1 alors que le franchissement de chacune des deux coupures correspondant à t_{ab} termine un tel processus conduisant à l'élément -1.

Étant donné un processus pr d'un état tr d'un RdPRS, nous notons $fire(pr)^{tr}$ le vecteur, indicé par T_{ab}, tel que pour toute transition abstraite t, $fire(pr)^{tr}(t)$ est égal à un si un sous processus initié par t a été créé par pr et n'a pas encore été terminé et égal à zéro dans le cas contraire. Nous sommes maintenant en position de justifier le choix de W comme matrice d'incidence.

Soit tr' un état d'un RdPRS accessible à partir d'un état tr via une séquence σ. Faisons l'hypothèse de l'existence d'un processus pr présent dans tr et dans tr' (et donc dans tous les états intermédiaires visités par σ). On note $m(pr)^{tr}$ le marquage du processus pr dans l'état tr. Soit x une solution de $x \cdot W = 0$, alors :

$$x \cdot (m(pr)^{tr'}, fire(pr)^{tr'}) = x \cdot (m(pr)^{tr}, fire(pr)^{tr})$$

Comme pour les réseaux de Petri ordinaires, lorsque tr est l'état initial du réseau, cette équation est appelée un invariant linéaire. Pour obtenir ces invariants, il est possible de calculer une famille génératrice de solutions $\{x_1, \ldots, x_n\}$ de cette équation. Pour un processus pr présent dans l'état initial tr_0 d'un réseau, nous obtenons ainsi un sur-ensemble des marquages accessibles par ce processus par l'ensemble d'équations $\forall 1 \leq i \leq n, x_i \cdot (m(pr)^{tr}, fire(pr)^{tr}) = x_i \cdot (m(pr)^{tr_0}, fire(pr)^{tr_0})$.

La même surestimation peut être obtenue pour les marquages atteints par un processus pr créé dynamiquement par le franchissement d'une transition abstraite tr : $\forall 1 \leq i \leq n, x_i \cdot (m(pr)^{tr}, fire(pr)^{tr}) = x_i \cdot (\Omega(t), \vec{0})$ (où $\Omega(t)$ correspond au marquage de départ associé à t).

Le fait que dans la pratique, de nombreuses transitions sont mortes en fonction du marquage initial est une source additionnelle de surestimation. Nous présentons à présent une méthode permettant de pallier ce problème. Cette méthode pourrait être employée pour les réseaux ordinaires même si cela est peu justifié vu que les transitions y sont rarement mortes.

L'algorithme 7 calcule simultanément un ensemble d'invariants linéaires satisfaits par les marquages accessibles à partir de m et un sur-ensemble des transitions franchissables au moins une fois à partir de m.

entrée : un marquage m
résultat : un ensemble d'invariants et un ensemble de transitions
1 $T_{live} = \emptyset$;
2 $New = \emptyset$;
3 $In = \emptyset$;
4 **repeat**
5 $New = \emptyset$;
6 $In = Invariant(N, m, T_{live})$;
7 **foreach** $t \in T \setminus T_{live}$ **do**
8 Construire un problème linéaire Pb dans $\mathbb{N}^P$ avec In et $Pre(-, t)$;
9 **if** Pb *admet une solution* **then**
10 $New = New \cup \{t\}$;
11 **end**
12 **end**
13 $T_{live} = T_{live} \cup New$;
14 **until** $(New == \emptyset)$;
15 **return** $\langle In, T_{live} \rangle$;

Algorithme 7 : $structReach$

Plus précisément, il initialise T_{live} comme étant l'ensemble vide. Puis, il calcule les invariants positifs dans le réseau récursif réduit à T_{live}. Dans l'algorithme 7, la fonction $Invariant$ retourne une famille génératrice d'invariants (voir [COL 90] pour un calcul efficace de telles familles).

Pour chaque transition absente de T_{live}, il construit un problème linéaire avec ces invariants et la pré-condition de cette transition. Si le problème admet une solution, elle est potentiellement franchissable et est donc ajoutée à T_{live}. Ce procédé est itéré jusqu'à ce que T_{live} soit saturé.

Finalement, nous décrivons comment les invariants linéaires peuvent être employés pour obtenir des informations relatives à la structure des états. Nous construisons un graphe dans lequel les nœuds sont les transitions abstraites. Il existe un arc de t à t' si à partir du marquage de départ de t, un processus peut franchir t'. Pour déterminer si un tel arc existe, nous calculons les invariants associés au marquage de départ de t par un appel à $structReach$. Si t' appartient à l'ensemble de transitions retourné alors un arc entre t et t' est ajouté.

Ce graphe est le squelette de la structure dynamique des états du réseau. Par exemple, si ce graphe est sans circuit alors tous les états accessibles ont une profondeur finie.

9.4.3. *Liens avec d'autres travaux*

Nous avons vu que le modèle des réseaux de Petri peut être étendu tout en préservant la décidabilité de nombreux problèmes. Toutefois, il est à noter que des extensions qui pourraient sembler mineures conduisent à des résultats d'indécidabilité. Par exemple, le problème d'accessibilité est indécidable pour des réseaux de Petri ayant deux arcs inhibiteurs alors qu'il reste décidable pour des réseaux n'en contenant qu'un (ou une structure d'arcs inhibiteurs imbriqués). Les réseaux de Petri auto-modifiants introduits par R. Valk (comme les réseaux à arcs inhibiteurs) ont la puissance des machines de Turing et, en conséquence, beaucoup de problèmes incluant l'accessibilité sont indécidables [VAL 78a, VAL 78b]. De plus, ces formalismes ne permettent pas de modéliser facilement la création dynamique de processus.

Parmi les travaux permettant une telle modélisation, nous pouvons citer le formalisme introduit par A. Kiehn et appelé les « net systems » [KIE 89]. Il est à noter que ce formalisme est strictement inclus dans celui des réseaux récursifs.

Dans [MAY 97], R. Mayr a introduit une hiérarchie d'algèbres de processus (incluant les réseaux de Petri) pour lesquelles il étudie d'une part l'expressivité de chacune ainsi que la décidabilité de nombreux problèmes. Cette hiérarchie peut être rapprochée de celle que forme les RdPR et les RdPRS. Dans [HAD 00], les auteurs ont montré que les « *Process Algebra Nets* » (un des modèles les plus expressifs de R. Mayr) sont inclus dans les réseaux de Petri récursifs. Même si les « *Process Algebra Nets* » (ainsi que les « *Process Rewrite Systems* », un modèle plus expressif) ne peuvent représenter des systèmes de transitions ayant un degré entrant infini, leur inclusion stricte par les RdPR reste un problème ouvert.

Une approche de vérification non abordée dans cette section concerne les relations d'équivalence qui peuvent lier deux modélisations d'un même système. Ce type de vérification est très utile dans une approche de conception par raffinements successifs. Dans un travail récent (mais pas encore publié), il a été montré que la bissimulation entre un RdPRS (satisfaisant des contraintes particulières) et un automate fini est un problème décidable. La section suivante étudie de façon plus précise le problème de la bissimilarité pour des algèbres de processus très expressives.

9.5. π-calcul et vérification

Dans cette section, nous explorons l'expressivité et les possibilités de vérification dans une *algèbre de processus* de la famille des π-calculs. Ces algèbres se distinguent par la possibilité de passer des canaux de communication entre processus, leur conférant un grand pouvoir d'expressivité. Nous explorons ici une variante proche de la version originale (parfois dite « pure ») du π-calcul, telle que proposée et développée par R.Milner [MIL 99].

9.5.1. *Bases du langage*

Le π-calcul (et avant lui CCS) est souvent présenté comme un (lointain) descendant de la longue tradition des λ-calculs, mais déplacé sur une nouvelle problématique : celle de la concurrence et de la mobilité[1]. Dans la tradition des *modèles de calcul* tel le λ-calcul, le π-calcul se veut minimaliste. Ceci se retrouve dans la syntaxe du langage, que nous proposerons sous la forme suivante :

$$\begin{array}{lll} \text{Definition } A, \ldots & ::= & A(x_1, \ldots, x_n) = P \\ \text{Process } P, Q, \ldots & ::= & 0 \mid \alpha.P \mid P \parallel Q \mid P + Q \mid [a = b]P \mid [a \neq b]P \\ & & \mid (\nu n)P \mid A(v_1, \ldots, v_n) \\ \text{Prefix } \alpha, \ldots & ::= & \tau \mid c!a \mid c?(x) \end{array}$$

Les spécifications en π-calcul sont composées de définitions paramétrées, d'expressions de processus et de préfixes d'action. Dans la version la plus épurée du langage, les données primitives manipulées par les processus sont les *noms*, que nous trouvons ici sous deux formes :

1) les *noms atomiques* comme c,d,x, etc. Ils seront pris dans un ensemble infini et dénombrable $\mathcal{N}$, et

2) les *noms substituables* notés $\underline{c}$,$\underline{d}$,$\underline{x}$,etc. Ces noms appartiennent à l'ensemble $\underline{\mathcal{N}}$.

La différence entre les deux catégories de noms est relativement subtile. Les noms atomiques ne sont pas substituables, leur « identité » en quelque sorte, est préservée. Les noms substituables (ou substituables tout court) peuvent être « modifiés », remplacés par d'autres noms. Ceci aura un impact important pour le passage des canaux de communication[2].

Les expressions de processus sont composées à l'aide des constructeurs suivants :

– 0 est le processus *inerte* qui marque la terminaison,

– $\alpha.P$ est la composition séquentielle d'un préfixe d'action α, suivi d'une continuation P elle-même une expression de processus.

– $P \parallel Q$ représente la *composition en parallèle* des processus P et Q, la sémantique sous-jacente (décrite à la section suivante) sera l'*entrelacement* du comportement des deux processus, et leur synchronisation éventuelle.

– $P + Q$ est le *choix non-déterministe* entre les deux processus,

– $[a = b]P$ effectue un test d'égalité entre les noms a et b, si le test est vérifié alors le processus continue en P. Si le test est invalidé, le processus correspond à 0. Une

1. Par mobilité, le π-calcul entend mobilité des liens de communication, donc plutôt réseau mobile que code mobile.

2. La distinction nom atomique/nom substituable n'est pas présente dans le π-calcul original, elle est introduite dans [PES 04] et permet de clarifier le passage de canal.

remarque importante est que l'égalité $[\underline{a} = \underline{b}]$ ne peut être évaluée, il est nécessaire « d'attendre » que les deux noms soient substitués (évaluation paresseuse).

– $[a \neq b]P$ effectue un test d'inégalité entre les noms a et b, la continuation est P.

– $(\nu n)P$ correspond à la construction d'un *nom privé*, uniquement connu de P (mais communicable) et lié à la variable n.

– $A(v_1, \ldots, v_n)$ représente un *appel* à une définition paramétrée.

Dans la composition séquentielle $\alpha.P$, α représente un préfixe correspondant à une action atomique de processus. Les préfixes d'actions proposés sont :

– τ est une *action interne* dont la nature ne peut être observée de l'extérieur.

– $c!v$ est l'*émission* du nom v sur le nom c utilisé en tant que canal. Lorsqu'aucune valeur n'est émise (signal simple), nous écrivons $c!$.

– $c?(x)$ est la *réception* d'un nom sur le canal c, ce nom est lié à la variable x dans la continuation. La synchronisation sur un signal pure s'écrit $c?$.

Ce langage, malgré le petit nombre de ses constructeurs, est connu (et reconnu) pour son expressivité. Du point de vue de l'expressivité « théorique », on peut par exemple caractériser de façon très concise le λ-calcul associé à une stratégie de réduction particulière dans la sémantique du π-calcul [MIL 99]. D'un point de vue plus pratique, on peut consulter [PES 06] par exemple, qui présente une implantation du langage, ainsi que d'un certain nombre d'encodages pour des paradigmes variés (*dataflows*, programmation fonctionnelle, orientation objet, etc.). En guise d'illustration, considérons l'exemple suivant :

$$(\nu c)\,(d!\underline{c}.0 \parallel \underline{c}?(x).([\underline{x} = b].e!a.0 + [\underline{x} \neq b].e!b.0)) \parallel d?(y).\underline{y}!b.0$$

Nous donnons la sémantique informelle de cette expression de processus. Les deux processus en parallèle au premier niveau commencent, à gauche, par la construction d'un nom privé identifié par c et, à droite, par une réception sur le canal d. Comme il n'y a pas d'émission en parallèle sur d, seule la partie de gauche est active au début, ce qui fait évoluer[3] le processus de la façon suivante :

$$\rightarrow d!\nu_c.0 \parallel \nu_c?(x).([\underline{x} = b].e!a.0 + [\underline{x} \neq b].e!b.0) \parallel d?(y).\underline{y}!b.0$$

Les occurences du substituable $\underline{c}$ dans la portée de la restriction (νc) ont été remplacées par un *nom frais* ν_c dont la principale propriété est d'être différent de tous les autres noms utilisés dans le processus. Il y a maintenant trois processus au premier niveau. Le processus le plus à gauche émet le nom nouvellement construit sur d, ce qui permet de synchroniser avec le processus le plus à droite, pour obtenir :

3. La relation de réduction $\rightarrow$ est utilisé de façon informelle ici ; elle correspond intuitivement à l'exécution en un pas d'un processus donné.

$$\rightarrow 0 \parallel \nu_c?(x).([\underline{x} = b].e!a.0 + [\underline{x} \neq b].e!b.0) \parallel \nu_c!b.0$$

Le canal reçu sur d dans le processus de droite est maintenant utilisé en tant que canal pour l'émission du nom b. Le nom ν_c qui était initialement privé et donc restreint aux deux processus de gauche a donc vu sa portée étendue (on parle d'*extrusion de portée*) au processus de droite. L'étape suivante consiste en la communication du nom b, ce qui donne :

$$\rightarrow 0 \parallel ([b = b].e!a.0 + [b \neq b].e!b.0) \parallel 0$$

Le choix non-déterministe restant peut-être maintenant résolu puisque seule la branche de gauche peut être activée (test validé), on obtient finalement :

$$\rightarrow 0 \parallel e!a.0 \parallel 0$$

Ce système ne peut plus évoluer de façon interne, il « attend » en quelque sorte qu'un processus extérieur tente de recevoir sur le canal e.

9.5.2. *π-calcul et sources d'infini*

Il est intéressant, dans ce chapitre, d'essayer de traduire notre discussion sur les sources d'infini (cf. section 9.2 ci-dessus) dans les termes du π-calcul. En fait, il est assez simple de définir un système concis mettant presque toutes les sources d'infini évoquées en jeu :

$$Server(port) = port?(com).port!\underline{com}.(Serveur(port) \parallel Echo(\underline{com}))$$
$$Echo(com) = com?(req).com!\underline{rep}.Echo(com)$$

Il s'agit de la description d'un comportement de serveur. Les deux définitions définissant ce comportement sont paramétrées sur un domaine infini (celui des noms). L'information portée par un nom étant en fait le nom lui-même, le paramétrage est d'une grande régularité. Nous ne l'aborderons pas ici mais on peut penser à l'introduction de types de données plus riches comme les entiers ou autres ; il ne faut donc pas mettre les questions posées par le paramétrage de côté.

Lorsqu'un client (ici non spécifié) se connecte à un serveur, il doit tout d'abord demander à ce dernier une connexion sur le canal $port$. Le serveur crée un canal de communication privé qu'il retourne au client. Il lance alors un processus mis en parallèle avec la continuation du serveur (qui se met donc en attente d'un nouveau client). Le second processus appelle la définition $Echo$ pour traiter les requêtes du client spécifique, la communication se faisant via le canal de communication privé entre le client et le serveur. Nous ne modélisons ici qu'un comportement minimal de service d'écho qui émet en sortie tout ce qu'on lui fournit en entrée. C'est l'architecture de contrôle du serveur plus que son fonctionnement interne qui nous intéresse. Le comportement du serveur met en lumière la création dynamique de deux types de ressources :

1) la création de noms privés pouvant servir de canaux de communication,

2) la création de processus parallèles.

Nous remarquons, de plus, que les définitions sont récursives, et l'association de ces structures récursives avec de la création dynamique de processus fait entrer cet exemple pourtant simple dans la catégorie intrinsèquement complexe des systèmes à contrôle infini.

Du point de vue de la compositionalité, il est intéressant de noter que nous pouvons décrire, même informellement, le comportement du serveur sans pour autant en décrire un système concret confrontant ce serveur à un ou plusieurs clients.

Si l'on caractérise la sémantique d'un système fermé comme par exemple :

$$(\nu ctx)\,(ctx!.ctx?(echo).echo!hello.echo?(x).0 \parallel Server(ctx))$$

il n'est pas possible, dans ce cas, de raisonner sur la possibilité de brancher le serveur à plusieurs clients, ou de mettre plusieurs serveurs en parallèle, etc. Décrire un sous-système comme un *système ouvert*, c'est-à-dire indépendemment de l'ensemble des contextes dans lesquels ce sous-système peut être placé, permet de raisonner non pas sur le système lui-même mais sur les possibilités de composer des systèmes de plus haut niveau et de plus grande taille à partir de ce sous-système. Du point de vue fermé, l'expression $e!a.0$ représente un processus bloqué sur une émission. Du point de vue ouvert, c'est un processus qui « veut » communiquer sur e puis terminer son exécution. L'objectif de compositionalité requiert donc un point de vue différent, plus large mais aussi plus complexe (et plus combinatoire) sur le comportement des processus. D'un point de vue formel (cf. section suivante), il faudra quantifier, en quelque sorte, sur l'ensemble des contextes possibles pour ce processus serveur ; il faudra donc caractériser très précisément, et de façon si possible exhaustive son *environnement*. Or, sur cet exemple précis, l'environnement du serveur peut être composé d'une infinité potentielle de clients.

Finalement, la vraie différence enter le π-calcul et les algèbres de processus qui l'ont précédées (comme CCS et moins directement CSP ou ACP) concerne la *dynamicité*, capturée par la possibilité de communiquer des noms en tant que canaux de communication. Notre exemple de serveur d'écho en fait grand usage. Dans l'expressions $c?(x).\underline{x}!a.0$, un nom lié à la variable x est attendu sur le canal c. L'occurrence substituable $\underline{x}$ est ensuite utilisée en tant que canal de communication pour émettre le nom a. Du point de vue des systèmes ouverts, il s'agit donc d'un processus désirant émettre sur un canal qu'il ne connait pas et dont la valeur précise dépend de l'environnement dans lequel on place le processus. Ceci dénote une structure de branchement infinie mais heureusement hautement régulière comme nous le verrons par la suite.

Retenons donc de ces quelques exemples que le π-calcul, malgré sa simplicité syntaxique, est d'une grande richesse sémantique et source de multiples infinis. Dans le reste de cette section, nous explorons certaines des pistes connues pour aborder les possibilités de vérification dans ce monde infini.

9.5.3. *Systèmes de transitions et bisimilarité*

Pour aborder la question pratique de la vérification dans le cadre des algèbres de processus, en particulier le π-calcul, nous devons faire une incursion dans les constructions sémantiques sous-jacentes. Celles-ci sont basées sur les *systèmes de transitions étiquetées* (LTS) et les équivalences comportementales par *bisimulation*. Un LTS est un automate potentiellement infini dont les états sont des expressions de processus et les transitions sont étiquetées par les actions atomiques effectuées pour « réduire » les expressions. La bisimulation est une relation permettant de comparer deux LTS en terme d'équivalence de comportement.

Dans le cadre du π-calcul, les actions observables (étiquettes des transitions) sont principalement les pas internes τ, les émissions $c!v$ et les réceptions $c?(x)$. Le passage de canal dénote pour sa part une action d'émission liée $c!\nu n$ dont nous étudierons les propriétés ci-après. La présentation usuelle (e.g. de [PAR 01]) de la sémantique du π-calcul s'éffectue en quatre étapes : (1) syntaxe (cf. ci-dessus), (2) congruence structurelle, (3) sémantique opérationnelle sous la forme d'un système d'inférence et (4) bisimilarité (relation d'équivalence).

Congruence structurelle. La notion de *congruence structurelle* correspond à l'établissement d'équivalences syntaxiques élémentaires, parfois dites « triviales » ou « évidentes ». En effet, malgré sa simplicité la syntaxe du π-calcul permet d'écrire des processus essentiellement égaux, comme $c?(x).\underline{x}!e$ et $c?(y).\underline{y}!e$ (les deux sont alpha-convertibles), de façon différente. Au-delà de ces cas « triviaux », la relation de congruence structurelle correspond parfois à des intuitions fondamentales sur la syntaxe du langage, en particulier sur l'*extrusion de portée* qui décrit le fait de communiquer un nom privé à un autre processus (la portée du nom privé étant alors étendue aux deux processus). D'un point de vue pratique, le principal intérêt de cette relation est de simplifier de façon importante la description de la sémantique opérationnelle (diminution du nombre de règles).

La relation de congruence structurelle $\equiv$ sur les termes de la syntaxe du π-calcul est définie formellement comme la plus petite congruence satisfaisant les règles suivantes :

– $P \equiv Q$ si P et Q sont alpha-convertibles.
– $P \parallel Q \equiv Q \parallel P$, $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$, $P \parallel 0 \equiv P$.
– $P + Q \equiv Q + P$, $P + (Q + R) \equiv (P \parallel Q) + R$, $P + 0 \equiv P$.

– $A(v_1, \ldots, v_n) \equiv P\{v_1/x_1, \ldots, v_n/x_n\}$ si $A(x_1, \ldots, x_n) = P$.
– $[a = b]P \equiv P$ si $a = b$ et $[a \neq b]P \equiv P$ si $a \neq b$.
– Règles d'extrusion de portée :
- $(\nu n)0 \equiv 0$ et $(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$.
- $(\nu n)(P \parallel Q) \equiv P \parallel (\nu n)Q$ si $\underline{n}$ n'est pas libre dans P.
- $(\nu n)(P + Q) \equiv P + (\nu n)Q$ si $\underline{n}$ n'est pas libre dans P.
- $(\nu n)[a = b]P \equiv [a = b](\nu n)P$ si $a \neq \underline{n}$ et $b \neq \underline{n}$.
- $(\nu n)[a \neq b]P \equiv [a \neq b](\nu n)P$ si $a \neq \underline{n}$ et $b \neq \underline{n}$.

Nous rappelons qu'une équivalence, telle que $\equiv$, est congruente (ou compositionnelle) si elle passe au contexte des constructeurs du langage. Ainsi, on a par exemple : $P \equiv P' \Longrightarrow P \parallel Q \equiv P' \parallel Q$. Les règles de congruence les plus intéressantes sont probablement celles liées à l'extrusion de portée. On voit que le contexte d'un nom privé, c'est-à-dire la portée d'un (νn), peut être manipulé de façon syntaxique. On peut soit fermer, soit étendre la portée à condition de ne pas introduire de collision en terme de nom.

Considérons par exemple le processus $c!v \parallel (\nu n)c?(x).\underline{x}!\underline{n}$. On voit que le nom n n'est pas libre : il est présent dans le processus à droite de l'opérateur parallèle $\parallel$ sous la forme d'une occurence substituable $\underline{n}$. De plus, il n'existe pas d'occurrence liée à gauche de l'opérateur $\parallel$, on peut donc en extraire la portée et obtenir comme expression structurellement équivalente : $(\nu n)\,(c!v \parallel c?(x).\underline{x}!\underline{n})$. En revanche, on ne peut effectuer la même transformation avec $c!\underline{n} \parallel (\nu n)c?(x).\underline{x}!\underline{n}$ puisque $\underline{n}$ est libre à gauche du parallèle. Ceci signifie, en fait, que le $\underline{n}$ de gauche n'est pas le même que celui à droite du parallèle. On peut utiliser l'alpha-conversion (renommage sans « collision » des noms liés) en réécrivant le processus de la façon suivante : $c!\underline{n} \parallel (\nu m)c?(x).\underline{x}!\underline{m}$. Maintenant on peut à nouveau extraire la portée et obtenir, de façon équivalence : $(\nu m)\,(c!\underline{n} \parallel c?(x).\underline{x}!\underline{m})$ puis $\underline{m}$ n'a pas d'occurrence libre à gauche. Les autres règles d'extrusion de portée fonctionnent de façon assez similaire. On voit ainsi qu'une bonne partie du « jeu » sur la portée dynamique des noms (et donc des canaux) est en fait de nature purement syntaxique.

Sémantique opérationnelle. La présentation usuelle de la sémantique opérationnelle du π-calcul se fait sous la forme d'un système d'inférence logique des transitions du LTS sous-jacent. Un arbre de preuve correspondra donc à l'inférence d'une transition possible, l'ensemble des transitions étant l'ensemble des preuves possibles sur le système d'inférence.

L'unique axiome du système correspond à l'observation directe d'une action atomique α (émission $c!v$, réception $c?(x)$ ou pas interne τ) :

$$\text{Prefix} : \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

Pour réduire le nombre de règles, on établit une connexion entre la congruence structurelle et le système d'inférence de la façon suivante :

$$\text{Struct} : \frac{P \equiv P' \quad P \xrightarrow{\alpha} Q \quad Q \equiv Q'}{P' \xrightarrow{\alpha} Q'}$$

La signification est la suivante. Si on peut prouver que P, en effectuant l'action atomique α, continue en Q et si les processus sont respectivement congruents à P' et Q' alors on peut également inférer que P' transite par α en Q'. La règle suivante donne la sémantique du choix non-déterministe :

$$\text{Sum} : \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

Ici, c'est la branche de gauche qui semble choisie mais si l'on combine avec la règle $Struct$ alors la branche de droite peut tout aussi bien être choisie puisque $P + Q \equiv Q + P$. Le parallélisme et la communication sont les aspects les plus complexes à décrire formellement. Les deux règles principales sont les suivantes :

$$\text{Par} : \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{Com} : \frac{P \xrightarrow{c!e} P' \quad Q \xrightarrow{c?(x)} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'\{e/\underline{x}\}}$$

La règle Par indique ce qu'il se passe par défaut, lorsqu'il n'y pas d'interaction entre les deux processus. Dans ce cas, chaque branche transite de façon indépendant. La condition d'indépendance n'est pas précisée mais revient à garantir que si α est une réception $c?(x)$, alors $\underline{x}$ n'est pas libre dans Q (cf. [PAR 01]). En revanche, la règle Com infère une communication entre une émission et une réception, enregistrée sous la forme d'une transition silencieuse ($\xrightarrow{\tau}$) et une substitution de la variable liée $\underline{x}$ par la valeur émise e dans le receveur Q'.

Les deux dernières règles correspondent respectivement au passage au contexte de la restriction (νn) et au pendant dynamique de l'extrusion de portée :

$$\text{Res} : \frac{P \xrightarrow{\alpha} P' \quad \underline{n} \notin \alpha}{(\nu n)P \xrightarrow{\alpha} (\nu n)P'} \qquad \text{Open} : \frac{P \xrightarrow{c!\underline{n}} P' \quad c \neq \underline{n}}{(\nu n)P \xrightarrow{c!\nu n} P'}$$

Dans le premier cas, la portée de (νn) n'est pas modifiée si le nom privé (ici $\underline{n}$) n'est pas émis. En revanche, dans la règle $Open$ la portée est étendue et pourra être restreint à nouveau via les règles de congruence structurelle.

Pour illustrer l'utilisation du système d'inférence, considérons le processus suivant : $P \stackrel{def}{=} c!a \parallel c?(x).\underline{x}!b$. On peut inférer la transition suivante : $P \xrightarrow{\tau} 0 \parallel a!b$ L'arbre de preuve est le suivant :

$$\text{Com} : \frac{\text{Prefix} : \dfrac{}{c!a \xrightarrow{c!a} 0} \quad \text{Prefix} : \dfrac{}{c?(x).\underline{x}!b \xrightarrow{c?(x)} \underline{x}!b}}{P \xrightarrow{\tau} 0 \parallel a!b}$$

Une autre transition possible est la suivante :

$$\text{Par} : \frac{c!a \xrightarrow{c!a} 0}{P \xrightarrow{c!a} 0 \parallel c?(x).\underline{x}!b}$$

On peut se demander pourquoi cette transition est enregistrée. En fait, il faut considérer le processus P dans l'ensemble des contextes possibles. Par exemple, si l'on écrit $P \parallel Q$ avec $Q \stackrel{def}{=} c?(y).\underline{y}!e$, il est possible que l'émission sur c dans P puisse interagir avec le nouveau processus Q. Si on n'enregistre pas la deuxième transition ci-dessus, ce qui revient à forcer la communication à l'intérieur de P, alors on se coupe des interactions possibles avec des processus externes, on se coupe en fait de l'environnement du processus.

Bisimilarité. Le dernier élément constitutif de la sémantique du π-calcul concerne les équivalences comportementales associées aux systèmes de transitions étiquetées. En terme de vérification, il s'agit du premier problème que l'on se pose, bien avant le problème du *model checking* et d'autres techniques de vérification.

Pour cela, on utilise généralement une définition dite *co-inductive* basée sur la notion de bisimulation. On dira qu'une relation $\mathcal{S}$ est une bisimulation entre deux processus P et Q si on peut vérifier, sur leur LTS respectifs, les propriétés suivantes :

– si $P \xrightarrow{\alpha} P'$ alors il existe Q' tel que $Q \xrightarrow{\alpha} Q'$ et $P' \mathcal{S} Q'$.

– de façon complémentaire, si $Q \xrightarrow{\beta} Q'$ alors il existe P' tel que $P \xrightarrow{\beta} P'$ et $Q' \mathcal{S} P'$.

La *bisimilarité* $\dot{\sim}$ est alors définie comme la plus grande bisimulation, c'est-à-dire l'union de toutes les bisimulations. Ainsi, on pourra prouver que $P \dot{\sim} Q$ si on peut trouver une bisimulation $\mathcal{S}$ telle que $P \mathcal{S} Q$.

9.5.4. *Techniques de vérification*

Nous étudions, dans cette section, les avancées les plus récentes en matière de vérification dans le cadre du π-calcul. Dans ce contexte « largement infini », des suppositions fortes sont faites pour garantir la décidabilité des problèmes réellement traités. Ainsi, dans le cadre général, aucun algorithme ne peut étudier de façon exhaustive un LTS infini. La classe de processus la plus largement étudiée (car « étudiable ») est celle du *contrôle fini*. Il s'agit des processus ne créant pas indéfiniment des ressources telles que canaux (ou noms) privés et processus. En revanche, ils intègrent la compositionalité (dénotant une infinité potentielle d'environnements possibles) et la dynamicité (passage des canaux de communication).

Équivalence comportementale. En terme de vérification, le problème de la *bisimilarité* définie ci-dessus n'est pas simple car il s'agit, dans un premier temps, de « trouver » une relation candidate et ensuite de prouver que cette relation est bien une bisimulation. Pour le cas « statique » (sans passage de canal), l'algorithme de Paige et Tarjan [PAI 87] est le plus largement employé, car le plus efficace connu. Le problème de la bisimulation y est ramené à un problème de raffinement de partition dans un graphe. Dans un premier temps, les espaces d'état (i.e. le système de transition représenté sous la forme d'un graphe) S et T des deux processus à comparer est calculé (sous une forme contrainte et minimisée dite *saturée*, essentiellement dépendante de la nature de la bisimulation testée). On voit bien ici la nécessité de « partir » de systèmes de transition finis pour garantir la terminaison de cette étape préliminaire. Dans un deuxième temps, on initialise la partition $\mathcal{P}$ comme étant le couple (S, T) et on la raffine pour obtenir, finalement, une partition stable, preuve d'équivalence.

La complexité au pire est de l'ordre de $mn \log n$ où m est le nombre de transitions et n le nombre d'état des graphes de départ. Il faut noter que cette complexité, polynomiale d'un point de vue algorithmique de graphe, est en fait exponentielle en fonction de la taille « syntaxique » des processus en entrée. Ceci est dû bien sûr à la sémantique d'entrelacement du parallèlisme qui « génère » un nombre exponentiel de branches non-déteriministes. Dans [MON 95] et [PIS 96] des variantes de l'algorithme sont appliquées au π-calcul , et intégrent donc la dynamicité. La plus grande difficulté consiste en l'infini potentiel d'une expression comme $c?(x)$ qui permet de recevoir n'importe quel x sur c. Pire, une expression comme $c?(x), \underline{x}!b$ dénote l'émission de b sur *n'importe quel* canal $\underline{x}$ reçu via c. Les techniques employées dépassent le cadre de ce chapitre introductif, mais correspondent globalement à la mise en œuvre de transitions symboliques, « enregistrant » de l'information contextuelle sur les transitions. Un autre intérêt de l'approche par raffinement de partition est que l'algorithme permet d'extraire une minimisation des systèmes de transition, un peu à la manière de la minimisation des automates finis.

Une autre approche, de type « *on-the-fly* », est proposée dans le *Mobility Workbench* [VIC 94]. Il s'agit de prendre plus directement la définition de bisimulation et

de « tenter » d'égaliser les transitions. Les avantages de l'approche sont bien sûr le fait de ne plus avoir à « pré-calculer » les espaces d'états (gain en mémoire) mais également qu'un résultat négatif peut-être trouvé en temps fini sur un système à contrôle infini. L'inconvénient concerne la complexité de l'algorithme et la mise en œuvre des mécanismes de *backtrack* associés.

Model checking. Le problème du *model checking* est plus difficile, bien sûr, que le test d'équivalence comportementale. A la complexité intrinsèque du langage s'associe l'expressivité de la logique considérée pour l'expression des propriétés à vérifier. Dans le cadre du π-calcul, plusieurs approches ont été proposées, tant en termes de logiques temporelles que de logiques spatiales, moins connues mais non-moins puissantes.

Pour intégrer la dynamicité liée au passage de canal qui conduit à des systèmes infinis, et donc permettre l'emploi de variantes de l'algorithme de Paige et Tarjan, deux approches ont été proposées. La première consiste en la préservation de l'historique des susbtitutions de noms, tout au long de la vérification [FER 03]. La seconde utilise une notion dérivée de bisimulation, dite « ouverte », et qui opère modulo permutation des noms substitués [PIS 96]. Une dernière possibilité est de recourir à une technique « *on-the-fly* », comme dans le cadre du *spatial logic model checker* [CAI 04]. Cet outil implémente une procédure de décision pour le *model checking* de processus π-calcul à contrôle fini (mais à substitutions potentiellement infinies) pour des formules exprimées dans une logique spatiale très expressive. L'approche à la volée permet d'enregistrer les substitutions au fur et à mesure de l'exploration de l'espace d'état, mais la gestion des sauts en arrière est largement complexifiée de ce fait.

Abstraction et types comportementaux. La technique d'*abstraction* consiste à extraire un modèle simplifié, plus facile à analyser (ou tout simplement analysable), à partir du système étudié. Dans le cadre des systèmes infinis, il s'agira donc d'extraire une représentation finie du système. La plus grande difficulté consiste à maintenir une connexion entre le système, qui sera effectivement déployé, et son abstraction utilisée pour la vérification. Dans le cadre du π-calcul, le travail de Igarashi et Kobayashi [IGA 04] est particulièrement intéressant.

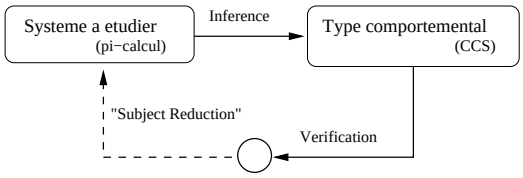


Figure 9.10. *Abstraction par types comportementaux*

La figure 9.10 décrit leur démarche de vérification. L'abstraction d'un processus P est définie comme un type $T(P)$ qui correspond (selon l'isomorphisme de Curry-Howard) à une spécification du processus. Contrairement à l'acceptation usuelle, la notion de type n'est pas ici liée à celle d'analyse statique. En fait, le type extrait est taxé ici de *comportemental* et correspond en pratique à un terme dans une algèbre de processus de type CCS. Ainsi, un ensemble de processus $P_1 \parallel P_2 \parallel \ldots \parallel P_n$ sera typé par un ensemble de processus CCS $T(P_1) \parallel T(P_2) \parallel \ldots \parallel T(P_n)$ où chaque $T(P_i)$ est obtenu *automatiquement* à partir de P_i par un algorithme évolué d'*inférence de type*. Les types extraits sont donc exprimés dans une algèbre de processus de plus faible expressivité que le π-calcul (sans dynamicité), et pour laquelle de nombreux outils de vérification sont disponibles (par ex. le *concurrency workbench* pour les formules CTL).

Un autre intérêt concerne l'existence de nombreux travaux dans le cadre des systèmes infinis, toujours pour CCS (notamment [ROC 99]). La cohérence du modèle proposé repose sur une notion faible de cohérence de type appelée la propriété de *subject reduction*. L'idée est qu'un type comportemental $T(P)$ doit correctement *simuler* le comportement du processus P qu'il type. En termes opérationnels, il faut donc garantir que si le processus P de type $T(P)$ peut évoluer en un processus P' de type $T(P')$ alors $T(P)$ peut « directement » évoluer en un type $T(P')$. Il est clair que des informations sont perdues lors de l'abstraction. Il n'est donc pas possible de vérifier des propriétés arbitraires sur le comportement des processus. La technique se destine plutôt à l'étude de propriétés générales, et plus structurelles que comportementales. Le système de type générique de [IGA 04] a été instancié pour les vérifications suivantes :

– l'analyse de situations de conflits entre processus (*race conditions*),

– l'analyse de verrous passifs (*deadlocks*) et actifs (*livelocks*),

– l'analyse de linéarité sur les canaux de communication (utilisation uniquement en lecture, en écriture, utilisation unique).

Les travaux en cours consistent en l'extension, autant que faire se peut, de l'expressivité du système de type sans sacrifier aux possibilités de vérification.

Compositionalité. Le fait de se concentrer sur les interactions plutôt que sur le comportement interne des systèmes représente un des aspects fondamentaux des travaux sur les algèbres de processus. Dans une sémantique fermée, un processus $c?(x).P(x)$ est bloqué en attente d'une information et ne « fonctionne » donc pas. Dans une sémantique ouverte, une transition $c?(x).P(x) \xrightarrow{c?(x)} P(x)$ est enregistrée, on conserve donc le fait que ce processus « peut » avancer s'il reçoit une information sur le canal c. La différence entre les deux points de vue n'est pas accessoire mais *essentielle*. Le gain est celui de la *compositionalité* qui permet de considérer un système comme une somme de sous-composants plutôt qu'un tout global.

Considérons un système S dont on veut vérifier une propriété $\mathcal{P}$. La démarche consiste en la décomposition du système S en un ensemble de sous-systèmes $S_1 \parallel \ldots \parallel S_n$. On s'intéresse alors à la décomposition de la propriété P en une composition de propriétés $\phi(P_1, \ldots, P_n)$ dont chaque P_i est spécifiques au sous-système S_i. Le processus de spécification et de vérification est alors le suivant :

– vérification indépendante de chaque P_i sur le système S_i,

– vérification de $\phi(P_1, \ldots, P_n)$,

– ce qui donne une preuve de P sur le système S global.

Par exemple, dans [ROC 99], la technique est employée pour composer les propriétés de canaux de communication (perte, duplication et altération). Une grande partie de la difficulté consiste en l'explication de la relation ϕ, le « ciment » entre les sous-propriétés. Dans le cadre de la bisimilarité, par exemple, le caractère congruent (vis-à-vis des constructeurs du langage) de la relation d'équivalence est essentiel. Ainsi, une étape majeure est franchie si on peut prouver que :

$$S_1 \dot{\sim} P_1, \ldots, S_n \dot{\sim} P_n \implies S_1 \parallel \ldots \parallel S_n \dot{\sim} P_1 \parallel \ldots \parallel P_n$$

Pour une algèbre de processus « statique » comme CCS ou CSP, cette propriété dite de congruence est vérifiée. En revanche, le problème se pose dans la variante dite « faible » de la sémantique, qui fournit une abstraction complète vis-à-vis des pas d'exécutions internes. Dans cette bisimilarité faible, notée $\dot{\approx}$, on a par exemple : $\tau.c!a \;\dot{\approx}\; c!a$. Mais se pose alors le problème du passage au contexte du choix : $\tau.c!a + c!b \;\dot{\not\approx}\; c!a + c!b$ (choix interne vs. choix externe). Le π-calcul pose également problème en sémantique « forte » (i.e. sans abstraction vis-à-vis de τ) en raison du passage de nom, et plus précisément au niveau des préfixes de réception. Ainsi, on a trivialement $c?(x) \parallel \underline{b}!e \;\dot{\sim}\; c?(x).\underline{b}!e + \underline{b}!e.c?(x)$ (sémantique d'entrelacement). Mais que dire de : $d?(b).(c?(x) \parallel \underline{b}!e)$? Désormais une communication est possible si le nom c est reçu via d et donc lié à $\underline{b}$, résultant en un pas interne τ initialement absent. Ce problème est la source de nombreuses réflexions dans le π-calcul (par exemple [PES 04]), et semble être au cœur du saut qualitatif du langage en terme d'expressivité.

Vérification et assistants de preuve. Une des approches les plus prometteuse concernant la vérification des systèmes infinis consiste à opérer un rapprochement entre les techniques automatiques (en particulier le *model checking*) et les techniques « assistées » pour la preuve de programme. D'un côté, les limites de la vérification automatiques sont sans cesse repoussées, mais se heurtent inexorablement aux frontères de la décidabilité théorique et de la complexité algorithmique. De l'autre côté, un effort important est actuellement fourni pour améliorer la simplicité et le confort d'utilisation des assistants de preuve. Le « pari » de l'intégration consiste en un idéal probablement réaliste mais indéniablement difficile où les techniques automatiques sont employées

dès que possible, et les passages difficiles, réclamant l'expertise humaine, sont encadrés par des outils aux interfaces évoluées[4].

Le chemin de l'intégration peut se faire selon deux voies complémentaires. On peut partir, par exemple, de procédures de décision et d'outils de vérification automatique existants, et en étendre le langage et/ou la logique pour sortir du cadre « décidable ». On peut dès lors développer des aides à la preuve « maison » (comme dans le cadre de l'atelier B), ou alors générer des obligations de preuve pour un ou plusieurs assistants de preuve. Le chemin opposé est lui aussi possible, consistant en l'implémentation de procédures de décision directement dans l'assistant de preuve. Une telle démarche est proposée dans [AFF 05]. Ce travail propose un encodage du π-calcul dans l'assistant de preuve Coq. Cet encodage est associé à une logique très expressive possédant des opérateurs à la fois temporels et spatiaux. La technique de vérification de propriétés se fait principalement de façon assistée. Cependant, une procédure de réduction par ordre partiel est implantée pour limiter le nombre de cas « manuels » à considérer. Ce travail a été validé dans le cadre du développement d'un serveur de messagerie de qualité industrielle, et à conduit à la découverte d'un certain nombre de bogues malgré l'approche de spécification formelle (mais manuelle) employée pour le développement du serveur.

En conclusion, cette voie d'intégration est certes difficile, mais permet effectivement de considérer l'infini dans toute sa mesure. Dans le monde classique de la programmation séquentielle, le résultat récent sur le théorème des quatre couleurs [GON 05] montre l'intérêt et la puissance du couplage automatique/assisté. Pour les systèmes concurrents et répartis, une bonne partie du chemin reste encore à faire...

9.6. Conclusion

La vérification de systèmes infinis est un problème difficile. Il faut constater que la puissance d'expression des formalismes de spécification (et donc des langages de programmation) conduit aux limites théoriques des techniques de vérification. La plupart des techniques employées en pratique sont basées sur des procédures de semi-décision ou sont semi-automatiques.

De nombreux travaux visent à combiner les différentes approches. Depuis peu, des outils s'attaquent à la vérification de logiciels. Une des approches proposées est basée sur la construction automatique d'une abstraction du programme devant être vérifié. Le processus de construction de l'abstraction doit assurer que l'espace d'état est rendu fini et que tous les comportements du programme original vis-à-vis de la

4. Notons les progrès récents en terme d'interfaces utilisateurs pour les assistants de preuve, et notamment la « preuve-par-pointage ».

propriété à vérifier y sont effectivement représentés. Par contre, comme le mécanisme d'abstraction conduit à l'éviction de certains détails du programme, des comportements nouveaux peuvent être introduits. Des techniques classiques (telles que celles présentées au chapitre 8) peuvent alors être appliquées pour vérifier l'abstraction du programme et si aucun contre-exemple ne peut être exhibé, nous sommes assurés que le programme vérifie la propriété. Dans le cas contraire et si le contre-exemple ne peut être rejoué dans le programme original, le mécanisme d'abstraction est raffiné de manière à interdire ce comportement (quitte à élargir l'espace d'état). Les techniques mises en œuvre dans les mécanismes d'abstraction et de raffinement peuvent faire intervenir des méthodes structurelles telles que celles vues dans le chapitre 7. Les travaux visant à mixer les assistants de preuve et les outils de *model checking* sont également en plein essor.

9.7. Bibliographie

[ABA 93] ABADI M., LAMPORT L., « Composing Specifications. », *ACM Trans. Program. Lang. Syst.*, vol. 15, n°1, p. 73-132, 1993.

[AFF 05] AFFELDT R., KOBAYASHI N., « Partial Order Reduction for Verification of Spatial Properties of Pi-Calculus Processes. », *Electr. Notes Theor. Comput. Sci.*, vol. 128, n°2, p. 151-168, 2005.

[BAR 03] BARDIN S., FINKEL A., LEROUX J., PETRUCCI L., « FAST : Fast Acceleration of Symbolic Transition systems », *Proceedings of the* 15^{th} *Int. Conf. on Computer Aided Verification*, vol. 2725 de *Lecture Notes Computer Science*, Springer-Verlag, p. 118–121, July 2003.

[CAI 04] CAIRES L., « Behavioral and Spatial Observations in a Logic for the pi-Calculus. », *FoSSaCS*, vol. 2987 de *Lecture Notes in Computer Science*, Springer, p. 72-89, 2004.

[CIM 02] CIMATTI A., CLARKE E. M., GIUNCHIGLIA E., GIUNCHIGLIA F., PISTORE M., ROVERI M., SEBASTIANI R., TACCHELLA A., « NuSMV 2 : An OpenSource Tool for Symbolic Model Checking. », *Proceedings of the* 14^{th} *Int. Conf. on Computer Aided Verification (CAV'02)*, vol. 2404 de *Lecture Notes in Computer Science*, Springer, p. 359-364, 2002.

[COL 90] COLOM J. M., SILVA M., « Convex geometry and semiflows in P/T nets. A comparative study of algorithms for computation of minimal P-semiflows », *Advances in Petri Nets*, vol. 483 de *Lecture Notes Computer Science*, Springer-Verlag, p. 79–112, juin 1990.

[ESP 94] ESPARZA J., NIELSEN M., « Decidability Issues for Petri Nets - a Survey », *Bulletin of the European Association for Theoretical Computer Science*, vol. 52, p. 245–262, 1994.

[FER 03] FERRARI G. L., GNESI S., MONTANARI U., PISTORE M., « A model-checking verification environment for mobile processes. », *ACM Trans. Softw. Eng. Methodol.*, vol. 12, n°4, p. 440-473, 2003.

[FIN 93] FINKEL A., « The Minimal Coverability Graph for Petri Nets », ROZENBERG G., Ed., *Proceedings of the* 12^{th} *Int. Conf. on Applications and Theory of Petri Nets (APN'91)*,

vol. 674 de *Lecture Notes in Computer Science*, Gjern, Denmark, Springer-Verlag, p. 210-243, 1993.

[GON 05] GONTHIER G., A computer-checked proof of the four color theorem, Rapport, Microsoft Research, 2005, Available at http ://research.microsoft.com/ gonthier/4colproof.pdf.

[HAD 99a] HADDAD S., POITRENAUD D., « Theoretical Aspects of Recursive Petri Nets », *Proc. of the* 20^{th} *Int. Conf. on Applications and Theory of Petri nets*, vol. 1639 de *Lecture Notes in Computer Science*, Williamsburg, VA, USA, Springer-Verlag, p. 228–247, juin 1999.

[HAD 99b] HADDAD S., POITRENAUD D., Decidability and Undecidability Results for Recursive Petri Nets, Rapport de Recherche n°LIP6 1999/019, Université Paris 6 - CNRS - Laboratoire d'informatique de Paris 6, Paris, France, septembre 1999.

[HAD 00] HADDAD S., POITRENAUD D., « Modelling and Analyzing Systems with Recursive Petri Nets », *Proc. of the* 5^{th} *Workshop on Discrete Event Systems - Analysis and Control*, Gand, Belgique, Kluwer Academics Publishers, p. 449–458, août 2000.

[HAD 01] HADDAD S., POITRENAUD D., « Checking Linear Temporal Formulas on Sequential Recursive Petri Nets », *Proc of the* 8^{th} *International Symposium on Temporal Representation and Reasonning*, Cividale del Friuli, Italie, IEEE Computer Society Press, juin 2001.

[HEL 05] HELJANKO K., JUNTTILA T. A., LATVALA T., « Incremental and Complete Bounded Model Checking for Full PLTL. », *Proceedings of the* 17^{th} *Int. Conf. on Computer Aided Verification (CAV'05)*, vol. 3576 de *Lecture Notes in Computer Science*, Springer, p. 98-111, July 2005.

[IGA 04] IGARASHI A., KOBAYASHI N., « A generic type system for the Pi-calculus. », *Theor. Comput. Sci.*, vol. 311, n°1-3, p. 121-163, 2004.

[JAN 79] JANTZEN M., « On the Hierarchy of Petri Net Languages », *RAIRO*, vol. 13, n°1, p. 19–30, 1979.

[KIE 89] KIEHN A., « Petri Nets Systems and their closure properties », *Advances in Petri Nets 1989*, vol. 424 de *Lecture Notes in Computer Science*, Springer-Verlag, p. 306-328, 1989.

[LER 05] LEROUX J., SUTRE G., « Flat Counter Automata Almost Everywhere ! », *Proceedings of the* 3^{rd} *Int. Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, vol. 3707 de *Lecture Notes in Computer Science*, Springer, p. 489-503, October 2005.

[LOI 95] LOISEAUX C., GRAF S., SIFAKIS J., BOUAJJANI A., BENSALEM S., « Property Preserving Abstractions for the Verification of Concurrent Systems. », *Formal Methods in System Design*, vol. 6, n°1, p. 11-44, 1995.

[MAY 97] MAYR R., « Combining Petri Nets and PA-Processes », *Proc. of the* 3^{rd} *Int. Symposium on Theoretical Aspects of Computer Software*, vol. 1281 de *Lecture Notes in Computer Science*, Sendai, Japan, Springer-Verlag, p. 547–561, 1997.

[MIL 99] MILNER R., *Communicating and Mobile Systems : The* π*-Calculus*, Cambridge University Press, 1999.

[MON 95] MONTANARI U., PISTORE M., « Checking Bisimilarity for Finitary pi-Calculus. », *CONCUR*, vol. 962 de *Lecture Notes in Computer Science*, Springer, p. 42-56, 1995.

[PAI 87] PAIGE R., TARJAN R. E., « Three Partition Refinement Algorithms. », *SIAM J. Comput.*, vol. 16, n°6, p. 973-989, 1987.

[PAR 01] PARROW J., « *Handbook of Process Algebra* », Chapitre An Introduction to the Pi-calculus, p. 479–543, Elsevier, 2001.

[PES 04] PESCHANSKI F., « On Linear Time and Congruence in Channel-passing Calculi », *Communicating Process Architectures*, IOS Press, p. 39–53, 9 2004.

[PES 06] PESCHANSKI F., HYM S., « A Stackless Runtime Environment for a Pi-calculus », *International Conference on Virtual Execution Environments (VEE 2006)*, Ottawa, Canada, ACM Press, june 2006.

[PIS 96] PISTORE M., SANGIORGI D., « A Partition Refinement Algorithm for the *i*-Calculus (Extended Abstract). », *CAV*, vol. 1102 de *Lecture Notes in Computer Science*, Springer, p. 38-49, 1996.

[ROC 99] ROCKL C., ESPARZA J., « Proof-Checking Protocols using Bisimulations », BAETEN J., MAUW S., Eds., *Proc. CONCUR'99*, vol. 1664 de *Lecture Notes in Computer Science*, Springer, p. 525–540, 1999.

[SEG 95] SEGHROUCHNI A. E. F., HADDAD S., « A Formal Model for Coordinating Plans in Multiagents Systems », *Proceedings of Intelligent Agents Workshop*, Oxford United Kingdom, Augusta Technology Ltd, Brooks University, November 1995.

[SEG 96] SEGHROUCHNI A. E. F., HADDAD S., « A Recursive Model for Distributed Planning », *Second International Conference on Multi-Agent Systems*, Kyoto, Japon, December 1996.

[SHA 98] SHANKAR N., « Lazy Compositional Verification. », *COMPOS'97*, vol. 1536 de *Lecture Notes in Computer Science*, Springer, p. 541-564, 1998.

[STE 76] STENNING V., « A Data Transfer Protocol. », *Computer Networks*, vol. 1, p. 99-110, 1976.

[VAL 78a] VALK R., « On the computational power of extended Petri nets », *Proceedings of the* 7^{th} *Int. Symposium on Mathematical Foundations of Computer Science*, vol. 64 de *Lecture Notes Computer Science*, Zakopane, Poland, Springer-Verlag, p. 526–535, September 1978.

[VAL 78b] VALK R., « Self-modifying nets, a natural extension of Petri nets », *Proc. of the* 5^{th} *Int. Colloquium on Automata, Languages and Programming*, vol. 62 de *Lecture Notes Computer Science*, Udine, Italy, Springer-Verlag, p. 464–476, July 1978.

[VIC 94] VICTOR B., MOLLER F., « The Mobility Workbench - A Tool for the pi-Calculus. », *CAV*, vol. 818 de *Lecture Notes in Computer Science*, p. 428-440, 1994.

TROISIÈME PARTIE

Application au développement de systèmes répartis et coopératifs

Chapitre 10

Panorama sur le développement de systèmes répartis et coopératifs

10.1. Approches de développement pour les systèmes répartis et coopératifs

Il est clair que la réalisation d'applications a considérablement changé ces dernières années. D'une approche de développement « en V » [JAU 90], la communauté est passée à une approche de développement « incrémentale » dite aussi « en cascade » puis, récemment vers une approche de type « prototypage » centrée sur un modèle [KOR 03].

Le terme « prototypage » mérite à lui seul une explication. Il est plus ancien et moins « à la mode » que d'autres termes comme MDA (*Model Driven Architecture*), MDD (*Model Driven Development*) ou encore MBD (*Model Based Development*) mais n'en reste pas moins synonyme. Cela dit, quel que soit le terme utilisé, la tendance est unique : il faut supprimer les ruptures dans ce cycle de vie du logiciel et utiliser au mieux les modèles qui ont permis son évaluation pour en extraire les informations cruciales qui serviront à construire un système fiable.

Nous résumons dans un premier temps les principales évolutions dans le domaine du développement de logiciels et ce qu'elles ont apporté. Nous expliquons ensuite comment, dans le domaine des systèmes répartis et coopératifs, les méthodologies de développement les plus récentes peuvent mieux intégrer les méthodes formelles afin d'accroître la fiabilité du logiciel.

Chapitre rédigé par Fabrice KORDON.

10.1.1. *L'approche classique « en V »*

L'approche de développement la plus classique est représentée de manière très naturelle par le cycle du logiciel « en V » [JAU 90], que nous indiquons en figure 10.1. Ce cycle, dès la fin de la décenie 1970, part des besoins et va jusqu'au produit en deux phases principales. La première, descendante, dénote l'analyse du système et la conception du logiciel associé. La seconde, ascendante, exprime le codage, l'assemblage, le test progressif, puis la recette du système complet.

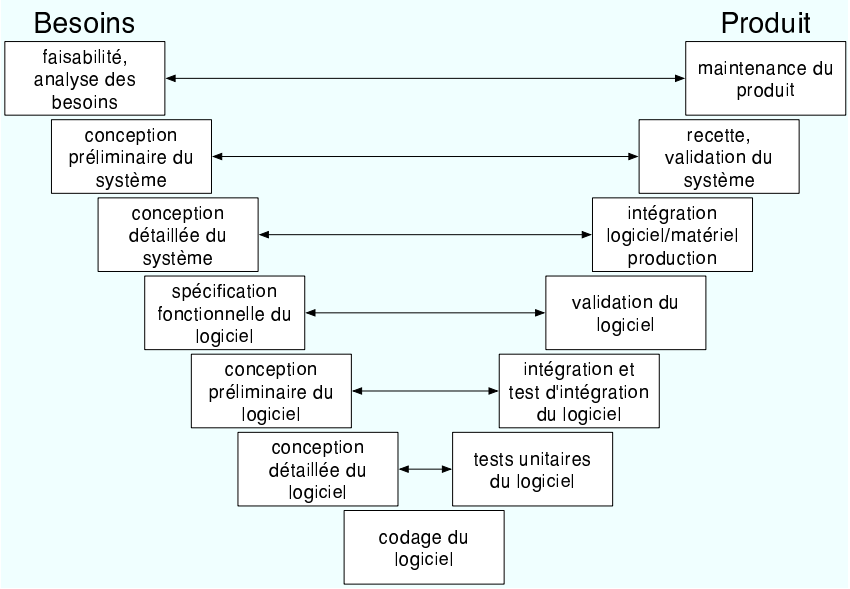


Figure 10.1. *Cycle de développement « en V »*

Notons que cette démarche introduisit un progrès très important : les étapes de la phase descendante avaient toute une correspondance avec une étape de la phase ascendante. Par exemple, lors de la conception préliminaire du système, les équipes de développement doivent préparer les éléments qui permettront de valider le système. L'importance des flèches horizontales sur le schéma de la figure 10.1 n'est pas à minorer.

Cependant, une telle approche de développement reste essentiellement tactique. Elle repose sur l'idée que les besoins sont identifiés dès le départ et qu'il suffit de se mettre « en ordre de bataille » pour résoudre le problème. C'est hélas beaucoup plus compliqué, et tout particulièrement pour les systèmes répartis et coopératifs pour lesquels des aspects stratégiques doivent également être pris en compte.

En effet, les systèmes ne sont plus développés à partir de rien. Il faut donc intégrer des composants logiciels et matériels que l'on ne maîtrise pas forcément (on parle de COTS pour « *Commercial Off-The-Shelf* » – il s'agit souvent de composants « sur étagère » achetés à des sociétés de service). On considère également (à juste titre) les conditions d'utilisation du logiciel et les contraintes qu'il impose sur ses utilisateurs eux-même [COG 94]. Enfin, la spécification des environnements d'exécution – par exemple les *intergiciels*, qui sont une part importante des systèmes répartis et coopératifs – varie si rapidement que les systèmes modernes se doivent d'être capable, d'évoluer avec eux.

Un autre problème est constitué par l'introduction de nombreuses « ruptures » tout au long du cycle de développement. Ces ruptures se matérialisent par l'utilisation de langages et de concepts différents. On part d'un cahier des charges en langage naturel, on passe par l'usage de diagrammes structurés et on termine avec un langage de programmation structuré. A chacune de ces ruptures, des incompréhensions peuvent apparaître, d'autant plus que sur de grand projets, les ingénieurs concernés sont souvent différents.

10.1.2. *Approches incrémentales ou en cascade*

Les approches dites « incrémentales » ou « en cascade » [BUD 84] s'appuient sur la constatation suivante : l'architecture d'une application est un élément essentiel de son évolutivité, et donc, de sa capacité à survivre aux évolutions de l'environnement d'exécution.

On parle déjà parfois de démarche « par prototypage » comme dans [BUD 84, VON 92]. Nous illustrons cette approche dans la figure 10.2. L'idée est de mettre l'architecture du système au centre du développement. Cette architecture est définie très tôt dans la phase de développement. Les composants logiciels sont ensuite développés au fur et à mesure. Plusieurs versions d'un composant logiciel peuvent même être développées, chacune offrant des fonctionnalités supplémentaires.

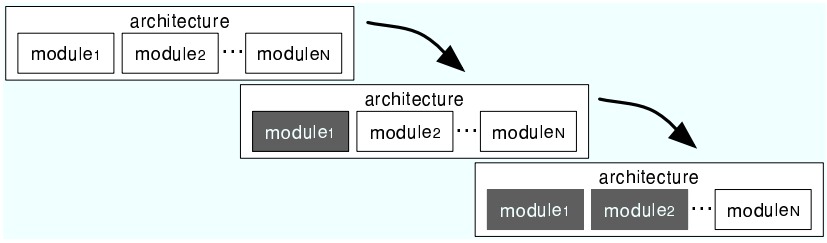


Figure 10.2. *Cycle de développement « incrémental » ou « en cascade »*

Ainsi, le système est développé par itérations successives. On parle d'approche incrémentale en se plaçant sur le plan des fonctions offertes par le système qui sont ajoutées les unes après les autres, et testées avant que l'on procède à une nouvelle intégration.

Cette approche du développement apparaît dans les années 1980. Elle est progressivement utilisée par les acteurs souhaitant s'affranchir des contraintes des environnements. Cela concerne au début les environnements graphiques. Elle se répand réellement dans les années 1990 avec l'apparition des premiers environnements d'exécution répartis comme MPI [MAR 96], CORBA [OMG98] ou l'annexe d'Ada-95 [ada95] pour les systèmes répartis.

Mais l'approche est moins intéressante avec les intergiciels qu'avec les environnements graphiques. L'avantage de ces derniers est une certaine stabilité (on assure en général la compatibilité ascendante) qui rend le passage d'un environnement graphique à l'autre plus facilement portable. Dans un premier temps, l'arrivée d'environnements plus disparates (gestion d'évènements et de *widgets* assez différents entre les principales plate-formes en vigueur) diminue l'intérêt de cette approche. Ensuite, l'arrivée d'approches de répartition assez différentes, supportés par des intergiciels dont aucun ne devient un standard unique porte un rude coup à cette approche.

En effet, le défaut de la démarche réside dans son essence même. Si l'architecture est suffisamment flexible, il sera possible d'adapter le système, non seulement au cours du développement, mais également pendant la phase de maintenance. Seulement, si les intergiciels masquent les disparités des systèmes d'exploitation, ils introduisent une autre dépendance sur le *modèle de répartition*, c'est-à-dire la philosophie d'assemblage et d'interaction des composants d'un système réparti et coopératif. Or le modèle de répartition a un impact non négligeable sur l'architecture du système. On ne peut aisément passer d'une approche de répartition centrée sur le passage de message à une autre basée sur les objets répartis. C'est le *paradoxe de l'intergiciel* identifié dans [PAU 01]. C'est la limite de la démarche incrémentale.

10.1.3. *Nouvelles approches centrées sur un modèle*

De nombreuses études récentes sur le processus de développement tendent à mettre en avant la notion de modèle. Cette approche est apparue dès les années 1990 [LUQ 96] mais ne s'est réellement imposée qu'au début des années 2000 avec les *Model Driven Architecture* (MDA) [mda03].

Différents termes désignent cette approche de développement, chacun mettant en avant certains aspects. MDA se focalise sur les aspects architecturaux. Dans son voisinage, on trouve une série de travaux en vue de décrire les architectures logicielle et matérielle des systèmes répartis [MED 00]. MBD (*Model Based Development*), ou

MDD (*Model Driven Development*) selon les auteurs, insiste plutôt sur la production automatique de code à partir d'un modèle ainsi que des propriétés que l'on peut en extraire par différentes techniques [KOR 03].

La démarche est résumée dans la figure 10.3. L'objectif est de définir, gérer et maintenir un modèle et non plus des programmes. Ce modèle décrit les différents aspects du système (architecture, comportement, etc.) et sert de base à l'évaluation de la solution. L'analyse du modèle peut se faire de manière empirique, par animation ou simulation, ou via des techniques plus formelles. Elle peut être qualitative (identification des caractéristiques du système) ou quantitative (évaluation des performances de la solution). Une fois le modèle analysé, on en dérive une implémentation de manière automatique ou semi-automatique. À ce titre, la génération automatique de programmes joue un rôle important. Dans le domaine des applications réparties coopératives, les intergiciels servent souvent d'exécutif (au sens donné à ce terme en compilation).

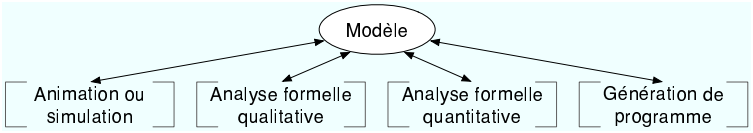


Figure 10.3. *Cycle de développement centré sur un modèle*

Le principal avantage de cette approche est sa flexibilité. L'architecture du système ne repose pas sur le seul modèle qui se situe à un niveau indépendant de la plate-forme d'exécution sous-jacente (PIM ou *Platform Independant Model* dans la terminologie MDA). L'architecture apparaît au niveau PSM (*Platform Specific Model* dans la terminologie MDA) ; elle est idéalement associée à un générateur automatique de programme qui transforme le modèle en un ensemble de sources suivant un point de vue donné. Il est parfaitement imaginable que la sémantique d'un modèle puisse se projeter sur différentes approches opérationnelles dépendantes d'une plate-forme d'exécution donnée. Affranchie d'une architecture et d'une technologie donnée, cette approche semble la plus pérenne de toutes celles envisagées jusqu'à présent.

Un autre avantage, plus spécifique aux systèmes répartis et coopératifs, mérite d'être mentionné. Il s'agit des possibilités d'analyse offertes par cette approche dans le domaine comportemental. En effet, si une démarche de modélisation du type de celles présentées dans la partie I de cet ouvrage est utilisée, alors elle permet une analyse beaucoup plus fine du système à développer, par exemple en utilisant les techniques décrites dans la partie II de cet ouvrage. L'application gagne alors en fiabilité et en sécurité. C'est important dans le contexte actuel où la société dépend de plus en plus des systèmes informatiques. C'est aussi crucial dans ce domaine où la complexité des systèmes réalisés n'est plus appréhendable avec les seules techniques classiques (comme cela a été souligné en page 1 de cet ouvrage).

Cependant, cette approche est extrêmement coûteuse à mettre en œuvre car elle doit être fortement outillée. Ainsi, peu d'environnements sont capables de supporter complètement une telle démarche. Cela dit, plusieurs projets adressent ce type de démarche, comme ALTA-RICA [ARN 00] ou MORSE [GIL 06], tous deux dans le cadre du RNTL (Réseau National des Technologies Logicielles). De même, des grands groupes comme Aérospatiale avec le projet TOPCASED [top], ou Thales avec le projet CARROLL [car] mènent leur propres travaux.

10.2. Du modèle au système réparti et coopératif

Les approches centrées sur un modèle constituent un « graal » à atteindre et ne sont pas encore complètement opérationnelles, même si des expériences sont menées avec succès dans certains domaines d'application comme l'avionique embarquée avec l'outil comme Scade [sca] ou les transports publics avec l'atelier B [pro]. Notons cependant que ces méthodes ne sont pas véritablement dédiées aux systèmes répartis qui sont fortement asynchrones.

10.2.1. *Limites actuelles des approches centrées sur les modèles*

Si des outils précurseurs comme HOOD [GRO 95] ont très tôt développé un modèle de développement assez proche de l'approche centrée sur un modèle, il s'agit plutôt de « programmation visuelle ». HOOD est en effet au départ une vue graphique du langage Ada. Le même reproche peut être fait aux évolutions successives d'UML [uml99, PIL 06]. En effet, l'OMG se focalise principalement sur l'extension du pouvoir de modélisation, ce qui a amené l'adjonction progressive et régulière de nouveaux diagrammes. Mais, cette notation qui offre des possibilités impressionnantes pour des systèmes d'information, notamment en génération automatique de programmes, reste très pauvre en fonctionnalités dans des domaines comme le développement d'applications réparties et/ou embarquées.

Le développement centré sur des modèles n'est pas encore une réalité, car de nombreuses techniques doivent être mises en œuvre tout au long de ce nouveau cycle de développement. Le problème principal reste le coût de réalisation des environnements de développement qui le supportent. Cela se matérialise par exemple sur les points suivants :

– Il faut pour cela revoir la façon « traditionnelle » de considérer ces environnements. Par exemple, un générateur automatique de programme (une sorte de « supercompilateur » travaillant au niveau du modèle) n'est assurément pas un outil aisé à développer. C'est encore moins le cas si l'on considère qu'il doit en plus pouvoir être refaçonnable à volonté pour s'adapter aux besoins spécifiques d'une application. Il s'agit là d'un premier exemple de besoins nouveaux qu'un pan entier de la discipline informatique, *l'ingénierie des modèles*, tente de combler.

– Un autre problème, qui nous intéresse plus dans le cadre de cet ouvrage, est posé par l'analyse formelle des modèles. Si les méthodes formelles commencent à être mûres, il ne leur en faut pas moins continuer à progresser pour permettre le traitement de systèmes en « grandeur réelle ». L'objet des parties I et II de cet ouvrage était de montrer les derniers progrès en la matière. Mais si l'évolution continue dans ce domaine est impressionnante, cela ne suffit pas encore. Les problèmes continuent de croître en importance alors même que les approches formelles évoluent.

La partie III de ce livre est consacrée à l'usage des techniques formelles dans le développement de systèmes répartis et coopératifs « concrets ». Il s'agit dans tous les cas d'approches de développement basées au moins partiellement sur des modèles et s'appuyant sur les méthodes formelles décrites dans les parties I et II.

10.2.2. *Utilisation pragmatique des méthodes formelles*

L'utilisation des méthodes formelles dans le développement d'applications industrielles doit être « pragmatique » afin de pallier le problème de l'explosion combinatoire propre aux domaines d'application qui nous concerne. Donnons quelques exemples ce que nous voulons dire par « démarche pragmatique ».

Une première approche consiste à ne pas modéliser le système dans son intégralité. Par exemple, dans le cas des systèmes répartis et coopératifs, on s'intéresse principalement aux interactions entre les composants car ces dernières sont la réelle source de complexité. Celles-ci sont donc modélisées et étudiées avec attention alors que les éléments séquentiels (typiquement les traitements qui sont déclenchés en fonction des évolutions du système), *a priori* moins complexes, sont développés de manière « traditionnelle ».

L'avantage d'une telle démarche est de considérer plusieurs plans d'action. On se situe dans un contexte proche de la notion d'aspects [PAW 04] car il faut, pour obtenir le système, « tisser » les différents plans entre eux. Par exemple, un générateur de programme pour la partie contrôle du système devra savoir intégrer les traitements spécifiques qui ne peuvent être capturés dans le modèle comportemental du système [GIL 06].

Une autre manière d'être pragmatique est d'utiliser les méthodes formelles très en amont du cycle de développement. Le principal intérêt est alors d'utiliser des abstractions fortes visant à simplifier considérablement la complexité de la spécification. L'objectif est alors de comprendre la dynamique comportementale du système réparti et non d'assurer son développement à partir du modèle.

D'une manière générale, on peut toujours jouer sur les abstractions, ce qui revient à modéliser non la manière dont un mécanisme se comporte mais son effet sur le

système. Par exemple, si l'on suppose que le réseau est fiable, on ne modélise pas le fait qu'une communication a lieu mais plutôt le fait qu'une donnée transmise est disponible pour le destinataire.

L'avantage de cette technique est de poser proprement les hypothèses système et son environnement d'exécution. En effet, chaque abstraction correspond à une série d'hypothèses ou de contraintes qui doivent être respectées pour que le modèle ait un sens. En revanche, elle suppose que l'on sait assurer la validité des hypothèses ainsi faites.

Une troisième manière de procéder pragmatiquement avec les méthodes formelles concerne plutôt la rétro-ingénierie. Il s'agit alors d'isoler une portion (considérée comme délicate) du système et d'en paraphraser le code sous la forme d'un modèle formel que l'on analyse. Cette démarche oblige souvent à un va-et-vient entre le programme qui est décrit et le modèle qui est analysé.

Comme l'on ne s'intéresse qu'à une partie du logiciel, on arrive ainsi à limiter la complexité de la démarche de vérification. En revanche, il faut s'intéresser de près à la relation entre la portion du système ainsi vérifiée et le reste qui ne l'est pas. La pertinence de cette approche est directement liée à l'architecture du système considéré. Il ne faut pas que la preuve partielle s'avère fausse car des éléments n'ont accidentellement pas été pris en compte.

Bien sûr, ces différentes techniques peuvent être utilisées simultanément dans un projet, à des étapes différentes du processus de développement.

10.3. Plan de la partie III

La troisième partie de cet ouvrage présente différentes utilisations des méthodes formelles dans le cadre du développement de systèmes répartis. Ces trois exemples mettent en avant la vérification de propriétés comportementales.

Le chapitre 11 utilise les réseaux de Petri comme modèle d'entrée. L'analyse se fait par *model checking*, l'objectif étant de garantir des propriétés comportementales. L'analyse est ici purement qualitative : on s'intéresse à l'absence de tout comportement déviant sans considération de performance.

Les chapitres 12 et 13 s'appuient sur des automates temporisés. Dans ces études, on s'intéresse également à la vérification de contraintes comportementales impliquant des aspects temporisés. L'analyse est ici à la fois qualitative et quantitative.

Dans les trois études présentées, on « flirte » avec les limites des techniques de vérification actuelles adaptées aux systèmes répartis et coopératifs. Il est intéressant

d'observer les choix qui ont malgré tout permis une utilisation pragmatique de ces techniques. Dans certains cas, cela a également donné lieu à des orientations de travaux de recherche en vue de traiter plus spécifiquement les problèmes posés par ces études afin de repousser les limites actuelles de l'état de l'art.

10.3.1. *Méthodes formelles et développement d'Intergiciel*

Le travail présenté dans le chapitre 11 est un excellent exemple d'utilisation des méthodes formelle dans un cadre de rétro-ingénierie. Il se situe dans le cadre du projet PolyORB et constitue l'une des contributions majeures du travail de thèse de J. Hugues [HUG 05].

Ce travail démontre qu'il y a une relation claire entre l'architecture logicielle d'un système et sa modélisation. Il s'agit de raffiner l'architecture d'un intergiciel afin de lui donner des caractéristiques importantes pour les systèmes temps-réel répartis. Il est intéressant de constater que le processus de restructuration du code en vue de donner à PolyORB de « bonnes » propriétés s'est accompagné d'une réflexion conjointe visant à en préparer la spécification formelle.

Le résultat majeur de cette réflexion est que l'architecture logicielle de PolyORB en a été impactée très favorablement. Notons en particulier l'identification d'un composant central, le μBroker, qui permet :

– de faciliter la vérification en concentrant les aspects difficiles du problème à traiter ; on réduit ainsi la vérification comportementale de l'intergiciel à la vérification de lui seul (les autres éléments de l'architecture pouvant être représentés par des asbtractions très simples) ;

– d'assurer la symétrie complète des éléments de l'intergiciel (threads, sources d'évènements, etc.) ; cela a permis d'utiliser des techniques de *model cheking* optimisées et de vérifier des configurations réalistes ;

– de proposer des gabarits de conception simples et adaptés aux contraintes d'une implémentation dans le domaine considéré (les applications temps-réel réparties et embarquées).

Cette liaison entre intergiciel et vérification est étudiée actuellement par des acteurs majeurs du domaine, en particulier autour de l'intergiciel TAO et des environnements de développement/configuration construits autour, comme Zen [GOR 05]. L'objectif est, à terme, de considérer l'intergiciel comme un COTS (*Commercial Off-The-Shelf*) en vue de faciliter l'approche de certification des logiciels répartis critiques [BUD 03].

10.3.2. *Méthodes formelles et services web*

Le travail présenté dans le chapitre 12 concerne également un sujet d'actualité. L'objectif est d'exploiter les méthodes formelles au langage BPEL (langage « standard » développé par des acteurs majeurs de l'industrie comme BEA, IBM et Microsoft) dans une direction originale : proposer une sémantique du langage afin de construire automatiquement un client capable d'interagir avec le service.

Le chapitre insiste sur l'intérêt de doter un langage d'une sémantique formelle. En effet, la notion d'objet, d'où dérivent les services web, met plus l'accent sur l'interface que sur une sémantique opérationnelle (i.e. un protocole d'interaction entre les composants). Cela peut conduire à une ambiguïté gênante : des services apparemment compatibles ne le sont finalement pas à cause de modes opératoires différents.

Ces travaux sont une base pour, à terme, permettre à un développeur d'applications basées sur les services web de décrire la sémantique opérationnelle des interactions pour chaque composant qu'il crée. C'est capital si l'on veut s'assurer que l'assemblage d'un certain nombre de composants web est cohérent d'un point de vue comportemental. Dans un contexte où le comportement de ce type d'applications réparties devient de plus en plus dynamique (un composant fait appel à un autre composant via un annuaire qui le lui désigne comme étant celui qui répondra le mieux à ses besoins), la notion de service devra, tôt ou tard, prendre en compte le mode opératoire des différents acteurs.

Le principal intérêt de la démarche présentée est de proposer une sémantique formelle de comportement des composants dans le cadre des services web. Cette description s'appuie sur JCWL (*Java Complex Web Service Language*), un langage de haut niveau dédié aux ingénieurs de développement.

10.3.3. *Méthodes formelles et systèmes répartis adaptatifs à contraintes de temps*

Le travail présenté dans le chapitre 13 adresse le cas des systèmes répartis adaptatifs à contraintes de temps. C'est également un point crucial qui doit être étudié car les « applications du futur » (dans le domaine des transports ou de la domotique par exemple) devront s'adapter à des environnements très évolutifs (c'est d'ailleurs déjà le cas pour certains services liés aux téléphones mobiles). L'un des problèmes majeurs dans ces domaines est de s'adapter pour tenir des contraintes temporelles ou de qualité de service.

L'étude de ce type de système est présentée à travers une étude de cas impliquant à la fois des problèmes de comportement et du temps-réel. Il s'agit de la synchronisation entre un danseur et des dispositifs matériels devant accompagner son numéro.

Ce chapitre présente également l'intérêt de bien montrer la relation entre le modèle, la vérification des propriétés attendues, et le code associé. Il s'agit typiquement d'une démarche complète de développement centrée sur des modèles au sens où nous l'avons présenté dans la section 10.1.3.

10.4. Bibliographie

[ada95] *Information Technology – Programming Languages – Ada*, ISO, 1995, ISO/IEC/ANSI 8652 :1995.

[ARN 00] ARNOLD A., GRIFFAULT A., POINT G., RAUSY A., « The AltaRica formalism for describing concurrent systems », *Fundamenta Informaticae*, vol. 40, p. 109-124, 2000.

[BUD 84] BUDDE R., KUHLENKAMP K., MATHIASSEN L., ZÜLLIGHOVEN H., *Approaches to prototyping*, Springer Verlag, 1984.

[BUD 03] BUDDEN T. J., « Decision Point : Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort », *STSC CrossTalk, The Journal of Defense Software Engineering*, 2003.

[car] « CARROLL, the project, `http://www.carroll-research.org` ».

[COG 94] COGUEN J., « *Requirements Engineering : Social and Technical Issues*, », Chapitre Requirements Engineering as the Reconciliation of Social and Technical Issues, p. 165-200, Academic Press, 1994.

[GIL 06] GILLIERS F., KORDON F., THIERRY-MIEG Y., « Processus de fabrication de systèmes répartis centré sur un modèle : l'expérience du projet MORSE », *REE*, A paraître en 2006.

[GOR 05] GORAPPA S., COLMENARES J. A., JAFARPOUR H., KLEFSTAD R., « Tool-based Configuration of Real-time CORBA Middleware for Embedded Systems », *International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, Seattle, USA, May 2005.

[GRO 95] GROUP H. T., « HOOD Reference Manual, release 4 », June 1995.

[HUG 05] HUGUES J., Architecture et Services des Intergiciels Temps Réel, PhD thesis, École Nationale des Télécommunications, Septembre 2005.

[JAU 90] JAULENT P., *Génie Logiciel, les méthodes*, Colin, 1990.

[KOR 03] KORDON F., HENKEL J., « An overview of Rapid System Prototyping today », *Design Automation for Embedded Systems*, vol. 8, n°4, p. 275–282, Kluwer, december 2003.

[LUQ 96] LUQI, « System Engineering and Computer-Aided Prototyping », *Journal of Systems Integration*, vol. 6, n°1, p. 15-17, 1996.

[MAR 96] MARC S., OTTO S., HUSS-LEDERMAN S., WALKER D., DONGARRA J., *MPI : The Complete Reference*, MIT Press, 1996.

[mda03] *MDA Guide Version 1.0.1, document no : omg/2003-06-01*, OMG, 2003.

[MED 00] MEDVIDOVIC N., TAYLOR R., RICHARD N., « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on Software Engineering*, vol. 26, p. 70-93, 2000.

[OMG98] *CORBA Messaging Specification*, OMG, may 1998, OMG orbos/98-05-05.

[PAU 01] PAUTET L., Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition, Habilitation à Diriger des Recherches, Université P. & M. Curie – Paris VI, décembre 2001.

[PAW 04] PAWLAK R., RETAILLÉ J.-P., SEINTURIER L., *La programmation orientée aspect avec Java/J2EE*, Eyrolles, 2004.

[PIL 06] PILONE D., PITMAN N., *UML 2 en concentré : manuel de référence*, O'Reilly, 2006.

[pro] « Atelier B, `http://www.atelierb.societe.com` ».

[sca] « Getting Started with SCADE, `http://www.esterel-technologies.com/technology/getting-started/index.html` ».

[top] « TOPCASED, open source engineering workshop, `http://www.topcased.org` ».

[uml99] Unified Modeling Language Specification, version 1.3, Rapport, OMG, 1999.

[VON 92] VONK R., *Prototypage : l'utilisation efficace de la technologie CASE*, Masson & Prentice Hall, 1992.

Chapitre 11

Construction d'un intergiciel vérifié

Ce chapitre s'intéresse à l'utilisation des méthodes formelles pour la construction et la vérification d'un intergiciel (ou *« middleware »*).

Nous motivons l'utilisation de méthodes formelles pour assurer le bon fonctionnement d'un intergiciel, puis nous présentons le cadre de notre étude : la vérification de configurations de l'intergiciel PolyORB. Enfin, nous détaillons le processus complet de vérifications de configurations complètes.

11.1. Motivations

Les intergiciels sont au centre des applications réparties. Leur rôle est critique : ils gèrent l'émission et la réception des requêtes pour le compte de l'application. Des services avancés (concurrence, tolérance aux pannes, politiques de qualité de service) peuvent être sélectionnés par l'utilisateur pour enrichir la sémantique de l'intergiciel et tirer parti de fonctionnalités avancées.

Ainsi défini, un intergiciel est une pièce maîtresse qui a sous sa responsabilité le comportement de l'application. Cette « brique logicielle » doit apporter les preuves de son bon fonctionnement, en particulier pour des applications réparties critiques (applications de commerce en ligne, applications réparties temps réel dans l'avionique) où une faute implique un préjudice humain ou financier.

Les architectures d'intergiciel classiques ont montré qu'elles pouvaient répondre à des besoins précis. Cependant, nous remarquons qu'elles ne fournissent que peu

Chapitre rédigé par Jérôme HUGUES et Fabrice KORDON et Laurent PAUTET.

d'informations sur leurs propriétés comportementales. Bien souvent, elles reposent sur un ensemble de tests et de scénarios permettant de tester des configurations bien spécifiques, par exemple dans le cas du projet Bold Stroke OFP [SHA 98] mené par Boeing.

Nous notons plusieurs évènements qui conduisent à une explosion combinatoire du nombre de scénarios d'exécution possible d'un système réparti : le nombre de chemins d'exécution admissibles pour une configuration d'intergiciel sur un nœud augmente avec le nombre de tâches et de requêtes à traiter. Le nombre de configurations de l'intergiciel est fonction de ses capacités d'adaptabilité et les plates-formes cibles qu'il supporte.

Ainsi, nous affirmons que l'exécution de tests ne peut permettre à elle seule d'analyser les propriétés comportementales d'un intergiciel telles que *l'absence d'interblocage, l'équité entre les requêtes, le dimensionnement correct des ressources.* Au contraire, les techniques formelles permettent de garantir, lorsqu'elles convergent, qu'une propriété est valide pour le modèle considéré, ou le cas échéant à un contre-exemple montrant en quoi elle est fausse.

Nous proposons donc d'utiliser des techniques de modélisation et de vérification formelle pour assurer certaines propriétés de notre architecture et de son implantation. Nous cherchons ainsi à réduire la *« distance »* entre le modèle et le système réel.

La vérification de systèmes est un domaine d'expertise distinct du domaine des intergiciels. Appliquer les techniques de modélisation et de vérification n'est pas immédiat, et requiert de s'intéresser aux mécanismes existants pour déterminer un processus de vérification permettant de déterminer précisément les propriétés du système.

Pour établir un lien entre la conception d'un intergiciel, et sa modélisation, il est nécessaire d'associer deux compétences distinctes. Ainsi, l'ENST qui a une longue expérience dans la conception et la réalisation d'intergiciels a travaillé en collaboration avec l'équipe MoVe (Modélisation et Vérification), dans le département *Réseaux et Systèmes Répartis* du LIP6 [HUG 04] pour mener notre étude à bien.

11.2. Présentation de l'intergiciel PolyORB

Dans cette section, nous introduisons les éléments clés de l'architecture d'intergiciel dite « schizophrène », et présentons son intérêt pour la vérification de propriétés comportementales d'un intergiciel.

Un intergiciel combine deux facettes complémentaires : (1) une cadre général pour implanter des systèmes répartis, utilisant les ressources de l'hôte et du système d'exécution (processus légers, entrées/sorties) ; et (2) un ensemble de services permettant

la construction d'applications portables. Dans [HUG 03], nous introduisons la notion d'intergiciel schizophrène, une architecture canonique qui promeut ces deux aspects, tout en permettant la séparation des fonctions clés de l'intergiciel.

Dans [VER 04], nous présentons PolyORB, notre implantation de cette architecture. Nous démontrons son intérêt en tant que plate-forme canonique pour supporter plusieurs spécifications (ou modèles) d'intergiciels (CORBA, l'annexe des systèmes répartis de Ada, les applications Web, basés message, etc.), mais aussi en tant que « composant pris sur l'étagère » (COTS) pour l'industrie.

Notre expérience montre qu'un ensemble réduit de services permet de décrire en détail plusieurs modèles de répartition. Nous avons identifié sept étapes clés dans le traitement d'une requête, définies comme autant de services fondamentaux de l'intergiciel.

Ces services sont des composants génériques pour lesquels une implantation basique est fournie. Les développeurs peuvent par ailleurs fournir des implantations alternatives. Chaque instance de l'intergiciel est alors un assemblage cohérent de ces entités.

Un composant central, le « μBroker » coordonne ces différents services : il est responsable de la propagation correcte des requêtes.

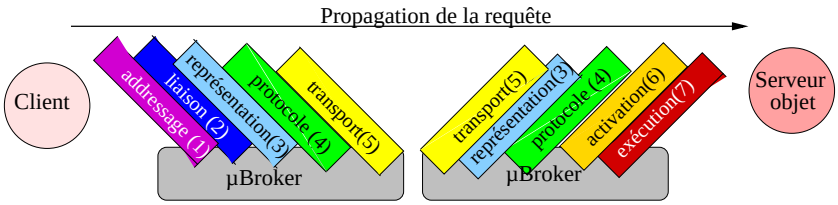


Figure 11.1. *Propagation d'une requête au sein de l'architecture schizophrène*

La figure 11.1 illustre la coopération entre les services de PolyORB pour transmettre une requête générique entre deux entités applicatives distantes.

Dans un premier temps, le client recherche la référence de l'entité distante en utilisant le service *d'adressage* (1), un composant proche d'un dictionnaire. À l'aide de cette référence, le service de *liaison* construit une connexion avec le nœud distant, ce service agit comme une fabrique d'objets.

Les paramètres de la requête sont ensuite transformés en une suite de données à même d'être transférée sur le réseau et interprétée par le nœud distant en utilisant le service de *représentation* (3). Il s'agit de l'application d'une fonction mathématique convertissant une donnée en un flot d'octets.

Un *protocole* (4) est implanté pour la transmission entre les nœuds, au travers du service de *transport* (5) qui établit un canal de communication. Ces deux services peuvent être réduits à des automates. Enfin, la requête est envoyée.

Lors de la réception d'une requête, le nœud assure qu'une entité applicative concrète pourra traiter la requête via le service d'*activation*. Enfin, le service d'exécution (7) affecte des ressources d'exécutions pour traiter cette requête. Ces services reposent sur les patrons de conception *« Fabrique »* et *« pool de ressources »*.

Ainsi, les services de notre intergiciel reposent pour une majeure partie sur des patrons élémentaires : ils calculent une valeur et la passent à un autre service. Notre expérience avec PolyORB montre qu'ils suivent tous le même patron sémantique, et sont simplement adaptés à certaines interfaces particulières. Les implantations réalisées peuvent toutes être ramenées à ces abstractions classiques.

Seul le composant μBroker prend en charge la coordination de ces services : il alloue les ressources et assure la propagation des données au sein de l'intergiciel. De plus, il est le seul composant qui contrôle l'intégralité de l'intergiciel : il manipule les données critiques que sont les processus légers et les entrées/sorties, et les confie de manière ponctuelle à certains services. Ainsi, il capture le patron comportemental de l'intergiciel.

L'architecture schizophrène fournit une description complète d'un intergiciel. Cette architecture sépare un ensemble de services canoniques dédiés au traitement d'une requête du composant μBroker, qui a la charge exclusive du comportement de l'intergiciel. Ainsi, nous isolons la boucle de commande de notre système, présente dans toute instance. Il s'agit donc d'un composant clé sur lequel porterons nos travaux de vérification.

11.3. Définition d'un processus de vérification

Dans cette section, nous présentons le processus de modélisation et de vérification que nous avons retenu. Le comité ISO définit la vérification comme suit :

« La vérification est la confirmation par l'examen et la disposition de preuves objectives que les spécifications ont été remplies » [ISO 94].

La notion de preuves objectives nous incite à rechercher des mécanismes de vérification fiables et non ambigus, limitant les erreurs d'interprétation.

La diversité des formalismes et outils existants nous amènent à faire un choix. Compte tenu du type de systèmes que nous modélisons, et la diversité des configurations à modéliser, nous privilégions une approche automatisable. D'autres paramètres tels que le pouvoir d'expression du formalisme et la disponibilité d'outils sont aussi à prendre en compte pour arrêter notre choix.

11.3.1. *Formalismes pour la modélisation*

Une analyse complète du patron de conception μBroker nécessite dans un premier temps une description précise de ses interfaces, de leur sémantique et des propriétés que l'on souhaite vérifier. Par la suite, cette description est exprimée grâce à une notation qui permet sa vérification formelle ; plusieurs transformations permettent de passer d'un modèle haut-niveau informel à la description précise d'un composant logiciel.

Ceci nous amène à nous interroger sur le choix de la notation (ou l'ensemble des notations) la plus adéquate. Nous avons considéré l'usage de plusieurs notations parmi lesquelles les automates, les diagrammes UML, les réseaux de Petri (colorés, stochastiques, temporisés, etc) ; ainsi que les langages de description d'architecture (*« ADL »*)[1].

Nous notons qu'aucune de ces notations ne vient avec un cycle complet incluant modélisation et vérification. Chacune ne couvre qu'une part restreinte du cycle de vie du système : les diagrammes UML s'intéressent aux spécifications haut niveau ; les réseaux de Petri à la spécification formelle de systèmes de commandes ; les ADLs à la description d'architectures.

Par conséquent, il est nécessaire d'utiliser au moins deux notations pour couvrir à la fois la spécification et la vérification d'un système. Les travaux de recherche actuels s'intéressent à la combinaison de différentes modélisations d'un même système pour fournir sa description complète :

– Une possibilité est de dériver des réseaux de Petri des machines à états et des diagrammes UML [MER 02] à des fins de vérification. Cependant, aucun outil ne permet de réaliser cette tâche automatiquement. L'avènement d'UML2 et notamment les profils UML devraient lever ces limitations ;

– Une autre solution consiste à utiliser des notations spécifiques d'un domaine, tel que AADL (*« Architecture, Analysis and Description Language »*) [FEI 03], initialement conçu pour l'avionique, ou LfP (*« Language for Prototyping »*) [REG 01]. Ces approches montrent comment une notation permet de fédérer plusieurs techniques de modélisation et de vérification. Cependant, ces approches sont embryonnaires, et le manque d'outils pour tirer partie de ces formalismes réduit leurs champs d'utilisation.

Ces projets fournissent cependant un point d'entrée pour la spécification et la vérification formelle de composants logiciels. Ils suivent une approche descendante, de spécifications de haut niveau vers des spécifications formelles plus détaillées, qui permettent la vérification de propriétés du système.

1. Voir le chapitre 5 pour plus de détails sur les langages de description d'architecture

Nous nous proposons de suivre une approche similaire, adaptée à un problème spécifique : la vérification de plusieurs configurations du μBroker. Les chapitres précédents ont fourni une vue précise des différents composants et interfaces mis en œuvre. Dans les sections suivantes nous nous intéressons à leur modélisation formelle.

11.3.2. *Techniques de modélisation et vérification*

Il existe deux grandes familles de méthodes formelles : la première basée sur une approche par preuves (tels que B [ABR 95] ou Z [DIL 94]), la seconde basée sur la vérification de modèles par parcours exhaustif du graphe d'états (*« model checking »*), en utilisant des outils ou langages tels que SPIN [HOL 04] ou LUSTRE [HAL 93].

Dans une approche par preuves, le modèle est décrit au moyen d'axiomes, les propriétés sont des théorèmes qui sont alors vérifiés par un prouveur.

Dans une approche par parcours du graphe d'états, le modèle est exprimé dans un langage formel dont on peut déduire un modèle d'exécution exhaustif. Ceci requiert généralement une définition mathématique rigoureuse. Un « moteur d'exécution » produit alors l'espace d'états complet associé au système sous forme d'un graphe où les actions sont associés aux états du système. Il est alors possible d'explorer le graphe et vérifier si une propriété est satisfaite. Ces deux approches sont complémentaires :

Les approches basées sur les preuves permettent l'analyse de systèmes dont l'espace d'états est infini. Cependant, l'utilisation d'un prouveur est une tâche technique qu'il est difficile d'automatiser.

Au contraire, le *model checking* est *a priori* dédié aux systèmes finis, cependant la plupart des étapes sont complètement automatisées [CLA 00], ce qui facilite leur utilisation par un utilisateur qui n'est pas expert dans le domaine. Par exemple, l'outil Qasar [EVA 03] permet de construire un réseau de Petri à partir d'un code source et analyser certaines de ces propriétés élémentaires.

Nous avons retenu les réseaux de Petri symétriques[2] [CHI 91, CHI 93] comme langage pour la modélisation. Cette famille de réseaux de Petri permet la modélisation à haut niveau. Ce formalisme permet de prendre en compte le typage des entités, et de définir des ensembles d'entités de cardinalité finie. Ceci permet une définition concise et paramétrée du système, tout en préservant sa sémantique.

2. Ce type de réseaux de Petri a été longtemps connu sous le vocable « réseaux de Petri biens formés », le nom « réseaux symétriques » provenant de la normalisation ISO des réseaux de Petri.

Un des principaux bénéfices des réseaux de Petri symétriques est qu'ils permettent la construction automatique et efficace du *« graphe des marquages accessibles symboliques »*. Il s'agit d'un espace d'états quotient dans lequel les nœuds sont des classes d'équivalence entre états concrets, et les arcs des classes d'équivalence entre évènements [CHI 91]. Ce graphe est construit en exploitant les symétries du modèle, ce qui conduit à un graphe des marquages qui est potentiellement exponentiellement plus petit que le graphe des marquages classiques, et donc plus simple à manipuler.

Des travaux récents ont automatisé l'analyse des symétries admises par un système [THI 03], et des symétries d'une propriété [BAA 04]. Ces techniques, couplées à des outils de vérification automatique, nous autorisent une vérification du modèle par des propriétés de logique temporelle LTL, tout en combattant efficacement l'explosion combinatoire de l'espace d'états inhérente à ce type d'analyse et en limitant ses effets. Les techniques utilisées sont présentées dans le chapitre 8 du présent ouvrage.

Comme nous le montrons dans les sections 11.4.4 et 11.4.5, l'utilisation de ces réductions basées sur les symétries nous permet de vérifier les propriétés de notre modèle, alors que les techniques classiques basées sur une exploration exhaustive de tous les états du système est impossible.

11.4. Vérification d'instances d'intergiciels

Dans cette section, nous présentons les différentes étapes qui nous ont permis de vérifier certaines propriétés comportementales clés d'instances d'intergiciels, construites suivant le modèle exposé à la section précédente.

11.4.1. *Processus de modélisation*

Nous décrivons maintenant la phase de modélisation de notre architecture en utilisant le formalisme des réseaux de Petri comme langage de modélisation formelle. Les différentes étapes sont représentées figure 11.2.

- *Étape 1* : Les différentes entités de l'intergiciel sont isolées, et ramenées à un code source de petite taille, moins de deux cents lignes significatives pour chaque fonction. Cette phase utilise les résultats sur la réduction des fonctions de l'intergiciel, et la définition du composant μBroker présenté section 11.2

- *Étape 2* : Nous construisons un réseau de Petri pour chaque implantation d'un élément de l'intergiciel. Les transitions du réseau de Petri représentent des actions atomiques ; les places sont soit des états de l'intergiciel, soit des ressources. Les interactions entre ces blocs sont représentées par des places communes à plusieurs réseaux, qui fonctionnent comme des *« places-canaux »* [SOU 89].

- *Étape 3* : Nous sélectionnons, pour une configuration du μBroker, plusieurs réseaux de Petri que nous assemblons afin de construire le modèle complet. Chacun des

modèles sélectionnés correspond à une variation d'une des fonctions de l'intergiciel. Les places de communications (représentées en noir) représentent les liaisons vers d'autres fonctions du μBroker, ou d'autres fonctions de l'intergiciel.

- *Étape 4* : Les modules sélectionnés sont fusionnés pour produire un modèle global en reliant les interfaces des différents composants modélisés. Ce modèle et un marquage initial permettent la vérification des propriétés de la configuration du μBroker ainsi modélisée.

Cette construction par étapes du modèle permet de vérifier séparément chacune des fonctions avant de les intégrer dans le modèle global. Par ailleurs, plusieurs modèles globaux peuvent être construits à partir d'une même bibliothèque de modèles. Ainsi, nous pouvons tester et vérifier des conditions d'exécution spécifiques représentées par la mise en place de certaines politiques (représentées par des modèles élémentaires) ou des états précis (sélection du marquage initial).

Le marquage initial du réseau de Petri définit les ressources disponibles (nombre de tâches, de canaux de communications) ou l'état interne du système. Ceci fournit un paramétrage du modèle, et permet de tester plusieurs configurations facilement. Les classes du système fournissent un moyen simple de tester l'impact de ressources (par exemple des tâches) sur l'exécution du système.

Nous avons défini un marquage initial définissant le nombre de tâches et de requêtes pouvant circuler dans le modèle. Tâches et requêtes peuvent passer à tout moment d'un état libre, hors du contrôle de l'intergiciel, à un état où elles ont un rôle à jouer. Ceci nous permet de rendre compte du caractère dynamique de l'arrivée des requêtes sur un nœud de l'application.

L'espace d'états construit à partir du marquage initial couvre tous les entrelacements possibles d'actions atomiques : nous examinons ainsi tous les ordonnancements possibles. Cette couverture de l'ensemble des états atteignables du système nous permet de garantir que les propriétés vérifiées restent valides quelque soit la plate-forme d'exécution utilisée et quelque soit l'ordonnancement imposé. Ces résultats sont réutilisables.

11.4.2. *Un modèle particulier : lecture des sources*

Nous détaillons ici un des modèles construits (figure 11.3). Ce modèle prend en charge la lecture des données entrantes et leur transmission à la couche neutre.

Cette procédure est déclenchée lorsqu'une tâche est bloquée, en attente d'évènements, et est composée de deux phases : *1)* attente sur les sources d'évènements et *2)* traitement des évènements.

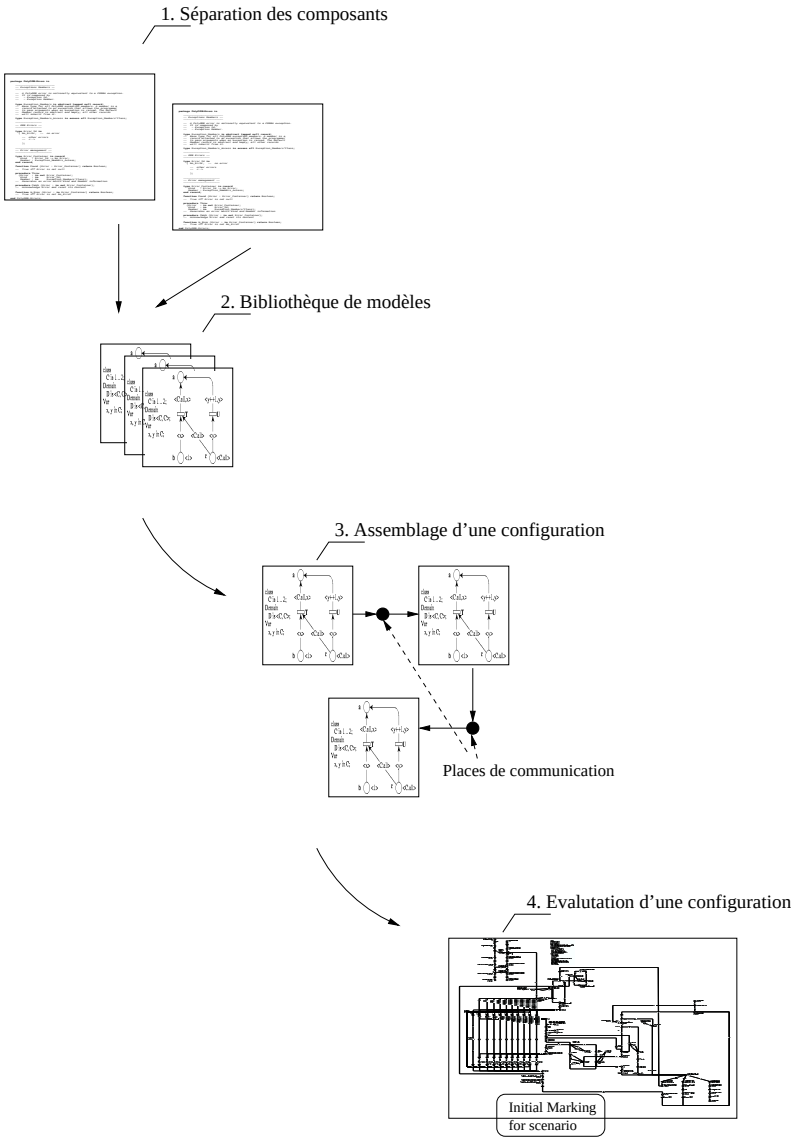


Figure 11.2. *Étapes de modélisation*

– **Attente sur les sources d'évènements :** La place *ThreadPool* contient toutes les tâches affectées à l'écoute des sources d'évènements. Par construction du système, au plus une tâche devrait occuper cette place. Lorsque la transition *ChckSrcB* est tirée,

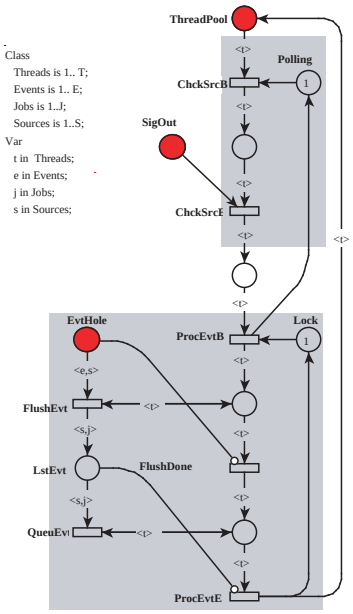


Figure 11.3. *Un modèle de la bibliothèque*

une tâche est sélectionnée pour entrer en phase d'attente. La transition *ChckSrcE* peut ensuite être tirée lorsqu'un évènement est détecté ((place *SigOut*). La place *Polling* joue le rôle de barrière et assure qu'au plus une tâche est en phase d'écoute.

– **Traitement des évènements :** Les évènements sont lus depuis la place *EvtHole*. Cette lecture doit se faire atomiquement (place *Lock*). La transition *FlushEvt* est tirée autant de fois qu'il y a d'évènements à traiter. La transition *FlushDone* est tirée lorsque tous les évènements sont consommés (arc inhibiteur entre *EvtHole* et *FlushDone*). Les évènements sont stockés dans une file, ils seront traités par la suite par une autre fonction, définie dans un autre modèle. Lorsque tous les évènements ont été traités, la transition *ProcEvtE* est tirée et libère le verrou. La tâche est alors libérée.

Les places représentées en noir ont un rôle spécial vis-à-vis du module : ils représentent les interactions avec d'autres modules. Leur marquage est généré par ces modules, et assurent une connexion entre les différents modules. Ils représentent soit des connections avec d'autres fonctions de l'intergiciel, soit la réception d'évènements transmis par d'autres composants.

11.4.3. *Configurations du μBroker et modèles*

Dans cette section, nous passons en revue les paramètres clés qui caractérisent le μBroker, et les propriétés attendues de ce composant.

Une configuration du μBroker est définie par l'ensemble des politiques et des ressources qu'elle utilise. Ces paramètres sont communs pour une large classe d'applications. Nous considérons par la suite une instance d'un intergiciel, en mode serveur, qui traite toutes les requêtes entrantes. Nous étudions deux configurations du μBroker : *Mono-tâche*, disposant d'une tâche principale fournie par l'environnement d'exécution ; *Multi tâches*, plusieurs tâches s'exécutant en parallèle, et coordonnées suivant la politique *« Leader/Followers »* [PYA 01].

Nous faisons l'hypothèse que les ressources utilisées par l'intergiciel sont allouées à l'initialisation du nœud. Nous considérons donc que l'intergiciel dispose d'un pool de tâches pré-allouées ; un nombre fixé de canaux de communications, représentées du point de vue du nœud modélisation par des sources d'évènements ; et une zone mémoire où stocker les requêtes en cours de traitement. Cette hypothèse est raisonnable et correspond aux techniques d'ingénierie logiciel pour les systèmes critiques. Notre implantation et le modèle correspondant sont contrôlés par trois paramètres :

S_{max}	nombre de sources d'évènements
T_{max}	nombre de tâches disponibles
B_{size}	la taille du tampon mémoire alloué

S_{max} and T_{max} définissent un profil de charge pour le nœud (nombres de requêtes et d'évènements à traiter) ; B_{size} impose une contrainte sur la mémoire allouée par le μBroker pour traiter des requêtes. Ces paramètres contrôlent l'exécution correcte de l'intergiciel, et vont aussi définir sa « bande passante ».

Nous définissons quatre propriétés fondamentales pour le μBroker :

$P0$ (symétrie)	les tâches et sources d'évènements ne sont pas ordonnées
$P1$ (pas d'interblocage)	le système peut toujours traiter de nouvelles requêtes
$P2$ (cohérence)	il n'y a pas de corruption de la mémoire
$P3$ (équité)	tout évènement détecté sur une source sera traité

Notons que $P0$ n'est pas en soit une propriété déterminante du système. Il s'agit cependant d'une propriété structurelle forte. Elle fournit une indication de la cohérence du modèle construit. On s'attend en effet à ce que les tâches et les sources

d'évènements soient interchangeables. Cette propriété nous permet d'utiliser les techniques d'analyse basées sur le graphe quotient, ceci nous permet de réduire grandement l'analyse du modèle.

$P0, P1, P3$ sont difficiles à valider même si l'on se limite à l'exécution de quelques scénarios : il faudrait être certain de couvrir tous les états du système. Cela peut ne pas être faisable techniquement en un temps limité, ni même possible à cause de l'entrelacement des tâches et des requêtes et du contrôle qu'exerce le système d'exécution sur l'ordonnancement du système.

Par ailleurs, le dimensionnement adéquat de ressources assurant le fonctionnement correct du système ($P2$) est un prérequis fort pour les systèmes répartis, mais aussi un problème difficile pour un système ouvert tel qu'un nœud d'une application répartie.

Nous présentons dans la section suivante la vérification de ces propriétés par *model checking*.

11.4.4. *Méthodes d'analyse*

Le système que nous modélisons est complexe, et coordonne plusieurs ressources. Nous nous attendons à ce que son espace d'états soit large. Nous détaillons ces chiffres à la section 11.4.5 : sa taille peut dépasser 10^{11} états pour des configurations raisonnables, ce qui rend ce système difficilement analysable par des outils classiques.

Ainsi, nous avons décidé de réduire la complexité algorithmique du problème à résoudre en utilisant des outils et méthodes développés dans l'équipe MoVe du LIP6.

Nous avons dans un premier temps effectué une analyse des symétries du modèle de façon à réduire le nombre d'exécutions du model checker. Elle nous permet aussi de construire le graphe des marquages symboliques. Cette analyse des symétries est complètement automatisée. Elle a été réalisée automatiquement pour chaque configuration du μBroker étudiée.

Enfin, nous prenons aussi en compte la propriété à vérifier, ce qui permet d'avantage de réductions de l'espace d'états par les outils utilisés. Ceux-ci ne considèrent alors que les comportements qui sont pertinents pour la propriété à vérifier, en se basant sur des algorithmes de parcours du graphe d'états adaptés à la propriété à vérifier.

Les outils utilisés ont joué un rôle crucial pour compléter ce travail de vérification et exploiter le travail de modélisation que nous avons effectué. Les risques d'explosion combinatoire sont élevés pour ce type de modèle. Ceci peut mettre en évidence les limites des outils utilisés, et rendre vain l'effort de modélisation, et impossible toute vérification.

– Nos modèles ont été construits en utilisant l'outil CPN-AMI [KOR 99], un environnement pour la construction et l'analyse de Réseaux de Petri, qui fédère de nombreux outils pour l'analyse structurelle et le *model checking*.

– Les symétries ont été calculées automatiquement en utilisant les outils de Yann THIERRY-MIEG [THI 03].

– L'analyse de l'espace d'états a été effectuée en utilisant une variation des bibliothèque du model checker GreatSPN [INF 03]. Elles intègrent les algorithmes exploitant les propriétés de symétries des modèles, et les techniques basées sur le produit synchronisé.

– La bibliothèque SPOT [DUR 04] a fourni une interface simple pour tous ces outils, masquant ainsi la complexité des algorithmes mis en œuvre, en particulier les formalismes utilisés lors des différentes phases de calculs intermédiaires.

P0 (symétrie) : Analyser les symétries du modèle. Nous souhaitons vérifier la propriété $P0$ (symétrie) ; la première étape est une analyse des symétries du système par une exploration de la structure du modèle. Ceci permet de déterminer les types de données présentant un comportement homogène.

Deux éléments e_1 et e_2 d'un type de base sont *« symétriques »* si échanger e_1 et e_2 lors de l'exécution du système ne modifie pas son comportement du point de vue de l'*observateur*.

L'algorithme présenté dans [THI 03] recherche les symétries d'un système en examinant chaque action (transition) du système et détermine quels sont les éléments distingués (si ils existent), et conclut que tout élément distingué par au moins une action doit être distingué par la relation d'équivalence $\mathbb{R}$ construite. $\mathbb{R}$ liste toute les valeurs symétriques pour le langage des actions du système.

Ce calcul automatique assure que tous les éléments équivalents pour la relation $\mathbb{R}$ ne seront pas distingués lors de l'exécution du système. $\mathbb{R}$ définit un ensemble de permutations et de rotations admissibles [CHI 91] qui peuvent être appliquées au système sans affecter les transitions de celui-ci.

Les symétries d'un modèle dépendent uniquement de sa structure. La complexité de ce calcul dépend de la taille du modèle (nombre de places et de transitions), ce qui rend ce calcul relativement rapide comparativement à un parcours de l'espace d'états associé au modèle. Le graphe quotient des marquages accessibles pour $\mathbb{R}$ est construit en utilisant les bibliothèques de GreatSPN [INF 03]. Les nœuds de ce graphe sont des classes d'équivalence d'états pour la relation $\mathbb{R}$.

L'analyse des symétries du μBroker montre que les tâches et sources d'évènements forment deux classes d'équivalence. Ainsi, nous vérifions $P0$ (symétrie).

Notons que cette propriété structurelle dépend autant du système à modéliser que des abstractions utilisées. Une première version de nos modèles utilisait une classe dédiée au comptage des requêtes stockées dans la file, conduisant à la construction d'une sous-classe statique ordonnée par ce compteur. Cette structure introduisait une asymétrie *« parasite »* induite par la modélisation et non le système à modéliser. Les outils nous ont permis de localiser l'asymétrie, que nous avons pu lever pour tirer partie de ces méthodes.

P1 (pas d'interblocage) : Analyse du graphes des marquages. La propriété $P1$ indique qu'il n'existe pas d'état d'interblocage.

Cette propriété peut-être vérifiée sur le graphe des marquages associé au modèle, en recherchant la présence d'éventuels nœuds terminaux, indiquant un état où le système cesse d'évoluer.

Cette analyse tire profit du graphe quotient pour rechercher les nœuds terminaux. Nous vérifions l'absence de tels nœuds pour les modèles considérés.

P2 (cohérence) : Vérification d'une propriété symétrique. Pour vérifier la propriété $P2$ (*cohérence*), nous devons vérifier que les accès à la zone mémoire sont corrects. Les places $(\mathcal{D}ata\mathcal{S}lots_i)_{i\in 1..M}$ de notre modèle représentent cette zone mémoire sous la forme de M « cases ». L'opération d'écriture insère un jeton (une donnée) dans cet ensemble, une opération de lecture en supprime un.

Une erreur mémoire apparaît lorsque l'on tente d'écrire deux fois de suite dans une même zone mémoire.

Ceci peut être testé grâce à la propriété de sûreté exprimée par la formule de logique temporelle LTL (11.1), qui affirme qu'un tel état n'est pas atteignable.

$$\forall d \in \mathcal{D}ata\mathcal{S}lots, \mathsf{G}(card(d) \leq 1) \tag{11.1}$$

Nous remarquons que cette propriété est directement observable sur le graphe quotient produit pour la relation $\mathbb{R}$: les permutations d'éléments ne changeront pas le cardinal de la place testé par $P1$. Ainsi, si on considère un nœud de l'espace d'états quotient, alors tous ces éléments seront équivalents pour $\mathbb{R}$, et nous pouvons tester si les places vérifient $card(d) \leq 1$. La vérification de cette formule utilise automatiquement cette analyse des symétries pour réduire la complexité de l'analyse.

Ainsi, nous vérifions qu'il n'y a pas de corruption de données pour $S_{max} \leq B_{size}$ pour différentes valeurs de S_{max} et B_{size}. Nous notons qu'il peut y avoir corruption de données pour d'autres valeurs.

Par ailleurs, ce résultat peut aussi être interprété comme suit : le modèle ne permet pas plus de $card(\mathcal{D}ata\mathcal{S}lots)$ opérations d'écriture successives, et donc ne sature pas la zone mémoire. De sorte que la vérification de $P2$ implique aussi que la propriété $P2'$ (pas de saturation mémoire) est vraie.

Cette analyse nous permet de lier la taille du tampon mémoire utilisé au nombre de sources d'entrées/sorties du nœud. Ceci nous fournit une indication sur la configuration de l'intergiciel à utiliser pour l'application.

P3 (équité) : Utilisation des symétries du comportement observé. La relation d'équivalence $\mathbb{R}$ que nous avons introduite à la section précédente, bien que permissive, permet de vérifier des propriétés symétriques efficacement, en réduisant l'espace d'états à explorer.

La vérification de $P3$ nécessite que tout évènement détecté pour la source s sera finalement traité par le système. Pour ce faire, le jeton représentant s doit aller de la place indiquant la présence d'un évènement `ModifiedSrc` (« le μBroker a détecté une requête en attente ») à la place initiale `DataOnSrc` (« nouvelle requête entrante »). Ceci amène donc à l'évaluation de la formule LTL suivante :

$$\forall s \in \mathcal{S}ources, \mathsf{G}(\{s\} \subseteq \texttt{ModifiedSrc} \Rightarrow \mathsf{F}(\{s\} \subseteq \texttt{DataOnSrc})) \tag{11.2}$$

La vérification de la formule $P3$ (*équité*) impose d'isoler la variable s lors du parcours du graphe des marquages. Ainsi, on associe à la variable s l'action « le μBroker a détecté une requête en attente sur la source s » pour suivre le cheminement de s à travers l'intergiciel. Il nous faudrait ainsi vérifier $P3$ pour chacune des sources.

La propriété $P0$ (*symétrie*) nous permet d'affirmer, puisque les sources d'évènements sont équivalentes, le résultat suivant :

$$\exists s \in \mathcal{S}ources / P3(s) \Rightarrow \forall s \in \mathcal{S}ources, P3(s) \tag{11.3}$$

Ainsi, la proposition 11.3 nous indique qu'il suffit de vérifier la propriété pour une des sources ; on a alors la garantie qu'elle sera vraie par extension pour toutes les autres sources. Cela réduit le nombre d'invocation au model checker d'un facteur $card(\mathcal{S}ources)$.

Nous choisissons une source s que nous individualisons, en construisant un graphe quotient sous une relation d'équivalence $\mathbb{R}'$ plus restrictive, telle que s ne puisse être

permuté avec une autre source. ainsi, l'observateur LTL peut isoler les évènements liés à s, et les distinguer des évènements liés aux autres sources.

Cependant, distinguer une source revient à considérer de nouveaux états pour chaque source isolée pour chaque nœud du graphe quotient initial, sous la relation $\mathbb{R}$.

Ainsi, même pour un modèle symétrique, l'analyse d'une formule non symétrique conduit à nouveau à une explosion combinatoire. Elle est seulement retardée par rapport à une analyse classique. Dans le pire des cas, le graphe d'états peut avoir la même taille que le graphe d'états initial, on perd donc le bénéfice de ces approches.

Les techniques d'analyse basées sur l'utilisation du graphe quotient d'accessibilité souffrent d'une limitation forte : l'incapacité à analyser des systèmes partiellement symétriques, ou de vérifier des propriétés partiellement symétriques.

Pour contourner ces problèmes, nous employons une autre technique d'analyse, elle aussi développée au LIP6/MoVe : le produit synchronisé symbolique (*« Symbolic Synchronized Product »* (SSP)) [BAA 04].

L'idée de cette méthode est de modifier l'observateur LTL pour que celui-ci s'adapte à la volée à la propriété à vérifier. L'observateur ne distingue que les états pertinents pour vérifier une propriété, et agrège les autres états. Ainsi, l'observateur « zoome » sur les états devant être distingués, et ne conserve qu'une vision floue des autres états.

Par exemple, la propriété peut distinguer un objet seulement sur une partie de son exécution, comme pour la propriété $P3$. Les sources peuvent être permutées lorsqu'il n'y a pas de requêtes en attente. Nous n'avons besoin de distinguer la source s que lorsque la place `ModifiedSrc` contient une requête sur la source s.

Les algorithmes SSP fournissent un outil puissant, capable de tirer partie des propriétés des systèmes partiellement symétriques, dans lesquels les objets ont un comportement asymétrique seulement sur une partie restreinte du modèle. Ceci permet de réduire considérablement l'espace d'états du modèle.

Ces algorithmes nous ont permis de vérifier la validité de la propriété $P3$ pour les différents modèles considérés.

11.4.5. *Résultats*

Le modèle *Mono-Tasking* a 47 places, 38 transitions et 134 arcs. Le modèle *Multi-Tasking* est deux fois plus large. Le nombre de tâches et de sources n'a aucun effet sur la taille du modèle : ils sont codés comme des types de jetons dans le réseau de Petri.

Le tampon est codé explicitement dans le modèle, sous la forme de N cases, sa taille a donc un impact sur le nombre de places et de transitions du modèle.

La figure 11.4 fournit la taille de l'espace d'états pour un tampon de taille B_{size}= 5, et S_{max}= 4 sources. Les modèles analysés ont une taille raisonnable. Cependant, l'espace d'états est grand, jusqu'à 10^{11} états pour les configurations testées. Ceci rend compte de la complexité du système, et du fort entrelacement entre tâches et sources. Sa taille croit exponentiellement avec T_{max} et S_{max}. Remarquons aussi que la taille du graphe d'états symboliques est inférieure de plusieurs magnitudes à celle du graphe concret, et que cette différence croit avec T_{max}.

Ce gain provient du graphe quotient ; chaque état représente jusqu'à $S_{max}!\times T_{max}!$ permutations de manière compacte. Il est donc exponentiellement plus petit que l'espace d'états initial, permettant une vérification accélérée de cette propriété.

Ces gains en espace se répercutent sur l'utilisation mémoire et CPU. Les calculs ont été réalisés dans un temps raisonnable, sur des stations de travail classiques. Ainsi, les calculs les plus longs requièrent moins d'une dizaine d'heures pour les configurations simples, jusqu'à un jour et demi sur des configurations complexes, sur un Pentium-IV cadencé à 3.2GHz, disposant de 3Go de mémoire, sans recours à la mémoire tampon.

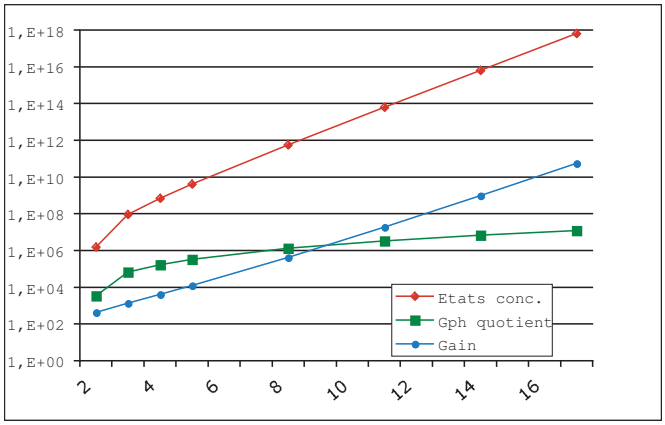


Figure 11.4. *Espace d'états et espace quotient du µBroker pour* S_{max}= 4 *et* B_{size}= 5

Importance des outils. Ces différentes mesures montrent que l'analyse de PolyORB n'aurait sans doute pas pu être menée sans l'utilisation de ces techniques avancées de *model checking* : l'espace d'états à parcourir est trop grand pour les stations de travail actuelles comme le montre la figure 11.4. On estime en effet qu'un model

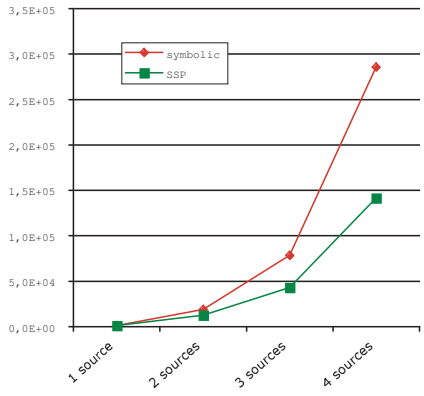


Figure 11.5. *Nombre de nœuds examinés pour évaluer* $P3$, $T_{max}= 2$ *et* $B_{size}= 4$

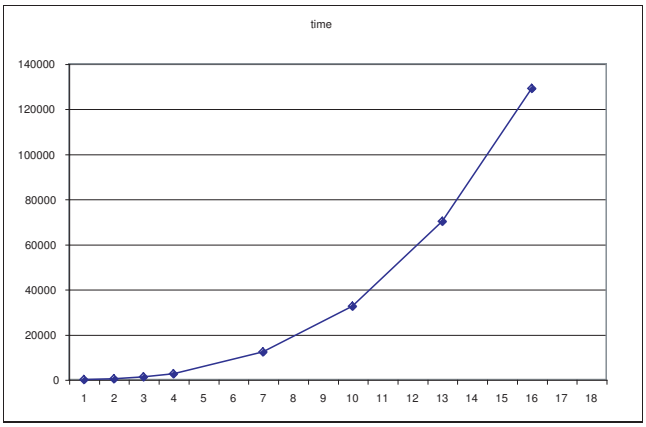


Figure 11.6. *Temps pour la vérification d'une configuration, en secondes*

checker devient inefficace à partir de 10^8 états du fait de l'encombrement mémoire, et le recours à la mémoire tampon qui réduit considérablement les performances.

Nous notons un gain exponentiel entre l'espace d'états concret et l'espace quotient qui lui est associé. Comme illustration, remarquons que pour une configuration de l'intergiciel raisonnable (sept tâches et quatre sources), le système présente 10^{17} états, mais il a pu être analysé en parcourant un graphe quotient contenant 10^7 états.

Nous remarquons aussi que le graphe quotient croit moins vite que le graphe concret. Ainsi, il croit de 80% lors du passage de quatre à cinq tâches, de 57% pour cinq à six. Ceci est intéressant lors de l'analyse de grands systèmes, et à mettre en perspective avec le graphe concret qui croit exponentiellement.

La technique basée sur le produit synchronisé (SSP) fournit d'avantage de réductions pour les systèmes partiellement symétriques comme le montre la figure 11.5. Elle détaille le gain obtenu lors de la vérification de la propriété $P3$, et en particulier l'augmentation de ce gain lorsque les paramètres varient. Nous ne fournissons aucune valeur pour $S_{max} \geq 5$; $B_{size} = 4$ car nous savons alors que $P1$ n'est plus vérifiée. Il est alors inutile de vérifier cette configuration.

Apports du travail de modélisation et de vérification. Des outils récents pour le *model checking* nous ont permis de réduire la complexité du temps de calcul nécessaire à la vérification. Cependant, l'utilisation d'outils combattant l'explosion combinatoire efficacement n'est qu'un aspect pratique de la vérification.

La réalisation de modèles conformes au système initial est une phase critique et difficile qui ne peut être automatisée. Comme nous l'avons indiqué lors de la vérification de la propriété $P0$, un modèle mal conçu peut rendre inopérantes certaines optimisations. Par ailleurs, il est nécessaire de mettre en œuvre une démarche rigoureuse de modélisation garantissant que le système modélisé et son modèle sont cohérents.

Nous avons effectué le travail de modélisation en parallèle au travail de définition et de conception des différents composants formant le μBroker : nous avons procédé à une définition précise de chacun, puis nous les avons réduits de façon à n'en conserver qu'une définition significative pour ensuite les modéliser à l'aide de réseaux de Petri. Ce travail de *« co-design »* entre le modèle et son implantation nous garantissent que le modèle et le composant modélisé sont proches sémantiquement.

Les propriétés que nous avons vérifiées suivant ce processus fournissent une preuve forte que l'architecture et l'implantation que nous en proposons sont correctes.

11.5. Conclusion

Dans ce chapitre, nous avons présenté les travaux de vérification formelles de notre architecture. Nous avons construit plusieurs modèles représentant le plus fidèlement possible la sémantique de plusieurs configurations du μBroker. Nous avons ensuite vérifié certaines de leur propriétés (absence d'interblocage, de famines, etc.) à l'aide de formules de logique temporelle.

Ces travaux démontrent par l'exemple qu'une validation de ces propriétés à l'aide de tests auraient été difficilement possible, et certainement moins fiable. Des paramètres tels que l'ordonnanceur du système d'exécution utilisé, la fréquence du processeur ou le nombre de tâches ou d'entrées/sorties rendent certains états non atteignables par la seule expérience. La couverture des états du système serait donc incomplète.

Ainsi, nous avons modélisé plusieurs entités formant le cœur de l'intergiciel, dont nous avons isolé le code source pour ensuite le modéliser. Nous réduisons ainsi la distance entre le système et son modèle. Ce travail repose sur l'identification du μBroker et ses différentes politiques et leur réduction à des entités simples. Nous proposons ainsi une modélisation complète d'un intergiciel, exploitable. Le modèle est en liaison directe avec l'implantation.

Les modèles construits sont de taille réduite, et peuvent être vérifiés séparément pour s'assurer de leur cohérence avant de les assembler. Plusieurs modèles ont été assemblés et vérifiés. Chacun correspond à une instance d'intergiciel. Cette méthode a pour avantage de reproduire au niveau modélisation ce qui est réalisé à l'échelle du code source et lors de la phase de compilation.

Nous avons par ailleurs montré que certaines fonctions de l'intergiciel pouvaient être réduites à des abstractions classiques, réduisant ainsi la taille des modèles à considérer. De fait, il suffit de modéliser le μBroker et certaines de ses politiques pour pouvoir analyser une instance pour les propriétés que nous avons considérées.

Les propriétés démontrées pour une configuration sont aussi valides pour toute autre configuration compatible, par exemple utilisant une personnalité protocolaire proche (IIOP ou SOAP). Ce faisant, nous étendons la portée des propriétés prouvées à une large classe de configurations.

Compte tenu des résultats obtenus lors de ce travail de vérification, nous apportons les contributions suivantes :

1) Nous avons défini un processus de modélisation de l'intergiciel complet. Il isole les blocs nécessaires à l'évaluation de ses propriétés comportementales.

2) Nous avons modélisé plusieurs configurations d'intergiciels en associant un modèle élémentaire et une portion de code source. La distance entre modèle et implantation est donc réduite.

3) Nous avons utilisé des techniques nouvelles de vérification, qui permettent de réduire l'explosion combinatoire. Nos travaux ont en outre permis de valider sur des exemples concrets ces techniques.

4) Les propriétés vérifiées fournissent une base solide pour la construction d'intergiciels fiables et prouvés.

Nos travaux ont montré la pertinence de l'architecture proposée, et son apport bénéfique pour la modélisation d'un intergiciel à des fins de vérification. Nous avons pu opérer la vérification de propriétés fondamentales de l'intergiciel de façon automatique en faisant certains assemblages de modèles manuellement.

Ces travaux de modélisations ont fourni des résultats nouveaux aux deux communautés intergiciel et vérification. En effet, modéliser à l'aide de réseaux de Petri le cœur d'un intergiciel est un processus complexe qui fournit une vue originale de l'intergiciel et permettant une analyse précise. Par ailleurs, nos travaux ont fourni un modèle original pour mettre à l'épreuve les outils de vérification et ainsi expérimenter leurs techniques sur des modèles moins classiques. Ce faisant, nous contribuons aussi à la mise au point des algorithmes et outils de vérification développés autour du projet CPN-AMI.

Une évolution naturelle serait d'enrichir PolyORB d'une description architecturale grâce à un Langage de description d'Architecture[3]. Cette description et une bibliothèque de modèles permettraient de simplifier grandement l'évaluation de plusieurs configurations, et pousser plus en avant les travaux que nous avons réalisés. Des propositions dans ce sens, à base du langage AADL (pour « *Architecture Analysis & Design Language* ») sont effectuées dans [VER 05].

Une seconde évolution serait d'étendre le processus de vérification à l'ensemble de l'application répartie. Comme le montre [GIL 03], la construction d'applications réparties et le choix de certaines politiques de configuration peut conduire à l'épuisement de ressources sur un nœud, conduisant au blocage d'autres nœuds. Ces situations résultent d'une erreur de conception du système qui doit être analysée à l'échelle de l'application. La validation de ce type de système devra mettre en œuvre un processus de vérification adapté, prenant en compte les échanges entre nœuds.

11.6. Bibliographie

[ABR 95] ABRIAL J., *Z : an introduction to formal methods*, Cambridge University Press, 1995.

[BAA 04] BAARIR S., HADDAD S., ILIÉ J.-M., « Exploiting Partial Symmetries in Well-formed nets for the Reachability and the Linear Time Model Checking Problems », *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, septembre 2004.

[CHI 91] CHIOLA G., DUTHEILLET C., FRANCESCHINI G., HADDAD S., « On Well-Formed Coloured Nets and their Symbolic Reachability Graph », *High-Level Petri Nets. Theory and Application, LNCS*, Springer Verlag, 1991.

3. On parle aussi d'ADL pour « Architecture Description Languages »

[CHI 93] CHIOLA G., DUTHEILLET C., FRANCESCHINIS G., HADDAD S., « Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. », *IEEE Trans. Computers*, vol. 42, n°11, p. 1343-1360, 1993.

[CLA 00] CLARKE E., GRUMBERG O., PELED D., *Model Checking*, MIT Press, 2000.

[DIL 94] DILLER A., *The B-book*, John Willey & SONS, 1994.

[DUR 04] DURET-LUTZ A., POITRENAUD D., « SPOT : an Extensible Model Checking Library using Transition-based Generalized Büchi Automata », *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, Volendam, The Netherlands, IEEE Computer Society Press, p. 76–83, octobre 2004.

[EVA 03] EVANGELISTA S., KAISER C., PRADAT-PEYRE J., ROUSSEAU P., « Quasar : a new tool for analyzing concurrent programs », *Proceddings of the International Conference on Reliable Software Technologies, Ada-Europe (Ada-Europe'03)*, vol. LNCS 2655, Springer Verlag, p. 166-181, juin 2003.

[FEI 03] FEILER P., LEWIS B., VESTAL S., « The SAE Avionics Architecture Description Language (AADL) Standard : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering », *RTAS 2003 Workshop on Model-Driven Embedded Systems*, mai 2003.

[GIL 03] GILL C. D., « A Vision for Integration of Embedded System Properties - via a Model-Component-Aspect System Architecture », *Proceedings of the Monterey Workshop*, Chicago, IL, septembre 2003.

[HAL 93] HALBWACHS N., A tutorial of Lustre, 1993.

[HOL 04] HOLZMANN G., *SPIN Model Checker, The : Primer and Reference Manual*, Addison Wesley Professional, 2004.

[HUG 03] HUGUES J., PAUTET L., KORDON F., « Contributions to middleware architectures to prototype distribution infrastructures », *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, San Diego, CA, USA, IEEE, p. 124-131, juin 2003.

[HUG 04] HUGUES J., THIERRY-MIEG Y., KORDON F., PAUTET L., BAARIR S., VERGNAUD T., « On the Formal Verification of Middleware Behavioral Properties », *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, vol. ENTCS 133, Linz, Austria, Elsevier, p. 139 - 157, septembre 2004.

[INF 03] DI INFORMATICA UNIV. TORINO D., « GreatSPN home page, `http://www.di.unito.it/~greatspn/` », 2003.

[ISO 94] ISO, *Quality management and quality assurance - vocabulary*, ISO, 1994, ISO 8402 :1994.

[KOR 99] KORDON F., PAVIOT-ADET E., « Using CPN-AMI to validate a safe channel protocol », *Proceedings of the International Conference on Theory and Applications of Petri Nets - Tool presentation part*, Williamsburg, USA, juin 1999.

[MER 02] MERSEGUER J., CAMPOS J., BERNARDI S., DONATELLI S., « A compositional semantics for UML state machines aimed at performance evaluation », *Proceedings of the*

Sixth International Workshop on Discrete Event Systems, october 2002.

[PYA 01] PYARALI I., SPIVAK M., CYTRON R., SCHMIDT D. C., « Evaluating and Optimizing Thread Pool Strategies for RT-CORBA », *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, ACM, 2001.

[REG 01] REGEP D., KORDON F., « **L***f***P** : a specification language for Rapid prototyping of Concurrent Systems », *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.

[SHA 98] SHARP D. C., « Reducing Avionics Software Cost Through Component Based Product Line », *Proceedings of the 10th Annual Software Technology Conference*, avril 1998.

[SOU 89] SOUISSI Y., MEMMI G., « Compositions of Nets via a communication medium », *10th International Conference on Application and theory of Petri Nets*, Bonn, Germany, June 1989.

[THI 03] THIERRY-MIEG Y., DUTHEILLET C., MOUNIER I., « Automatic Symmetry Detection in Well-Formed Nets », *Proc. of ICATPN 2003*, vol. 2679 de *Lecture Notes in Computer Science*, Springer Verlag, p. 82–101, juin 2003.

[VER 04] VERGNAUD T., HUGUES J., PAUTET L., KORDON F., « PolyORB : a schizophrenic middleware to build versatile reliable distributed applications », *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, vol. LNCS 3063, Palma de Mallorca, Spain, Springer Verlag, p. 106 - 119, juin 2004.

[VER 05] VERGNAUD T., HUGUES J., PAUTET L., KORDON F., « Rapid Development Methodology for Customized Middleware », *Proceedings of the 16th International Workshop on Rapid System Prototyping*, IEEE Computer Society, p. 111-117, june 2005.

Chapitre 12

Une approche formelle de l'interopérabilité des services Web

Les services Web donnent actuellement lieu à des recherches très actives et ceci est dû à des facteurs à la fois industriels et théoriques. D'une part, les services Web s'imposent comme le modèle de conception des applications dédiées au commerce électronique. D'autre part, l'introduction de ce modèle a pour objectif la conception d'applications distribuées et coopératives dans un environnement ouvert (l'Internet). Dans ce chapitre, nous nous focaliserons sur le problème de l'interopérabilité des services Web et nous décrirons une solution à ce problème basée sur une approche de type sémantique formelle. Cette approche est intégrée dans un environnement de développement de services que nous décrirons de manière synthétique.

12.1. Introduction

L'intégration d'applications. Les entreprises d'aujourd'hui reposent sur des systèmes informatiques qui sont devenus névralgiques. A l'heure des fusions, des restructurations, des délocalisations, de la forte concurrence, des gammes de produits, les Systèmes d'Information (SI) doivent avoir la capacité à prendre en compte l'ensemble de ces contraintes économiques. Ce qui se traduit par des besoins de flexibilité, d'adaptabilité, d'ouverture et même d'interopérabilité entre SI distants et/ou hétérogènes, c'est-à-dire basés sur des socles techniques différents. L'interopérabilité suppose que les applications soient capables de se localiser, de s'identifier, d'exposer les fonctionnalités (services) qu'elles offrent, et enfin, d'échanger des données.

Chapitre rédigé par Céline BOUTROUS SAAB et Serge HADDAD et Valérie MONFORT.

Ces échanges, longtemps d'application à application (point à point), se sont traduits par des coûts maintenant prohibitifs liés à la maintenance des interconnexions entre applications. Cette approche est remplacée, peu à peu, par des bus applicatifs, appelés bus ESB (Bus de Services d'Entreprise), connectant des applications et des composants logiciels. L'utilisation d'un bus suppose d'une part l'implémentation des flux d'information et des données dans des référentiels, et d'autre part, le développement ou la spécialisation de connecteurs applicatifs. Toutefois, si ce modèle répond au problème d'intégration de SI locaux (moyennant un coût significatif), il ne permet pas, à lui seul, de relier des SI distants. Pour ce faire, les services Web se présentent aujourd'hui comme la solution la plus appropriée afin d'interconnecter des SI distants, éventuellement hétérogènes.

Les Architectures Orientées Services (SOA), d'abord basées sur les composants et leur capacité à communiquer au travers de leurs interfaces, ont vite montré leurs limites en termes de couplage faible (nécessaire pour répondre aux besoins de flexibilité et d'adaptabilité) et d'interopérabilité. Les services Web apportent ces propriétés aux SOA qui faisaient justement défaut aux architectures à base de composants. Les services Web reposent sur des normes pour l'échange d'informations ainsi que sur des protocoles pour leur transport.

Des services Web élémentaires aux services complexes. Les services Web sont « des applications modulaires, autonomes et décrivant leurs fonctionnalités qui peuvent être publiées, recherchées et invoquées à travers le Web » [TID 00]. Ils sont basés sur un ensemble de standard des environnements ouverts afin de rassembler le plus d'acteurs possibles du marché. Leur environnement se décompose en trois domaines : les protocoles de communication, la recherche de services et la description de leurs fonctionnalités. C'est ce dernier point qui est l'objet de notre chapitre. Le langage WSDL [WSD 01] (« *Web Services Description Language* ») offre une description formelle des services susceptibles d'être analysée par un programme. Une telle description spécifie les interfaces des composants logiciels en énumérant l'ensemble des opérations qui sont accessibles à travers le réseau *via* des messages XML. Elle inclut toute l'information dont une application a besoin pour invoquer le service : la structure des messages et des réponses ainsi que le protocole de transport, l'adresse, le port, etc.

Cependant l'invocation d'opérations est insuffisante pour dialoguer avec un service complexe. Celui-ci requiert de conserver une « mémoire » des interactions passées afin d'en déduire les interactions à venir. Ce type de service se rencontre dans deux cas de figure : d'une part lorsque le service Web est décrit par un modèle explicite d'exécution et, d'autre part, lorsqu'un service est bâti par composition d'autres services. Pour prendre en compte les aspects comportementaux des services complexes, des propositions de langage sont apparues provenant à la fois du monde industriel et académique. Parmi celles-ci, « *Business Process Execution Language for Web Services* » (BPEL4WS ou plus succinctement BPEL) développé par des acteurs de l'industrie (BEA, IBM et Microsoft) est rapidement devenu un standard [AND 03].

Les deux facettes des services Web complexes. BPEL offre deux types de processus (voir par exemple [JUR 05a], [JUR 05b]) :

– Un processus exécutable spécifie de manière exhaustive et détaillée le processus métier correspondant. Il peut être exécuté sur une machine par un « interpréteur » de services.

– Un processus abstrait (en réalité un protocole) spécifie l'échange de messages entre le client et le service. Il n'inclut pas les aspects internes de l'exécution du processus métier mais permet *a priori* au client d'interagir correctement avec le service.

Partant d'une description de processus exécutable, le processus abstrait correspondant est obtenu par un mécanisme d'abstraction qui masque les opérations internes du service. Cependant les problèmes soulevés par ces deux types de processus sont de nature très différente.

– La spécification d'un processus exécutable est proche de la définition d'un programe et soulève le problème de *l'expressivité du langage*.

– La spécification du protocole d'interaction soulève quant à elle le problème de *la synthèse d'un client* capable de respecter le protocole dans tous les scénarios possibles.

Ce chapitre a pour objectif de montrer que les méthodes formelles et plus spécifiquement l'utilisation d'une sémantique formelle permet à la fois de définir les problèmes et, à l'aide d'algorithmes appropriés, d'y apporter des réponses satisfaisantes.

L'organisation du chapitre. Dans la section 12.2, nous rappelons les principes des services Web et nous détaillons les constructions de BPEL. Puis dans la section 12.3, nous proposons une sémantique formelle de ce langage, nous définissons l'interopérabilité entre un client et un service et nous décrivons l'algorithme de synthèse d'un client. Enfin dans la section 12.4, nous survolons l'architecture de notre environnement et nous introduisons JCSWL, notre extension de BPEL.

12.2. Les services Web

Les services Web sont une instanciation du modèle d'architecture logicielle appelé architecture orientée service. Aussi nous décrirons tout d'abord les caractéristiques intrinsèques de ce modèle. Puis nous nous détaillerons les principes des services Web en insistant sur les aspects dynamiques. Enfin, nous présenterons les principales constructions syntaxiques de BPEL, le langage de conception de services Web (le standard actuel).

12.2.1. *Caractéristiques d'une architecture orientée service*

Une architecture orientée service consiste à structurer une application, un bloc applicatif voire un SI, en services contractualisés afin de répondre aux enjeux suivants :

– la mise en œuvre de services globaux entre blocs applicatifs, en s'inscrivant dans une politique d'interopérabilité,

– la recherche de la réutilisation à l'intérieur d'un bloc applicatif ou d'une application, en particulier au niveau des services d'infrastructure ou des services métier unitaires, en s'inscrivant dans une politique de réutilisation.

L'enjeu de la réutilisation prend ainsi tout son sens tant au niveau des services d'infrastructure, qui sont sans valeur métier mais que chaque application doit inévitablement mettre en œuvre pour ses besoins propres (sécurité, échanges, etc.), qu'au niveau des services métier de fine granularité, qu'il s'agisse de la réutilisation de composants logiciels au sein d'une application (par exemple, une couche logicielle de services communs pour les chaînes « batch ») ou de l'invocation d'un Service Web transverse de type « validation d'adresse ». La réutilisation de tels services évite la duplication de code entre applications ou entre modules de l'application. En revanche, pour les services métier gros grain, qui exposent la valeur d'un système d'information à l'extérieur de ses frontières, l'enjeu porte plus sur la capacité de ces services à interopérer avec les autres blocs du SI qui vont en être consommateurs.

Une architecture orientée service permet aux entreprises d'offrir, à la demande, des bouquets de services à leurs clients, à leurs filiales et à leurs partenaires. Pour ce faire, les services offerts doivent garantir :

– **La neutralité technologique** : ils doivent être invocables au travers de standards masquant la diversité des socles techniques.

– **Le couplage faible** : ils doivent masquer totalement la manière dont ils sont implémentés et donc, la logique métier et la logique applicative.

– **La transparence vis-à-vis de leur accessibilité** : l'interface du service et des informations sur sa localisation doivent être accessibles à un utilisateur potentiel. Il existe toutefois différents niveaux d'accessibilité en fonction du profil utilisateur.

– **L'orientation message** : les messages, de type asynchrone, sont l'unique moyen de communiquer entre services en sachant que les messages ont souvent une durée de vie plus longue que leur temps de transmission.

– **La composition** : les services peuvent être simples ou composites. Le service composite permet l'assemblage de services existants qui accèdent, combinent des informations et des fonctions pouvant provenir de différents fournisseurs de services. Les blocs et non pas les monolithes vont permettre de construire de tels services.

– **L'autonomie** : les points de terminaison peuvent être construits, déployés, administrés, mis à jour et sécurisés de façon indépendante les uns des autres.

Les applications orientées services sont développées comme des services indépendants offrant des interfaces bien définies à leurs utilisateurs potentiels. Les services Web sont aujourd'hui la solution technologique la mieux adaptée pour répondre aux objectifs cités ci-dessus.

12.2.2. *De l'architecture orientée service aux services Web*

Un service Web est un type spécifique de service identifié par une URI et présentant les caractéristiques suivantes :

– Il expose ses fonctionnalités par l'Internet utilisant des standards et des protocoles de l'Internet.

– Il est implémenté via une interface auto-descriptrice basée sur un standard de l'Internet.

Le concept des Web Service s'articule actuellement autour des trois acronymes suivants :

– SOAP (« *Simple Object Access Protocol* ») est un protocole d'échange entre applications indépendant de toute plate-forme, basé sur le langage XML.

– WSDL (« *Web Services Description Language* ») donne la description au format XML des services Web en précisant les méthodes invocables, leur signature et le point d'accès (URL, port, etc.).

– UDDI (« *Universal Description, Discovery and Integration* ») normalise une solution d'annuaire distribué de services Web, permettant à la fois la publication et l'exploration. UDDI se comporte lui-même comme un service Web dont les méthodes sont appelées via le protocole SOAP.

Le fournisseur d'un service Web publie l'interface de son service dans le format WSDL. Le client, par la suite, recherche le service Web selon des caractéristiques définies par l'annuaire UDDI. L'annuaire trouve le service et envoie la localisation du serveur qui héberge le service. Le client interroge le serveur concernant le contrat de service proposé. Le serveur lui renvoie le format d'appel en WSDL. Le client invoque le service par un message SOAP et le serveur lui répond en fournissant un résultat.

La normalisation actuelle autour des services Web est cependant un vaste travail qui va bien au-delà de la simple invocation d'une méthode d'un objet distant. Différents travaux ont ainsi démarré pour tenter de définir une véritable infrastructure distribuée, capable de satisfaire l'ensemble des besoins d'une application distribuée, aussi bien en terme de normalisation des échanges qu'en terme de services transversaux. Ces normes ont été spécifiées par un organisme rassemblant des industriels, acteurs principaux du marché : le WS-I [WSI 02]. On peut schématiser cette organisation des comités de normalisation selon le découpage matriciel suivant.

– Normalisation des services transverses sur trois axes horizontaux.

1) **Couche transport** : Définition de la structure des messages utilisés par les applications pour se découvrir et dialoguer entre elles.

2) **Couche sémantique** : Normalisation des données participant aux échanges selon des critères métier.

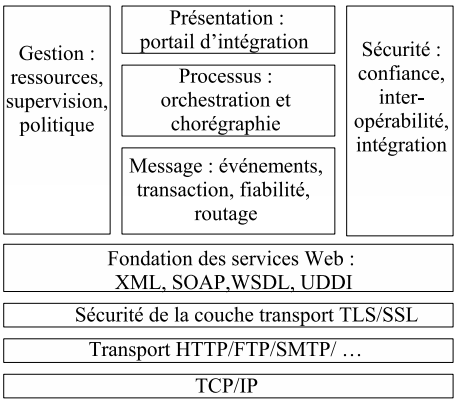


Figure 12.1. *L'architecture fonctionnelle des services Web*

3) **Couche de gestion des processus** : Standardisation de la gestion des processus métier qui s'étendent sur plusieurs applications disponibles sur l'Internet.

– Normalisation des services transversaux sur trois axes verticaux.

1) **Service d'annuaire** : Standardisation des moyens d'accès à un service à partir d'une requête portant sur le contenu d'un service ou sur un fournisseur.

2) **Service de sécurité** : Normalisation des moyens permettant de couvrir les problématiques d'authentification et de gestion des droits d'accès.

3) **Service de transaction** : Normalisation des moyens permettant de garantir l'intégrité des transactions longues impliquant plusieurs Web Services.

La figure 12.1 présente les différents niveaux de normes élaborés par le WS-I.

12.2.3. *Le langage BPEL*

BPEL repose essentiellement sur WSDL pour prendre en charge les informations relatives à la localisation du service et au format des données dans les messages. Nous nous limitons donc ici à la présentation des élements constitutifs du processus métier en les décomposant en deux catégories : les opérations élémentaires et les constructeurs. Nous ne nous préoccupons pas dans cette présentation de la syntaxe (verbeuse) de BPEL. BPEL permet de manipuler des variables et des types, le lecteur en trouvera une illustration dans la section 12.4. Précisons enfin que nous occultons certains aspects importants de BPEL comme le mécanisme de compensation mais qui n'ont pas d'impact sur l'interopérabilité.

Les opérations élémentaires.

– La primitive `receive` correspond à la réception d'un message (issu d'un client ou d'un autre service). BPEL offre la possibilité d'une communication synchrone ou asynchrone. Nous avons choisi de n'étudier que le mode asynchrone car il est facile de simuler une communication synchrone à l'aide de ce mode alors que la simulation inverse nécessite la création de processus auxiliaires. De plus, dans le cadre de services composites, le mode asynchrone est plus approprié aux transactions longues et au couplage faible.

– La primitive `invoke` correspond à l'envoi d'un message (vers un client ou un autre service). Contrairement à la réception d'un message qui suspend le service si le message demandé n'est pas présent dans le tampon, le processus continue son activité après l'envoi.

– La primitive `throw` permet à un processus d'abandonner un contexte d'exécution en signalant une faute. La faute est traitée par le contexte courant ou à défaut transmise au contexte englobant. Si la faute n'est prise en charge par aucun contexte, elle provoque la terminaison du processus.

– La primitive `terminate` permet à un processus de terminer immédiatement le service. On peut l'assimiler à une faute non traitée aussi nous ne l'étudierons que dans le cas d'un processus qui achève normalement son traitement.

Les constructeurs que nous présentons peuvent être vus comme des processus bâtis à partir de sous-processus et d'opérations élémentaires.

Les constructeurs.

– Le processus `empty` (comme son nom l'indique) ne fait rien. Il est introduit dans BPEL pour spécifier l'absence de traitement dans certaines branches d'une exécution conditionnelle.

– Le processus `sequence` enchaîne l'exécution de différents processus.

– Le processus `switch` est constitué de sous-processus dont l'exécution est conditionnée par une expression booléenne constituée de variables internes (donc inconnues du client) et exécute la première branche dont la condition est satisfaite. De manière optionnelle, la dernière branche peut ne pas être conditionnée par un test.

– Le processus `while` exécute itérativement un sous-processus tant qu'une expression booléenne (similaire à celles apparaissant dans `switch`) est satisfaite.

– Le processus `scope` définit un contexte d'exécution d'un sous-processus de la manière suivante. Ce processus peut être abandonné sur réception d'un message dont le type appartient à un certain ensemble, sur expiration d'un délai qui débute avec l'activation du contexte ou sur le signal d'une faute. Dans tous les cas, un processus est attaché à chacun de ces évènements.

– Le processus `pick` attend un message dont le type appartient à un certain ensemble pour exécuter un processus correspondant à chaque type de message. De manière optionnelle, un délai peut contrôler cette attente. Il est aisé de simuler un processus `pick` à l'aide d'un processus `scope`. Aussi nous ne l'étudierons pas explicitement.

– Le processus `flow` lance l'exécution parallèle de processus et se termine lorsque tous ces processus se terminent. Ces processus peuvent se synchroniser en cours d'exécution. Dans la mesure où cette synchronisation n'est pas perçue par le client du service, nous ne la modéliserons pas. Il s'agit évidemment d'une simplification que nous lèverons dans des travaux à venir.

12.3. La synthèse de clients

L'un des apports des services Web est le fait que les clients potentiels découvrent lors de l'invocation la spécification du service. Cependant cela soulève pour le client le problème de conduire de manière correcte l'interaction avec le service au vu du flux des messages échangés. Dans cette section, nous développons une approche formelle qui conduit à la synthèse d'un client, étant donnée la spécification d'un service. Cette méthode se décompose en trois étapes. Nous définissons d'abord une sémantique formelle d'un service Web sous forme d'une algèbre de processus temporisée. Cette sémantique conduit à un système de transition temporisé (STT) engendré par un automate temporisé. Puis nous introduisons une relation d'interopérabilité entre deux STT (l'un représentant le serveur et l'autre le client). Enfin nous décrivons un algorithme de synthèse qui, partant de l'automate du serveur, vérifie l'existence d'un automate (déterministe) correspondant au client interopérable et le construit le cas échéant.

12.3.1. *Une sémantique formelle pour le comportement d'un service Web*

BPEL fournit un ensemble d'opérateurs pour spécifier le comportement d'un service Web. Comme il a été souligné dans [STA 03], ce type de description est proche du paradigme des algèbres de processus illustré par CCS [MIL 89], CSP [HOA 85] et ACP [BER 84]. Cependant le temps est explicitement présent dans certains des constructeurs BPEL et par conséquent la sémantique associée aux algèbres de processus standard est inappropriée pour BPEL.

Aussi la sémantique que nous proposons associe un automate temporisé (AT) à un processus abstrait BPEL. Rappelons brièvement ce qu'est un automate temporisé. Il s'agit d'un automate fini (non déterministe) complété par un ensemble fini d'horloges. Une exécution d'un AT consiste en pas de temps (éventuellement nuls) entrelacés avec des transitions discrètes. Une configuration d'un AT est composée d'un état de l'automate et d'une valeur par horloge (appelée valuation d'horloge). Les transitions discrètes correspondent aux arcs entre états de l'automate. Les horloges contrôlent le comportement de l'AT de la façon suivante.

Une expression booleénne relative aux horloges, appelée un invariant, étiquette chaque état. Le temps peut s'écouler dans un état tant que l'invariant de l'état reste vérifié. En plus de son label, la spécification d'un arc comprend une expression booléenne similaire à un invariant, ici appelée une garde, et un sous-ensemble d'horloges à réinitialiser. Afin d'« emprunter » un arc depuis une configuration, la garde correspondante doit être satisfaite et la valuation des horloges après les réinitialisations doit satisfaire l'invariant de l'état destination de l'arc.

DÉFINITION 12.1 (Automate temporisé [ALU 94]) *Un automate temporisé (AT) est un tuple* (L, C, A, E, l_0) *où* L *est l'ensemble des* localités *ou états de contrôle,* C *est l'ensemble des* horloges*,* A *est l'ensemble des* actions*,* E *est l'ensemble des* arcs*. Un arc* e *est un tuple* (s, g, a, r, d) *avec* s *la localité source,* g *la garde,* a *l'action,* r *le sous-ensemble d'horloges à réinitialiser et* d *la localité de destination.* l_0 *est la localité initiale.*

L'alphabet de l'automate temporisé. Nous commençons par définir les labels des transitions de l'automate. Quatre types d'actions sont possibles :

– Les actions immédiates, correspondant à des actions logiques comme le choix d'une alternative ou la levée d'une exception. Elles ne sont pas observées par le client et sont notées τ. Nous introduisons l'ensemble des exceptions E_x qui apparaissent dans les règles sémantiques mais ne doivent pas apparaître dans l'automate si toutes les exceptions sont traitées.

– Les expirations de délais sont notées par *nameto*.

– Les réceptions et les envois de messages : l'ensemble des types de messages est noté par M. Une émission est notée $!m$ et une réception par $?m$ avec $m \in M$. Nous introduisons aussi $!M = \{!m \mid m \in M\}$ et $?M = \{?m \mid m \in M\}$. Enfin le caractère générique $*$ représente indifféremment ! ou ?.

– Afin de vérifier que le client détecte la fin du service, nous introduison $\surd$, l'évènement de terminaison. Cette action simplifie aussi la définition de certaines règles.

Les états de l'automate temporisés. Chaque état est associé à un processus BPEL obtenu par des transformations successives à partir du processus étudié. Les processus de deux états sont par construction différents. Initialement, il y a un seul état (la localité initiale) correspondant au processus étudié. Après la construction d'une nouvelle transition de l'automate, un nouveau processus est calculé et si ce processus n'étiquette pas déjà un état de l'automate, un nouvel état est créé. En raison de la définition des règles sémantiques décrites ci-après, le nombre de processus ainsi dérivés est fini (et par conséquent le nombre d'états de l'automate).

Les transitions de l'automate temporisé. Les transitions partant d'un état sont obtenues par une analyse descendante de l'expression du processus étiquetant l'état.

Cette analyse est habituellement définie à l'aide de règles de sémantique opérationnelle. La définition d'une règle semantique $[op_x]$ pour un processus générique $P = op_x(P_1, P_2, \ldots)$ est similaire à celle des algèbres des processus temporisés [NIC 94]. Les composants d'une règle sont :

– une expression booléenne relative à des transitions potentielles de certains composants de P : *name*$Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})$;

– cette condition est complétée par une seconde condition sur les étiquettes apparaissant dans les transitions, notée par *name*$garde(\{\alpha_i\})$.

– Si les deux conditions sont remplies alors une transition est possible pour P dont le label *name*$Lexp(\{\alpha_i\})$ est une expression dépendant des labels des transitions des sous-processus et

– le nouveau processus est une expression *name*$Nexp(P, \{P'_{o(i)}\})$ dépendant du processus courant et des nouveaux sous-processus.

Aussi, une règle générique, présentée sous forme standard a la structure suivante :

$$[op_x] : \frac{\textit{name}Bexp(\{P_{o(i)} \xrightarrow{\alpha_i} P'_{o(i)}\})}{P \xrightarrow{\textit{name}Lexp(\{\alpha_i\})} \textit{name}Nexp(P, \{P'_{o(i)}\})} \text{ si } \textit{name}garde(\{\alpha_i\})$$

Les gardes et les réinitialisations d'une transition seront définies plus loin. nous présentons d'abord les règles correspondant aux processus élementaires de BPEL. Ces processus élémentaires sont : `empty`, $?o[m]$, $!o[m]$ and `throw`$[e]$.

Le processus `empty`**.** `empty` ne fait que « signaler » sa terminaison (0 dénote le processus qui n'admet aucune transition).

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

Les processus $?o[m]$ **and** $!o[m]$**.** Le processus $?o[m]$ consiste à recevoir un message de type m. Le processus $!o[m]$ consiste à envoyer un message de type m.

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

Le processus `throw`**.** Le processus `throw`$[e]$ lève l'exception e qui doit être traitée dans un processus `scope`. Comme il s'agit d'une fin anormale le processus atteint est 0 et non pas `empty`.

$$\forall\, e \in E_x,\ \texttt{throw}[e] \xrightarrow{e} 0$$

Le processus `sequence` **(** ; **).** Le processus $P \ ; \ Q$ exécute le processus P puis le processus Q. Puisque l'opérateur « ; » est associatif, nous restreignons sans perte de généralité le nombre d'opérandes à deux processus. Le processus $P \ ; \ Q$ agit comme P tant que celui-ci n'indique pas sa terminaison. Dans ce cas, le processus $P \ ; \ Q$ agit comme Q.

$$\forall\, a \neq \surd,\ \frac{P \xrightarrow{a} P'}{P \ ; \ Q \xrightarrow{a} P' \ ; \ Q}$$

$$\forall\, a,\ \frac{P \xrightarrow{\surd} \wedge\ Q \xrightarrow{a} Q'}{P \ ; \ Q \xrightarrow{a} Q'}$$

Notons que, par définition de l'ensemble des règles, si une action $a \neq \surd$ est telle que $P \xrightarrow{a}$ alors $P \xrightarrow{\surd}$ n'est pas possible.

Le processus `switch`**.** Le processus $\texttt{switch}[\{P_i\}_{i \in I}]$ choisit de se comporter comme l'un des processus de l'ensemble $\{P_i\}$. Chaque branche de l'alternative est gardée par une condition *interne*. De plus, les conditions sont evaluées par ordre d'apparence dans la spécification. Cependant puisque le client n'a aucun moyen de prédire le choix du service, cet ordre n'est pas significatif. La conséquence principale est que du point vue du client, *ce choix est non déterministe*. Le processus `switch` devient l'un des sous-processus de manière inobservable. Notons que nous avons implicitement supposé qu'au moins une condition est remplie. Si ce n'est pas le cas, il suffit d'ajouter le processus `empty` parmi les sous-processus.

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

Le processus `while`**.** Le processus $\texttt{while}[P]$ exécute de manière itérativement un sous-processus aussi longtemps qu'une condition *interne* est remplie. Comme `switch`, `while` évalue de façon silencieuse sa condition. Aussi nous avons deux règles dépendant de l'évaluation.

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

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

Le processus `flow`**.** Le processus $\texttt{flow}[\{P_i\}_{i \in I}]$ active simultanément $\{P_i\}$, un ensemble de processus. Puisque les primitives de synchronisation de BPEL sont internes, nous ne les prenons pas en compte pour la sémantique. Par conséquent, cette

exécution parallèle est semblable à un « *fork-join* » et le processus finit son exécution lorsque tous les sous-processus ont achevé leur exécution. Les sous-processus du processus `flow` agissent indépendamment mais se synchronisent pour indiquer leur terminaison. Dans ce dernier cas, le processus `flow` devient le processus 0. De plus, les actions internes sont immédiates et donc la possibilité d'une telle action dans un sous-processus empêche l'occurrence d'une action non immédiate (envoi ou réception d'un message) dans un autre sous-processus.

• Actions autonomes :

1. $$\frac{\forall\, a \in E_x \bigcup \{\tau\},\quad \exists\, j \in I,\ P_j \xrightarrow{a} P'}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{a} \texttt{flow}[\{P_i \mid i \in I \backslash \{j\}\} \bigcup \{P'\}]}$$
2. $$\frac{\forall\, m \in M,\quad \exists\, j \in I,\ P_j \xrightarrow{*m} P' \text{ and } \forall\, i \neq j,\ \forall\, a \in E_x \bigcup \{\tau\},\ \nexists\, k \in I,\ (P_i \xrightarrow{a})}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{*m} \texttt{flow}[\{P_i \mid i \in I \backslash \{j\}\} \bigcup \{P'\}]}$$

• Terminaison :

$$\frac{\forall\, i \in I,\ P_i \xrightarrow{\surd}}{\texttt{flow}[\{P_i \mid i \in I\}] \xrightarrow{\surd} 0}$$

Le processus `scope`. Le processus $\texttt{scope}(P, E)$ défini par $E \stackrel{def}{=} [\{(m_i, P_i) \mid i \in I\}, (d, Q), \{(e_j, R_j) \mid j \in J\}]$ peut évoluer soit par l'évolution de P, la réception d'un message m_i, l'expiration du délai d ou l'occurrence d'une exception e_j. Nous notons $M_I = \{m_i \mid i \in I\}$ et $E_J = \{e_j \mid j \in J\}$.

• actions de P : seule la terminaison permet de « quitter » le constructeur `scope`.

$$\frac{P \xrightarrow{\surd}}{\texttt{scope}(P, E) \xrightarrow{\surd} 0} \qquad \frac{P \xrightarrow{a} P'}{\texttt{scope}(P, E) \xrightarrow{a} \texttt{scope}(P', E)} \quad \text{avec } a \neq \surd$$

• Réception d'un message m_i :

$$\forall\, i \in I,\ \frac{\forall\, a \in E_x \bigcup \{\tau, \surd\},\ \neg(P \xrightarrow{a})}{\texttt{scope}(P, E) \xrightarrow{?m_i} P_i}$$

• Traitement d'une exception : varie selon que l'exception est capturée ou pas.

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

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

Si une exception e n'est capturée dans un aucun `scope` alors le processus est erronné ce qui se détecte en vérifiant qu'aucune exception n'étiquette une transition de l'AT produit.

Les horloges de l'automate temporisé. Nous associons une horloge à chaque sous-processus `scope` du processus et une horloge particulière (x_{im}) pour gérer les actions immédiates. Etant donné un processus, nous déterminons par une analyse descendante quelles horloges sont actives, *i.e.* quels sous-processus `scope` sont en cours d'exécution. L'invariant d'un état dépend de la possibilité d'une action immédiate. Si tel est le cas, alors l'invariant est $x_{im} = 0$, sinon celui-ci est la conjonction de conditions élémentaires $x \leq d$ avec x, une horloge active et d le délai défini dans le sous-processus correspondant à x.

Les horloges à réinitialiser pour une transition sont les horloges inactives du processus source de la transition qui deviennent actives dans le processus destination. x_{im} est toujours réinitialisée.

Les transitions définies par les règles sémantiques ont pour garde $true$. De plus, nous ajoutons à chaque état qui comporte des horloges actives, un ensemble de transitions étiquetées par to (une par sous-ensemble d'horloges actives qui peuvent atteindre simultanément leur borne temporelle). Pour une telle transition, la garde spécifie que les horloges du sous-ensemble ont atteint leur borne tandis que les autres horloges actives ne l'ont pas encore atteint.

La construction de l'automate temporisé. La contruction de l'automate temporisé du service peut être résumée ainsi :

– L'algorithme maintient un ensemble de processus à examiner et un automate partiellement construit. Il démarre avec le processus du service et un automate réduit à un seul état.

– Lorsqu'il analyse un processus, il construit d'abord les transitions correspondant aux règles sémantiques et il insère chaque processus destination, non encore présent dans l'automate, dans l'ensemble des processus à examiner.

– Puis il détermine l'ensemble des horloges actives du processus. A partir de cette information et des transitions déjà construites, il déduit l'invariant de l'état. Enfin, il génère les transitions de « timeout ».

– Le calcul du sous-ensemble des horloges à réinitialiser d'une transition est effectué soit lors de la contruction de la transition si le processus destination a déjà été examiné, soit lors de l'examen (ultérieur) de ce processus.

12.3.2. *Relation d'interoperabilité entre client et service*

Il est clair que l'automate temporisé construit est une représentation compacte du comportement observable du processus abstrait BPEL. Cependant les automates temporisés comme nous l'avons déjà indiqué ont eux-mêmes une sémantique formelle en termes de systèmes de transitions temporisés (STT). Un STT est un tuple $(S, s_0, A, \rightarrow)$ avec S l'ensemble des états, $s_0 \in S$ l'état initial, A un ensemble fini d'actions et $\rightarrow \subseteq S \times (A \cup \mathbb{R}_{\geq 0}) \times S$ l'ensemble des transitions. Si $(q, e, q') \in \rightarrow$, nous le notons plus commodément $q \xrightarrow{e} q'$. Une transition $q \xrightarrow{d} q'$ with $d \in \mathbb{R}_{\geq 0}$ correspond à l'écoulement d'une durée d. Les états du STT associé à un AT sont naturellement ses configurations accessibles et ses transitions sont soit ses transitions discrètes soit l'écoulement du temps dans une localité.

Nous décrivons tout d'abord de manière informelle ce que devrait être une interaction correcte entre deux STT. Comme pour la relation de bisimulation, celle-ci repose sur une relation entre états des deux STT. Evidemment la paire des états initiaux doit appartenir à la relation.

De plus, les états d'une paire doivent avoir une vue cohérente de l'interaction à venir. Ceci implique que la relation doit prendre en compte les transitions mutuellement observables. Aussi, nous définissons les transitions observables d'un STT par $s \stackrel{a}{\Rightarrow} s'$ ssi $s \stackrel{\tau^* a \tau^*}{\rightarrow} s'$, $s \stackrel{\epsilon}{\Rightarrow} s'$ ssi $s \stackrel{\tau^*}{\rightarrow} s'$ et $s \stackrel{d}{\Rightarrow} s'$ ssi $s \stackrel{d_1 \tau \ldots \tau d_n}{\rightarrow} s'$ avec $\sum d_i = d$.

Nous pourrions alors exiger que (de manière similaire à la bisimulation) lorsqu'un état s d'une paire (s, s') peut évoluer par une transition observable vers un état s_1, s' devrait posséder une transition de même étiquette conduisant à un état s'_1 composant avec s_1 une autre paire de la relation.

Cependant nous devons être prudents. Premièrement, si un STT envoie un message alors l'autre STT doit être capable de le recevoir. Aussi il est nécessaire d'introduire la notion d'actions complémentaires $\overline{?m} = !m$, $\overline{!m} = ?m$ et $\forall a \notin \{!m\}_{m \in M} \cup \{?m\}_{m \in M}$ $\overline{a} = a$ et d'exiger que l'évolution « synchronisée » s'effectue par le biais d'actions complémentaires.

Mais une telle définition est trop forte car elle ne distingue pas entre la nature différente d'une émission et d'une réception de message : une émission est une action alors qu'une réception est une *réaction* et ne peut spontanément se produire. Par conséquent, de manière plus appropriée la relation d'interaction requiert que si dans un état s d'une paire (s, s'), un STT peut recevoir un message m, alors d'une part il existe un état s'' de l'autre STT non distinstinguable de s' au vu des transitions observables qui peut envoyer m et d'autre part dans s' l'autre STT peut envoyer un message (éventuellement différent de m). La première condition reflète le fait que le premier STT n'est pas sur-spécifié tandis que la seconde implique qu'il n'attendra pas indéfiniment un message.

Ces considérations conduisent à la définition suivante.

DÉFINITION 12.2 (Relation d'interaction) *Soient deux STT* $\mathcal{N}_1 = (S, s_{01}, A, \rightarrow_1)$ *et* $\mathcal{N}_2 = (S, s_{02}, A, \rightarrow_2)$. *Alors* $\mathcal{N}_1$ *and* $\mathcal{N}_2$ interagissent correctement *ssi* $\exists \sim \subseteq S_1 \times S_2$ *telle que* $s_{01} \sim s_{02}$ *et* $\forall s_1, s_2$ *tels que* $s_1 \sim s_2$ *:*

– *Soit* $a \notin \{?m \mid m \in M\}$

si $\exists\, s_1 \stackrel{a}{\Rightarrow} s'_1$, *alors* $\exists\, s_2 \stackrel{\overline{a}}{\Rightarrow} s'_2$ *avec* $s'_1 \sim s'_2$

et si $\exists\, s_2 \stackrel{a}{\Rightarrow} s'_2$, *alors* $\exists\, s_1 \stackrel{\overline{a}}{\Rightarrow} s'_1$ *avec* $s'_1 \sim s'_2$

– *Soit* $m \in M$

si $s_1 \stackrel{?m}{\Rightarrow} s'_1$ *alors*

$\exists\, s_2^- \stackrel{w}{\Rightarrow} s_2$, $\exists\, s_2^- \stackrel{w}{\Rightarrow} s_2^+$, $\exists\, s_2^+ \stackrel{!m}{\Rightarrow} s'_2$ *avec* $s_1 \sim s_2^+$, $s'_1 \sim s'_2$ *et* w *un mot*

et $\exists\, s_2 \stackrel{!m'}{\Rightarrow} s'_2$

si $s_2 \stackrel{?m}{\Rightarrow} s'_2$ *alors*

$\exists\, s_1^- \stackrel{w}{\Rightarrow} s_1$, $\exists\, s_1^- \stackrel{w}{\Rightarrow} s_1^+$, $\exists\, s_1^+ \stackrel{!m}{\Rightarrow} s'_1$ *avec* $s_2 \sim s_1^+$, $s'_2 \sim s'_1$ *et* w *un mot*

et $\exists\, s_1 \stackrel{!m'}{\Rightarrow} s'_1$

12.3.3. *L'algorithme de synthèse*

Nous sommes maintenant en mesure de présenter l'algorithme de synthèse du client. Tout d'abord, le client doit être implémentable, ce qui signifie que son comportement doit être déterministe. D'autre part, puisqu'il doit prendre compte les *horloges* du service, son comportement doit s'exprimer sous forme d'un STT. Ceci nous conduit pour le client au choix d'un automate temporisé déterministe qui doit être en relation d'interaction avec l'automate du service.

Avant de détailler cet algorithme, remarquons que certains processus BPEL n'admettent pas de client capable d'interagir correctement avec eux. Par exemple, le processus `switch`$[?o[m], ?o[m']]$ choisit de manière interne de recevoir soit un message m soit un message m'. Un AT déterministe dans son état initial doit donc envoyer soit m, soit m'. Or une fois son choix effectué le service n'attend plus que l'un des deux messages. Autrement dit, ces deux états ne peuvent être en relation d'interaction avec l'état initial du client. Par contre, le processus `switch`$[!o[m], !o[m']]$ admet comme client, un processus qui attend soit le message m soit le message m'. Nous dirons qu'un processus est *ambigu* s'il n'admet pas d'AT déteministe qui soit en relation d'interaction avec lui.

Notre algorithme ne recherche pas n'importe quel AT déterministe mais restreint sa recherche à un AT qui a les mêmes horloges que l'automate du service. Par conséquent lorsque l'algorithme annonce ambiguïté, il se peut qu'un AT client existe. Cependant cette restriction nous semble raisonnable car les cas d'erreur de l'algorithme correspondent à des processus BPEL irréalistes.

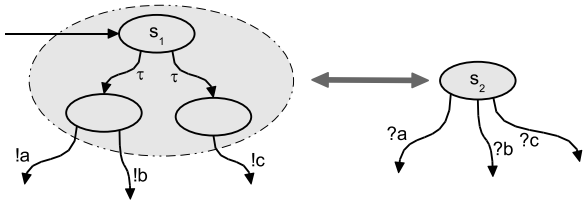


Figure 12.2. *Sous-ensemble d'états du service (à gauche) - état du client (à droite)*

Le principe général de notre algorithme est semblable à une procédure de déterminisation : un état de l'AT client correspond à un sous-ensemble d'états de l'AT du service (voir la figure 12.2).

De manière plus précise, chaque état potentiel de l'AT client est associé avec un sous-ensemble d'états de l'AT du service qui doivent être en relation d'interaction avec cet état. La construction maintient une pile constituée de telles paires à analyser. Initialement, la pile contient l'état initial du client et le singleton contenant l'état initial de l'AT du service. L'algorithme se termine soit lorsque la pile est vide (*i.e.* le client est construit) soit lorsqu'il détecte l'ambiguïté du service.

Le traitement d'une paire consiste d'abord à enrichir le sous-ensemble avec tous les états accessibles par des τ transitions. Une fois complété, si ce sous-ensemble est déjà associé à un état client alors on supprime l'état client de la paire et on redirige les transitions entrantes vers l'état existant. Dans le cas contraire, on extrait du sous-ensemble d'états ceux qui n'ont pas de τ transitions pour vérifier que leurs horloges actives sont identiques. Si tel n'est pas le cas, alors il y a ambiguïté temporelle et on arrête la construction. Dans le cas contraire, on vérifie la relation d'interaction pour les transitions discrètes et on construit pour chaque étiquette une transition complémentaire dans l'automate client avec comme sous-ensemble associé les états atteints par une telle transition.

12.3.4. *Travaux similaires*

Dans un travail précédent auquel a participé l'un des auteurs [MEL 03, HAD 04], une démarche semblable a été suivie. Cependant, la sémantique temporelle adoptée a été celle du temps discret. Ce choix a l'avantage de conduire à un algorithme complet, *i.e.* qui produit un client s'il n'y a pas ambiguïté. Cependant, il comporte deux inconvénients majeurs : le client produit est difficilement interprétable par un être humain et il y a risque d'explosion combinatoire si les échelles de temps des horloges n'ont pas le même ordre de grandeur.

La plateforme WSAT [FU 04a, FU 04b] permet aux concepteurs d'un service Web conçu de manière modulaire de vérifier des propriétés exprimées par des formules LTL avec l'outil SPIN. La sémantique formelle est définie par composition de patrons pour chaque construction BPEL. Un patron est connecté par des transitions à partir de son état final à l'état initial du patron suivant dans la description BPEL. Une autre approche pour définir une sémantique formelle aux services Web est basée sur les « *message sequence charts* » [FOS 03]. La démarche de vérification consiste à tester des équivalences *via* l'égalité des langages. Dans [TUR 05], la notation CRESS (*Chisel Representation Employing Systematic Specification*) est appliquée afin de formaliser les services Web. Ce modèle présente deux avantages importants : une traduction automatique en LOTOS suivie d'une d'analyse par des outils tels que TOPO, LOLA and CADP et la possibilité d'un génération de « code » pour un déploiement du service. Aucun de ces travaux ne couvre les aspects temporels.

12.4. Un environnement pour la conception de services Web

12.4.1. *Architecture de la plate-forme*

La plate-forme permet de concevoir un service Web complexe à partir de services Web existants. Elle offre plusieurs modules permettant de définir, compiler et déployer un service complexe. Elle est composé de quatre modules : un module de recherche, une interface d'agrégation, un compilateur et un module de déploiement. La figure 12.3 présente les différents modules de la plate-forme ainsi que leurs différentes interactions.

Le module de recherche : Ce module est un client UDDI permettant d'explorer le registre UDDI afin de localiser et de rechercher des services Web. Il communique avec le registre UDDI afin de localiser et de télécharger des services Web.

L'interface homme machine : C'est un environnement de développement de services Web complexes. L'environnement est composé d'un langage de composition (JCWSL), une extension du langage JAVA. Il inclut un browser de service permettant de rechercher un service Web en spécifiant les critères de recherche, de lister les services répondant à ces critères suite à une requête au registre UDDI, et d'importer le ficher de description d'un service.

Le compilateur : le compilateur prend en entrée la description du service complexe écrit en JCWSL et génère le code correspondant ainsi que le comportement en BPEL. Le compilateur est composé d'un analyseur et d'un générateur.

– *L'analyseur :* applique deux sortes d'analyse, syntaxique et sémantique. Pendant l'analyse syntaxique, l'analyseur vérifie que le fichier est conforme à la grammaire du langage, vérifie la validité de chaque service importé, construit pour chaque service

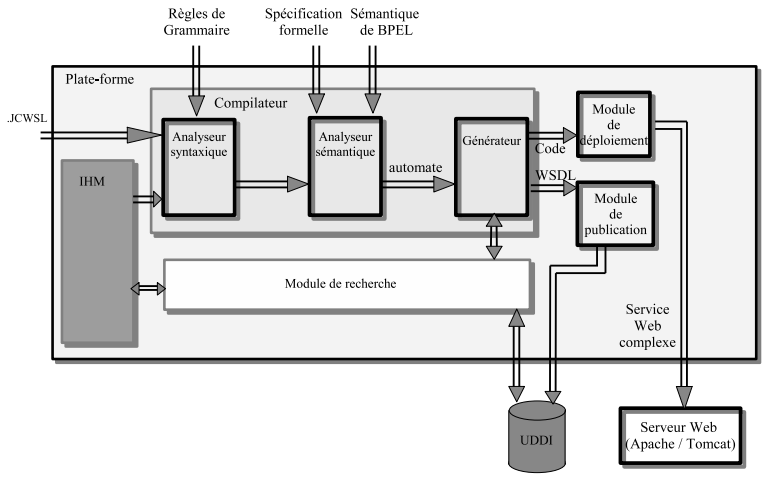


Figure 12.3. *Architecture de la plate-forme*

importé une librairie (paquet) et génère l'arbre syntaxique du service complexe. L'analyseur sémantique doit s'assurer que le service complexe vérifie certaines propriétés. En effet, moyennant l'arbre syntaxique et l'algorithme d'abstraction, il génère la description observable du service et analyse son automate afin de vérifier la non ambiguïté du service.

– *Le générateur :* une fois les vérifications syntaxiques et sémantiques effectuées, le générateur produit automatiquement le code du service et son processus abstrait.

Le module de déploiement : génère l'application Web qui héberge le service Web. Il définit aussi l'implémentation du modèle de communication du service. En effet, les informations concernant la communication du service sont utilisées afin de définir l'interface d'accès (« *binding part* ») de la description du service.

Le module de publication : permet de publier, dans le registre UDDI, le fichier WSDL du service.

La plate-forme fournit en plus des utilitaires nécessaires pour la conception et le développement de services complexes, un module, nommé client générique, qui génère un client pour un service complexe (voir la section précédente).

Le client générique : permet de générer automatiquement soit un client pour interagir correctement avec le service si le service n'est pas ambigu, soit une erreur dans le cas contraire. Le client générique est composé de deux sous-modules : un module de synthèse et un module d'exécution. Le module de synthèse récupère, à partir de

l'annuaire UDDI, les spécifications du service en BPEL, les analyse et produit ensuite l'automate temporisé correspondant. Le module d'exécution est un intergiciel qui, lors de l'invocation du service, charge l'automate temporisé correspondant afin de gérer l'interaction client/service.

12.4.2. *Cycle de vie d'une application*

Cette section décrit, en détail, les différentes étapes de mise en place d'un service Web complexe en utilisant notre plate-forme. Celle-ci comprend la phase de conception du service, de génération automatique du service comprenant aussi la publication et le déploiement, et finalement la création du client correspondant.

Conception d'un service Web complexe. Le développeur dispose d'un environnement de développement pour décrire son service complexe. Un utilitaire particulier, en relation avec le registre UDDI, peut être employé pour la recherche de services Web existants. La description du service est faite par le langage JCWSL qui permet de décrire à la fois le comportement observable du service avec les constructeurs BPEL et l'implémentation du service avec le langage JAVA.

Génération, publication et déploiement du service complexe. La seconde étape traite le déploiement. Elle s'occupe donc de la génération du service complexe, de la génération et de la publication de son comportement et du déploiement du service.

A partir du fichier .jcwsl, le compilateur génère le comportement du service complexe en WSDL et l'application Web en JAVA. La publication du service complexe est effectuée exactement de la même manière que pour un service élémentaire. Ensuite, le déploiement d'un service complexe consiste en la création d'une servlet JAXM qui joue le rôle d'intermédiaire entre le client et le service. Quand la servlet reçoit un message correspondant à l'activation d'un objet (le premier message dans le protocole du service), elle crée une instance de la classe correspondante du service complexe, initialise les attributs de corrélation (suivant la valeur du message), l'ajoute dans la file et exécute son comportement (*i.e.* démarre le thread correspondant). L'URI de la servlet correspond à l'adresse du service complexe. Chaque message reçu est redirigé vers l'instance de la classe correspondante suivant l'objet et les valeurs des attributs de l'instance de corrélation.

Invocation du service. La dernière étape est la génération du client à partir de la description du service afin de permettre une interaction correcte avec le service. Elle consiste en la génération et le déploiement d'une servlet JAXM qui joue le rôle d'interface d'entrée/sortie de l'instance du client d'un service donné. La servlet joue le rôle de répartiteur qui, à la réception d'un message SOAP, le redirige vers l'instance du client correspondant. L'initialisation d'une interaction crée un nouvel automate et

l'attache à la servlet. L'adresse URI de la servlet est partagée par tous les clients courants.

12.5. Un langage de description de services Web : JCWSL

Java Complex Web Service Language (JCWSL) est un langage de programmation étendant JAVA avec des constructeurs BPEL. Le but principal est de concevoir un service Web complexe en utilisant le langage JAVA tout en définissant son comportement avec les constructeurs BPEL. Le choix de JAVA est essentiellement dû à sa popularité dans la communauté des services Web. JCWSL offre une implémentation des constructeurs BPEL : *while, empty, switch*, etc., conforme avec la sémantique observable. Les constructeurs BPEL sont définis et implémentés en respectant la sémantique définie par le langage de description. JCWSL possède les caractéristiques suivantes :

– **haut niveau d'expressivité :** étant basé sur le langage JAVA, JCWSL offre ainsi un haut niveau d'expressivité pour définir et concevoir des services Web. JCWSL est un langage de programmation.

– **transparence :** l'intégration des services Web locaux ou distants est intégrée dans le langage de manière transparente.

– **flexibilité :** JCWSL offre deux types d'invocation d'opérations : invocation visible et invisible. Une invocation visible apparaît dans le comportement du service, tandis qu'une invocation invisible est exécutée d'une manière interne sans aucune trace dans le comportement.

– **facilité de programmation :** JCWSL étend d'une manière élégante et naturelle le langage JAVA. Ainsi, la manipulation des messages et des opérations est très simple et est accessible dans les parties implémentation et comportement.

12.5.1. *Exemple de développement d'un service complexe*

Nous présentons dans cette section, le développement en JCWSL d'un service Web complexe implémentant un jeu de quiz basé sur un service Web élémentaire existant. Le service élémentaire est situé à l'adresse www.hlrs.de/quiz/quiz.wsdl, et implémente un jeu simple de quiz consistant en l'invocation de questions de différents niveaux de difficultés, ainsi qu'une vérification de l'exactitude de la réponse. Notre exemple présente un nouveau jeu où le but est de répondre à un nombre déterminé de questions dans un délai fixe. Le niveau de difficulté de chaque question peut être choisi au hasard ou spécifié par le joueur. Les différentes étapes de l'application sont les suivantes :

– le joueur spécifie le niveau de difficulté ou laisse le service choisir aléatoirement un niveau de jeu ;

– il doit répondre à 10 questions en temps limité ;

– de plus l'envoi de la réponse doit aussi se faire dans un délai limité ; soit il a répondu pendant le délai et la réponse est prise en considération par le service, soit le délai a expiré avant la réception de la réponse. Dans les deux cas, le service envoie la question suivante ;

– à la fin du jeu, le score final, calculé en fonction du nombre de réponses correctes et du niveau de difficulté de chacune, est envoyé au joueur.

La déclaration en JCWSL de cet exemple est la suivante :

```
1  //--------------------------------------------------
2  // Importation block
3  //
4  import java.io.*;
5  importBWS squiz = "http://www.hlrs.de/quiz/quiz.wsdl";
6  //--------------------------------------------------
7
8  //--------------------------------------------------
9  // Declaration block
10 //
11 public defMessage answer{String choice;}
12 public defMessage requestChoiceMode{String mess;}
13 public defMessage Score{int value;}
14 public defOperation checkAnswer(Input: answer m){}
15 public defOperation finalScore(Output: Score m){return m;}
16 public defOperation chosenMode(Output: requestChoiceMode m)
17                                      {return m;}
18 int nbQuestion = 0;
19 squiz.randomQuestion0In M1;
20 squiz.randomQuestionByDifficulty1In M2;
21 squiz.randomQuestion0Out currentQuestion;
22 squiz.checkCorrectAnswerById2In currentAnswer;
23 squiz.checkCorrectAnswerById2Out questionAnswer;
24 answer clientAnswer;
25 score result = new score();
26 requestModeChoice mode = new requestModeChoice ();
27 mode.mess = "Choose a Mode (Random or specified level): ";
28 //--------------------------------------------------
29
30 //--------------------------------------------------
31 // Main block
32 //
33 void main{
34   sequence{
```

```
35    while {JAVACODE{nbQuestion<10}}{
36      sequence{
37        invoke(choixMode,mode);
38        pick
39        {
40          eventHandler
41          {invoke(S1.randomQuestion, M1, currentQuestion);}
42          {
43            pick
44            {
45              eventHandler{invoke(checkAnswer, clientAnswer);}
46              {
47                JAVACODE{
48                  currentAnswer =
49                  new S1.checkCorrectAnswerById2In();
50                  currentAnswer.setId(currentQuestion.getId());
51                  currentAnswer.setGuessedAnswer
52                  (clientAnswer.choice);
53                }
54              execute(S1.checkCorrectAnswerById,
55              currentAnswer, questionAnswer);
56              } // end of eventHandler (2)
57              delayFor(2){}
58            } // end of pick (2)
59          }
60          eventHandler{invoke
61          (S1.randomQuestionByDifficulty,M2,currentQuestion);}
62          {
63            pick
64            {
65              eventHandler{invoke(checkAnswer,clientAnswer);}
66              {
67                JAVACODE{
68                  currentAnswer =
69                  new S1.checkCorrectAnswerById2In();
70                  currentAnswer.setId(currentQuestion.getId());
71                  currentAnswer.setGuessedAnswer
72                  (clientAnswer.choice);
73                }
74                execute(S1.checkCorrectAnswerById,
75                currentAnswer,questionAnswer);
76              } // end of eventHandler (2)
77              delayFor(2){}
```

```
78            } // end of pick (2)
79          } // end of eventHandler (1)
80          delayFor(5) { JAVACODE{ nbQuestion++;}}
81        } // end of pick (1)
82        JAVACODE{
83          if ((questionAnswer != null) &&
84          (questionAnswer.getResult() == true)){
85            result.value =
86            result.value+currentQuestion.getDifficultyLevel();
87            nbQuestion++;
88            M1 = null;
89            M2 = null;
90            currentQuestion = null;
91            currentAnswer = null;
92            questionAnswer = null;
93            clientAnswer = null;
94          }
95        }
96      } // end of sequence (2)
97    } // end of while
98    invoke(finalScore,result);
99  } // end of sequence (1)
100 } // end of main
101 //-------------------------------------------------
```

Le service Web complexe utilise le service Web élémentaire cité précédemment afin d'implémenter ce jeu. Pour ce faire, le service complexe inclut, dans son **bloc d'importation**, l'importation du service élémentaire afin de pouvoir invoquer ses services. Il a aussi besoin d'inclure la libraire d'entrée/sortie de JAVA pour l'affichage des questions (lignes 4 et 5) etc.

Le bloc de définition comprend un bloc de déclaration et un bloc principal (*main bloc*). Les opérations et les messages utilisés ultérieurement doivent être définis dans le bloc de déclaration (lignes 11 à 17) :

– *answer :* représente la réponse à une question donnée et se compose de la question et de la réponse,

– *requestChoiceMode :* définit le niveau de difficulté d'une question,

– *score :* représente le score final,

– *checkAnswer :* est une opération de type « input » qui reçoit la réponse à une question donnée,

– *finalScore :* est une opération de type « output » qui renvoie le score final,

– *choosedMode :* est une opération de type « output » qui renvoie le mode choisi.

Plusieurs messages et opérations définis précédemment sont instanciés. Ils sont soit importés du service élémentaires « squiz », soit définis par le service complexe (lignes 18 à 27). Ainsi :

– *randomQuestion :* retourne une question d'un niveau aléatoire,

– *randomQuestionByDifficulty :* retourne une question du niveau spécifié,

– *checkCorrectAnswer :* vérifie l'exactitude de la réponse du joueur à une question donnée.

L'implémentation du service complexe commence avec la fonction main, qui est composée de blocs de comportement (*behaviour bloc*) identifiés par l'un des constructeurs BPEL et de blocs d'implémentation identifiés par le mot-clef *JAVACODE*. Toutes les variables déclarées dans la section JAVACODE sont accessibles dans les constructeurs BPEL.

Nous définissons le comportement du service complexe à l'aide des constructeurs BPEL. Il existe deux manières d'appeler une opération WSDL : *execute* ou *invoke*. Les deux fonctions permettent d'assigner des messages de type sollicitation, notification ou requête/réponse. La différence fondamentale entre ces deux fonctions réside dans la visibilité extérieure. Ainsi, cet exemple fait appel à la fonction *invoke* (ligne 37) qui prend un message de type requête/réponse (« ModeChoice » et « mode » respectivement) qui apparaîtra par conséquent dans le comportement observable du service complexe.

Le constructeur *pick* est utilisé pour attendre, pendant un certain temps, la réponse du joueur (ligne 38 à 81). Le service commute ensuite suivant le mode choisi afin de renvoyer une question du niveau demandé. Une fois la question envoyée, le service utilise à nouveau le constructeur *pick* afin d'armer un délai de garde pour la réception de la réponse après quoi il passe à la question suivante. A la fin des 10 itérations, le service renvoie le score final calculé en fonction des réponses reçues (ligne 98).

12.5.2. *Description de JCWSL*

La description d'un service Web complexe avec le langage JCWSL se compose de deux parties : un bloc d'importation **importBloc** et un bloc de définition **defBloc**. Le bloc de définition correspond au corps du service complexe.

```
CWSLanguage ::≡ ( <importBloc> <defBloc> )
```

Bloc d'importation. Ce bloc se compose de deux types d'importation, les paquets JAVA et les services Web. Comme JCWSL est une extension du langage JAVA, les paquets JAVA, utilisés par le service complexe sont importés de la même manière que dans JAVA. L'importation de services Web permet d'inclure, dans le service complexe,

les services Web nécessaires. Un service importé est identifié par un nom unique (son URL) qui agit comme un espace de noms du service. Ainsi, dans le bloc définition, une structure de données du service peut être accédée exactement de la même manière qu'un paquet JAVA. En effet, un service importé est considéré comme un paquet JAVA composé d'un ensemble de classes représentant les messages, les opérations et les types définis dans le fichier WSDL. Chaque librairie d'un service est composée d'un ensemble de classes : une classe pour chaque type complexe défini dans la section *type* du fichier WSDL, une classe pour chaque message, et un ensemble de classes *stub* représentant un client pour chaque type de *binding*. Les classes *stub* ont le même nom que l'élément *binding* dans le fichier WSDL et sont composées de la liste des opérations du service accessibles par le lien. Le nombre d'instructions d'importation peut être quelconque.

```
importBloc ::≡ ("importBWS" <ID> = <STRING>";"

               | "import" <ID> (.<ID>)* ";")*
```

Bloc de définition. Ce bloc contient le comportement du service complexe. Il est composé de deux blocs : un bloc de déclaration **declaration** et un bloc principal **main**.

```
defBloc ::≡

"public" "CWSDefinition" <ID> (<declaration> <main>)
```

Le bloc de déclaration **declaration** contient la déclaration des opérations locales et/ou des messages. Ce bloc est optionel.

```
declaration ::≡ (<opdeclaration> | <messDeclaration>)*
```

Une opération est identifiée par un nom et comprend un ou deux messages suivant son type. Il existe trois types d'opération : sollicitation, notification et requête/réponse. Les opérations définies dans ce bloc sont ajoutées à celles définies par les services importés. Une opération peut inclure un corps composé de code JAVA.

```
opDeclaration ::≡  "public defOperation" <operationName>

"(" "input" ":" <messageType> <ID>

| "output" ":" <messageType> <ID>

| "input" ":" <messageType> <ID> ","

  "output" ":" <messageType> <ID>

 ")" "{"<operationBody>"}"
```

Ce corps est composé d'instructions JAVA **javaInstructions** et de constructeurs BPEL *execute*. **javaInstructions** est précédé par le mot-clef JAVACODE.

```
operationBody ::≡ (<java> | <execute>)*

<java> ::≡ "JAVACODE" {(<javaInstructions>)+}
```

Un message est identifié par son nom et est composé d'une ou de plusieurs variables. La déclaration de messages ne contient pas de méthodes. Automatiquement, lors du développement du service complexe, un ensemble de méthodes de gestion est créé. Ces méthodes ont la forme suivante : get<variableName> et set<variableName>.

```
messDeclaration ::≡ "public defMessage" <messageName>

"{"(<variableType> <variableName>";")+"}"
```

Le bloc principal contient le comportement du service complexe. Il se compose de constructeurs BPEL et de code JAVA. Ces deux types de code sont cloisonnés. Ainsi dans le corps d'un constructeur, seules l'imbrication des constructeurs ou l'invocation d'opérations est possible. Cependant, les variables JAVA sont visibles en dehors du bloc JAVACODE.

```
main ::≡ "void" "main" "{"(<process>|<java>)* "}"
```

L'identifiant **process** correspond aux constructeurs BPEL :

```
process ::≡ <pick> | <switch> |

<opCall> | <sequence> | <while> | <wait>
```

A titre d'exemple, nous détaillons le constructeur conditionnel **switch**. L'utilisation de JAVA permet de spécifier des expressions booléennes plus sophistiquées que celles de BPEL.

```
switch ::≡ Switch

( "{" "JAVACODE" "{"<condExpr> "}" "}" "{ <process> "}")+
```

L'identifiant **opCall** correspond à un appel d'opération WSDL de manière visible ou invisible (pour le client). Elle est ainsi composée des constructeurs *invoke* et *execute*.

```
opCall ::≡ (<invoke> | <execute>)

invoke ::≡ "invoke" "(" <operationName> [,<inputVariable> |
```

```
(<inputVariable>,<outputVariable>)]")"

execute ::≡ "execute" "(" <operationName> [,<inputVariable>|

(<inputVariable>,<outputVariable>)]")"
```

12.6. Conclusion

L'approche développée dans ce chapitre souligne un nouvel intérêt de la sémantique formelle. Traditionnellement, doter un langage d'une sémantique formelle a deux objectifs principaux : permettre au programmeur de comprendre les constructeurs du langage afin d'écrire des applications correctes et garantir que l'exécution d'un programme est indépendante du compilateur et de la machine choisie. Dans notre cas, la recherche d'une sémantique pour les services Web complexes nous a conduits à soulever un problème ignoré des praticiens : l'ambiguïté d'un service.

De plus, notre démarche apporte une réponse à la dynamicité des services puisqu'un client découvrant un nouveau service (ou un service existant mais modifié) génère un automate temporisé dont l'exécution permet d'assurer une interaction correcte avec le service. Enfin, cette sémantique diminue les développements logiciels car la génération de l'interface et le déploiement du service partagent de nombreux composants.

La prochaine étape de ce travail consiste à exploiter cette sémantique afin de résoudre le problème de la composition de services, à savoir comment garantir que le service composé ne se bloque pas, ne se termine jamais, etc. De manière complémentaire, nous portons nos efforts sur l'intégration des concepts de l'orientation aspect dans les services Web. Le *tissage* est évidemment une activité liée à l'exécution et se prête donc naturellement à notre approche.

12.7. Bibliographie

[ALU 94] ALUR R., DILL D. L., « A theory of timed automata », *Theoretical Computer Science*, vol. 126, n°2, p. 183–235, 1994.

[AND 03] ANDREWS T., CURBERA F., DHOLAKIA H., GOLAND Y., KLEIN J., LEYMANN F., LIU K., ROLLER D., SMITH D., THATTE S., TRICKOVIC I., WEERAWARANA S., « Business Process Execution Language for Web Services », may 2003.

[BER 84] BERGSTRA J., KLOP J., « Process algebra for synchronous communication », *Information and Control*, vol. 60, n°1-3, p. 109–137, 1984.

[FOS 03] FOSTER H., UCHITEL S., J.MAGEE, , J.KRAMER, « Model-based verification of web service compositions », *Proc. of. the 18th Int. Conf. on Automated Software Eng.*, 2003.

[FU 04a] FU X., BULTAN T., SU J., « Analysis of Interacting BPEL Web Services », *Proc. of the 13th International World Wide Web Conference (WWW'04)*, USA, ACM Press, 2004.

[FU 04b] FU X., BULTAN T., SU J., « WSAT : A Tool for Formal Analysis of Web Services », *Proc. of the 16th International Conference on Computer Aided Verification (CAV'04)*, 2004.

[HAD 04] HADDAD S., MELLITI T., MOREAUX P., RAMPACEK S., « Modelling WEB services interoperability », *Proc. of the 6th Int. Conf. on Enterprise Information Systems (ICEIS04)*, Porto, Portugal, avril 14–17 2004.

[HOA 85] HOARE C., *Communicating Sequential Processes*, Prentice Hall, Englewood Cliffs, NJ, USA, 1985.

[JUR 05a] JURIC M., « BPEL and Java », *On line journal theserverside.com*, 2005, `http://www.theserverside.com/articles/article.tss?l=BPELJava`.

[JUR 05b] JURIC M., SARANG P., MATHEW B., *Business Process Execution Language for Web Services*, Packt Publishing, 2005.

[MEL 03] MELLITI T., HADDAD S., « Synthesis of Agents for Web Services Interaction », *Workshop Semantic Web Services for Enterprise Application Integration and E-Commerce of the Fifth International Conference on Electronic Commerce*, Pittsburgh, USA, Sept 2003.

[MIL 89] MILNER R., *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1989.

[NIC 94] NICOLLIN X., SIFAKIS J., « The algebra of timed processes, ATP : theory and application », *Inf. Comput.*, vol. 114, n°1, p. 131–178, Academic Press, Inc., 1994.

[STA 03] STAAB S., VAN DER AALST W., BENJAMINS V., SHETH A., MILLER J., BUSSLER C., MAEDCHE A., FENSEL D., GANNON D., « Web Services : Been There, Done That ? », *IEEE Intelligent Systems*, vol. 18, p. 72–85, 2003.

[TID 00] TIDWELL D., « Web services - The Web's next revolution », *IBM developerWorks*, nov 2000.

[TUR 05] TURNER K. J., « Formalising Web Services », *Proc. of Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, vol. LNCS 3731, Taipei, Taiwan, Springer, p. 473–488, october 2005.

[WSD 01] WSDL, Web Services Description Language (WSDL) 1.1, Rapport, World Wide Web Consortium, march 2001, http ://www.w3.org/TR/wsdl.

[WSI 02] WS-I, « Web Services Interoperability Organization », World Wide Web page, 2002, http ://www.ws-i.org/.

Chapitre 13

Systèmes répartis adaptatifs à contraintes de temps

Nous nous intéressons dans ce chapitre à la conception et au prototypage rapide de systèmes répartis adaptatifs à contraintes de temps, c'est-à-dire adaptant dynamiquement leur architecture en fonction du contexte et du temps. Pour représenter le comportement des modules composant le système, nous utilisons le formalisme des automates temporisés, ce qui permet d'évaluer *a priori* les propriétés du système, grâce à des techniques de *model checking* et de simulation.

Nous montrons ensuite comment produire rapidement, à partir du modèle, un prototype opérationnel qui satisfait les mêmes propriétés. Nous modélisons entièrement un système particulier dans le cadre d'une étude de cas, en montrant comment l'utilisation des méthodes formelles permettait à la fois de garantir le respect de certaines propriétés, d'évaluer *a priori* la pertinence de stratégies adaptatives, et de guider la génération automatique du code.

13.1. Introduction

Les systèmes temps-réels réactifs sont définis par leur capacité à réagir constamment aux sollicitations de leur environnement en se conformant à un certain nombre de contraintes de temps. En un temps limité, le système doit acquérir et traiter les données et les évènements caractérisant l'évolution de cet environnement au cours du temps, prendre les décisions appropriées et les transformer en actions. La fonctionnalité du système provient ainsi de sa capacité à présenter les bonnes sorties (correction

Chapitre rédigé par Guillaume HUTZLER et Hanna KLAUDEL.

logique) au bon moment (correction temporelle). Du fait du caractère souvent critique de ce type d'applications, les architectures logicielles et matérielles correspondantes sont spécifiées, développées, validées avec le plus grand soin et sont ensuite figées de manière à s'assurer que le système aura un comportement déterministe et prédictible.

Du point de vue conceptuel, le problème auquel nous nous intéressons est de faire fonctionner un système réparti composé d'agents, dont on souhaite qu'il respecte des contraintes de temps. Dans le cadre de notre étude, ces contraintes ne sont pas critiques (le fait de répondre avec un délai un peu plus long que prévu n'est pas rédhibitoire) ni même strictes (lorsque le délai de réponse normal est dépassé, la valeur du résultat ne devient pas immédiatement nulle mais diminue plus ou moins rapidement avec le temps).

Du point de vue de la conception du système, il s'agira donc d'optimiser le compromis entre respect de la correction logique et respect de la correction temporelle, en relâchant au choix l'une ou l'autre des contraintes lorsque les ressources disponibles ne permettent pas de respecter les deux simultanément. Pour ce faire, les traitements sont organisés en entités élémentaires pouvant fonctionner à différents niveaux de précision, et pouvant être assemblées de manière dynamique pour construire une chaîne de traitement. Cela permet d'ajuster au besoin la qualité globale du traitement et de fonctionner en mode dégradé. La valeur de l'analyse diminue à mesure que le traitement est dégradé mais ne devient pas immédiatement nulle.

La nature des traitements à effectuer et leur priorité peuvent par ailleurs être variables et imprédictibles, de même que la disponibilité des entités (processeurs) chargées du traitement. En conséquence, il n'est pas réaliste de considérer que la durée d'un traitement est fixe, le traitement pouvant se terminer plus tôt ou plus tard que prévu, sans compter les pannes possibles de machines ou encore les pertes de messages provoquant un ralentissement global du système.

Dans ce cadre, le problème auquel nous nous intéressons plus particulièrement concerne la reconfiguration adaptative de la chaîne de traitement des données (i.e. le système à base d'agents chargé du traitement) au cours de l'exécution [HUT 04]. Cette reconfiguration peut se produire en fonction des ressources disponibles (capteurs, processeurs, effecteurs), de la qualité logique souhaitée, de la qualité temporelle assurée et des évènements se produisant dans l'environnement. Elle ne pourra s'effectuer de manière centralisée et devra donc être prise en charge par les agents eux-mêmes, en plus de leur activité de traitement des données.

Nous nous plaçons dans une démarche de prototypage rapide, qui vise à proposer une approche complète de conception de systèmes multi-agents temps-réels adaptatifs. La méthode se base sur le formalisme des automates temporisés [ALU 94], qui permet la spécification de systèmes comme un ensemble de processus concurrents dans lesquels on peut exprimer des contraintes de temps. Nous illustrons l'utilisation

de ce formalisme pour la modélisation d'un système d'agents ainsi que l'utilisation du *model checking* et de la simulation pour vérifier certaines propriétés du système et analyser son fonctionnement. Nous abordons enfin le passage semi-automatique de la spécification sous forme d'automates à des agents exécutables. Nous décrivons auparavant l'application cible et ses spécificités.

13.2. Présentation de l'étude de cas

L'application à laquelle nous nous intéressons est le projet J'ai dansé avec Machine [HUT 02], dans lequel il s'agit d'établir un dialogue multimodal et multimédia entre un danseur humain sur une scène et un système matériel et logiciel décentralisé. Cette application, que nous allons utiliser par la suite comme exemple pour la modélisation et la validation, constitue le fil conducteur du chapitre.

Dans le cadre de cet exemple, le système réparti a pour charge de capter par différents moyens la prestation du danseur, de l'analyser en temps-réel et finalement d'y répondre par la production d'animations visuelles projetées sur des écrans autour du danseur, la production de séquences musicales ou encore par la mise en mouvement d'objets physiques (robots ou autres).

Nous considérons que le système informatique d'interaction avec le danseur est constitué d'un ensemble de processeurs, dotés de moyens de communication et associés ou non à des capteurs (caméras vidéo, capteurs biométriques, capteurs de localisation, etc.) et/ou des effecteurs (écrans, haut-parleurs, moteurs, etc.). Sur chaque processeur peuvent s'exécuter un ou plusieurs agents, chacun spécialisé dans un certain type de traitement. Les données issues des capteurs doivent être traitées par différents agents avant de pouvoir être converties en actions au niveau des effecteurs.

L'action des agents consiste à analyser, synthétiser, transformer les données qu'ils reçoivent. Les nouvelles données ainsi produites par un agent sont ensuite transmises à d'autres agents chargés de poursuivre le traitement. Lorsque l'analyse est suffisamment avancée, ou que le temps disponible est insuffisant pour poursuivre cette analyse, les données sont finalement utilisées pour générer des images, des sons ou toute autre action grâce à des effecteurs. La figure 13.1 montre une vision très simplifiée de ce processus dans laquelle n'apparaît que la modalité de perception correspondant à une caméra vidéo qui filme la scène.

L'utilisation d'agents dans ce contexte se justifie de par la nature répartie des différents moyens (captation, traitement, action) mis en œuvre dans cette application [ATT 97] mais également et surtout pour rendre le système adaptatif dans différents contextes : lorsque des composants sont ajoutés ou supprimés, lorsque l'on souhaite modifier le comportement global du système ou encore lorsque le système ne respecte pas les contraintes de temps qui lui ont été assignées.

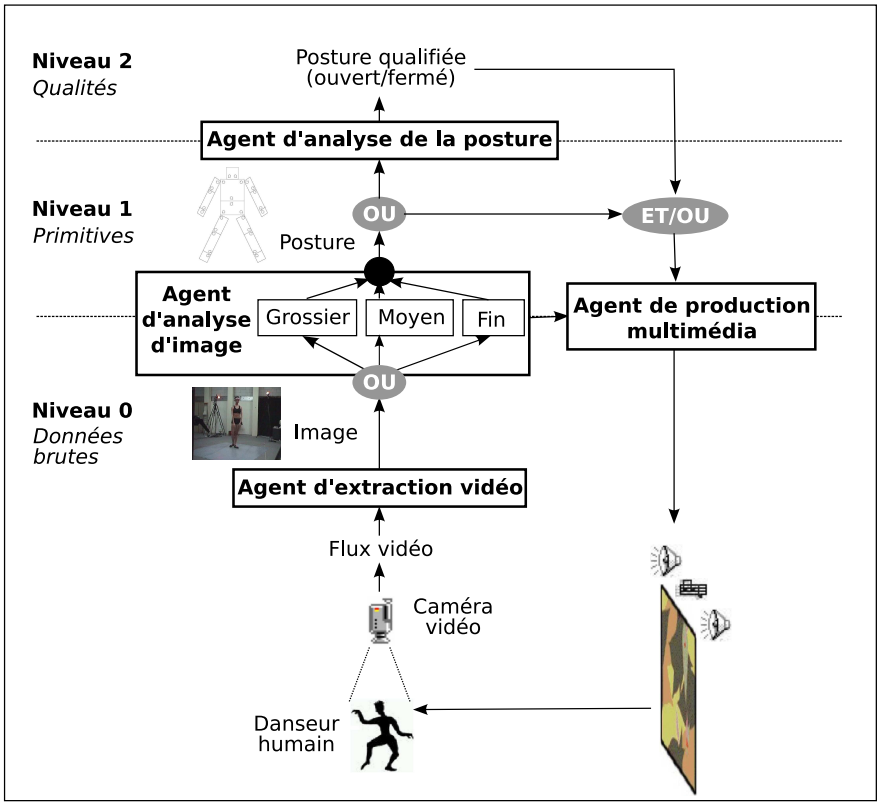


Figure 13.1. *Architecture globale de la chaîne de traitement pour la modalité visuelle dans le projet « J'ai dansé avec Machine »*

La contrainte de temps essentielle correspond au temps de réponse du système, c'est-à-dire le temps entre le moment où des données sont acquises par les capteurs et le moment où ces données provoquent une réponse au niveau des effecteurs. Ce temps de réponse doit naturellement être maintenu aussi faible que possible pour ne pas provoquer de délai désagréable entre une action du danseur et la réponse correspondante du système. Dans le même temps, l'analyse de la performance du danseur doit rester aussi précise et approfondie que possible.

Ces deux contraintes sont potentiellement contradictoires puisqu'une analyse précise et approfondie nécessite évidemment un temps de traitement plus long qu'une analyse grossière et superficielle. La qualité de l'analyse peut être mesurée selon deux dimensions complémentaires, la précision d'analyse (mesure plus ou moins précise

d'une caractéristique de la performance) et le niveau d'abstraction (ajout d'une étape supplémentaire de traitement pour obtenir de nouvelles mesures).

On peut formaliser ce compromis en exprimant une mesure de l'efficacité du système (qualité globale) comme une pondération de la qualité de l'analyse (avec quelle précision le système analyse-t-il la performance du danseur ?) et de la qualité temporelle (avec quelle tolérance le système respecte-t-il les contraintes de temps ?). Dans la mesure où l'on ne place pas dans un contexte de temps-réel dur, ces deux mesures de qualité quantifient par une valeur réelle, le respect relatif des contraintes logiques et de temps. On peut alors proposer la mesure de qualité globale suivante :

$$\text{qualité\_globale} = \alpha * \text{ qualité\_analyse} + \beta * \text{ qualité\_temporelle}.$$

Il ne s'agit donc pas d'assurer et de prouver le respect de propriétés de correction (de type vrai/faux) mais d'optimiser dynamiquement le critère de qualité globale ci-dessus.

L'objectif de la modélisation est alors de pouvoir tester le plus rapidement possible différentes stratégies pour le contrôle de la chaîne de traitement, et de fournir les outils pour automatiser au maximum la phase d'implantation jusqu'à un prototype opérationnel. La modélisation doit aussi pouvoir servir à vérifier *a priori* le respect par le système de certaines propriétés (temps de réponse global, non-blocage, ordre d'exécution des traitements, etc.). La simulation doit par ailleurs permettre d'obtenir des éléments d'appréciation de la qualité du compromis effectué entre la qualité du traitement effectué et le respect des contraintes de temps.

13.3. Automates temporisés

Dans cette étude de cas, nous utilisons les automates temporisés [ALU 94] qui constituent un formalisme relativement simple à manipuler mais possédant l'expressivité nécessaire pour la modélisation de systèmes concurrents réactifs, et pour lequel existent des outils puissants [LAR 98] de *model checking* et de simulation.

13.3.1. *Le modèle standard*

Un automate temporisé est un automate à états finis comportant une représentation du temps continu par l'intermédiaire de variables à valeurs réelles, positives ou nulles, appelées horloges, qui permettent d'exprimer des contraintes de temps.

De façon générale, un automate temporisé est représenté par un graphe constitué de places et d'arcs. Une place, associée à l'ensemble des valeurs d'horloge, correspond à un état du système. Un arc définit une transition entre ces états. Les contraintes de temps s'expriment au travers des contraintes d'horloges et peuvent apparaître sur les

places et sur les arcs. Une contrainte d'horloge est une conjonction de contraintes atomiques qui comparent la valeur d'une contrainte x, appartenant à l'ensemble fini d'horloges, à une constante rationnelle.

Chaque automate temporisé possède donc un nombre fini de places parmi lesquelles on distingue une place dite initiale. Dans chaque place, l'écoulement du temps est exprimé par la progression (uniforme) des valeurs d'horloges. Ainsi, dans une place, à tout instant, la valeur d'une horloge x correspond au temps écoulé depuis la dernière remise à zéro de x. A chaque place est associée une contrainte d'horloge, appelée invariant, qui doit être vérifiée pour que le système puisse être dans la place correspondante.

Les transitions sont instantanées. Elles sont conditionnées par des contraintes d'horloges, appelées gardes, et peuvent remettre certaines horloges à zéro. Elles peuvent aussi porter des étiquettes permettant des synchronisations. Un exemple d'automate temporisé et de son exécution possible en fonction du temps est représenté sur la figure 13.2.

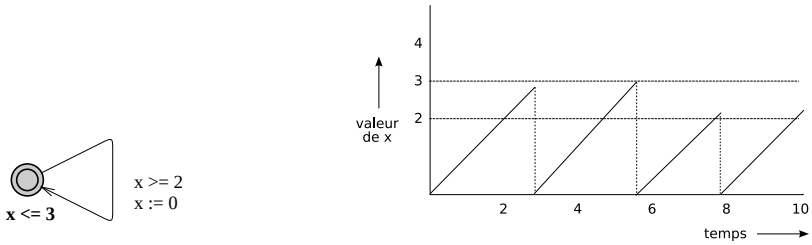


Figure 13.2. *Exemple d'un automate temporisé où x est une horloge. La garde* $x \geq 2$ *et l'invariant* $x \leq 3$ *impliquent que la transition ne pourra se déclencher qu'après* 2 *et avant* 3 *unités de temps passées dans la place*

Un système plus complexe, ou un agent, peut être représenté par un produit synchronisé d'automates. L'ensemble des places de cet automate résultant est le produit cartésien des places des automates qui le composent, l'ensemble des horloges est l'union des horloges et similairement pour les étiquettes. Chaque invariant dans l'automate résultant est la conjonction des invariants des places des automates qui le composent et les arcs correspondent à la synchronisation selon les étiquettes des arcs correspondants.

13.3.2. *Les extensions d'UPPAAL*

Une présentation plus détaillée de l'outil UPPAAL, que nous utilisons pour notre modélisation, peut être trouvée dans [LAR 98]. Nous rappelons rapidement les principales caractéristiques et extensions par rapport au modèle original.

Pour UPPAAL, un modèle consiste en un ensemble d'automates temporisés, qui communiquent par une synchronisation binaire, utilisant des étiquettes de transitions et une syntaxe du type émission/réception. Ainsi, par convention, l'étiquette $c!$ indique l'envoi par un émetteur d'un signal sur le canal c. Celui-ci est censé être synchronisé avec le signal de réception symbolisé par l'étiquette complémentaire $c?$ du côté du récepteur. L'absence d'étiquette de synchronisation indique une action interne de l'automate. Le formalisme permet également la manipulation de variables discrètes, et leur utilisation dans les gardes et les invariants.

L'exécution du modèle part de la configuration initiale (un état initial de chaque automate avec les valeurs des variables à zéro), et est une suite de configurations accessibles. Le changement de configuration peut se faire de trois façons :

– par écoulement du temps d'une durée d dans les états courants des composants, à condition que l'invariant de chacun de ces états reste satisfait. Dans la nouvelle configuration, les valeurs des horloges augmentent de d et les valeurs des variables discrètes ne changent pas.

– par synchronisation si deux actions complémentaires de deux composants sont possibles, et que les gardes associées aux transitions sont satisfaites. Les états correspondants sont modifiés dans la nouvelle configuration, et les valeurs des horloges et des variables discrètes sont modifiées selon les indications de remise à zéro et de mise à jour.

– par action interne si une telle action d'un composant est possible, elle peut être exécutée indépendamment des autres composants : l'état et les variables du composant sont modifiés comme dans le cas précédent.

Une autre caractéristique utile pour exprimer le « synchronisme » de mouvements est la notion d'états instantanés (*committed* pour UPPAAL), qui portent l'étiquette C dans les figures. Dans un tel état, aucun délai n'est permis, ce qui impose un mouvement immédiat du composant concerné. Ainsi, deux transitions liées par un état instantané seront exécutées sans délai intermédiaire.

UPPAAL permet de simuler l'exécution du système ainsi spécifié, détecter la présence des blocages (*deadlocks*) et de vérifier par *model checking* des propriétés d'accessibilité. Il peut ainsi répondre à des questions de type « à partir de son état initial, le système peut-il atteindre un état vérifiant une propriété donnée ? », « à partir de l'état initial, une propriété est-elle toujours valide ? » ou encore « à partir de l'état initial, le système peut-il atteindre un état donné dans un délai donné ? ».

13.4. Modélisation

Le comportement de base de nos agents consiste à traiter des données qui leur sont fournies en entrée pour les transformer et produire de nouvelles données en sortie. Ce traitement a une durée considérée comme fixe, dans des conditions idéales de fonctionnement, et doit être effectué de manière répétée. En pratique, ce temps sera encadré par des bornes pour tenir compte des différents facteurs susceptibles de modifier cette durée d'exécution. La modélisation correspondante, reproduite sur la figure 13.3 à gauche, devient ainsi non-déterministe, et sort du cadre classique des méthodes analytiques standards.

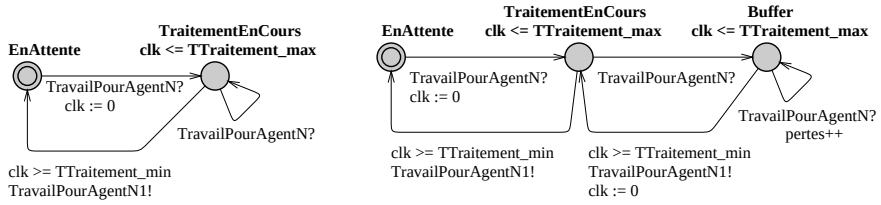


Figure 13.3. *Modèle d'agent simple (à gauche) et avec buffer (à droite) dédié au traitement de données au sein d'une chaîne de traitements*

Initialement dans l'état $EnAttente$, l'agent démarre son traitement à la réception du signal $TravailPourAgentN$, passant alors dans l'état $TraitementEnCours$. Il revient dans l'état $EnAttente$ à la fin de son traitement, soit après un temps compris entre $TTraitement_{min}$ et $TTraitement_{max}$, informant l'agent suivant qu'il peut démarrer son traitement (synchronisation sur le signal $TravailPourAgentN1$). L'introduction de bornes min et max sur le temps de traitement correspond ici à une incertitude sur le temps de traitement réel.

L'inconvénient de cette modélisation est que si une demande de traitement parvient à l'agent alors qu'il est en train de traiter la donnée précédente, la donnée correspondante est perdue. Cette boucle sur l'état $TraitementEnCours$ est cependant indispensable car son absence donnerait lieu à un blocage lorsque la situation que nous venons de décrire se produit. La solution est d'ajouter un état dans le modèle, équivalent à un *buffer* (voir figure 13.3, à droite).

A présent, lorsqu'une demande de traitement parvient à l'agent alors qu'il est dans l'état $TraitementEnCours$, ce dernier passe dans l'état $Buffer$, puis il revient dans l'état $TraitementEnCours$ à la fin de son traitement, pour en démarrer immédiatement un nouveau. Si une nouvelle demande de traitement parvient à l'agent alors qu'il est déjà dans l'état $Buffer$, la donnée correspondante est perdue, ce que l'on peut comptabiliser en incrémentant à chaque fois une variable $pertes$.

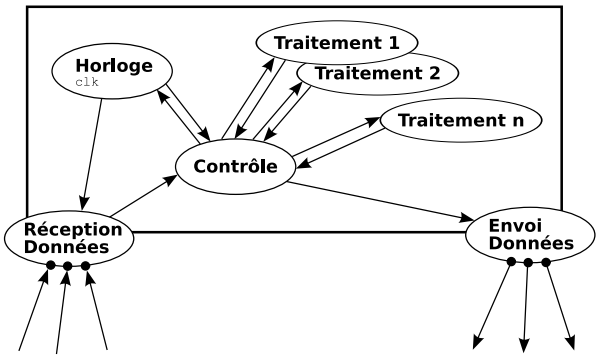
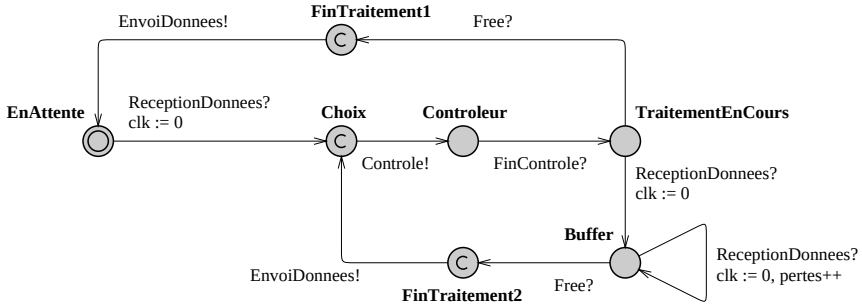


Figure 13.4. *Modèle d'agent temps-réel adaptatif : les données reçues sont étiquetées temporellement ; en fonction des contraintes à respecter, le contrôleur choisit les traitements à effectuer ; les données analysées sont transmises à un autre agent*

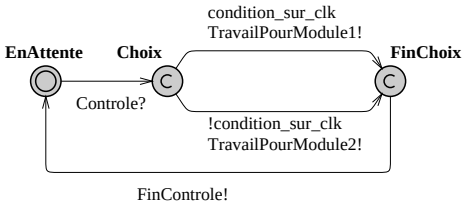
Ce premier type de modélisation correspond cependant à une architecture de traitement des données entièrement figée : un « agent » reçoit des données, les analyse puis les transmet à un autre « agent », toujours le même. La modélisation ne tient donc pas compte de l'adaptation souhaitée en fonction du temps effectif d'exécution des tâches et de la qualité du traitement à obtenir. Pour ce faire, il est nécessaire d'intégrer un module de contrôle au sein de l'agent pour lui permettre, en fonction des situations rencontrées, d'adapter le traitement qu'il effectue ainsi que le ou les agents auxquels il transmet les données traitées (voir figure 13.4).

Dans ce nouveau schéma de fonctionnement, l'agent qui reçoit une donnée lui associe une étiquette de temps lui permettant de raisonner par rapport à la date d'arrivée des données. En fonction de ces informations de temps et des contraintes à respecter, le module de contrôle choisit de faire analyser ces données par un ou plusieurs des modules de traitement qu'il a à sa disposition. Une fois analysées, les données peuvent alors être transmises à d'autres agents chargés de poursuivre l'analyse des données ou de les transformer en une réponse pour l'utilisateur.

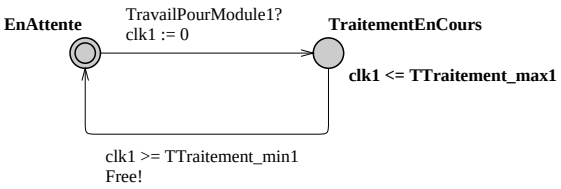
La figure 13.5 illustre la modélisation correspondante avec les automates temporisés dans le cas simplifié où l'agent doit choisir entre deux modules de traitement pour chaque donnée à analyser. Lorsqu'une donnée est reçue, l'horloge interne de l'agent est réinitialisée puis l'agent passe dans l'état $Choix$, ce qui revient à donner la main au module de contrôle. En fonction de la valeur de l'expression booléenne $condition\_sur\_clk$, le module de contrôle choisit de faire effectuer le traitement par l'un ou l'autre des deux modules de traitement puis rend la main. Lorsque le module



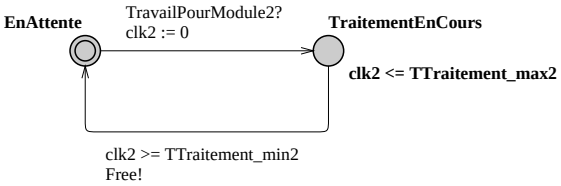
a)



b)



c)



d)

Figure 13.5. *Modèle d'agent (a), module de contrôle (b) et 2 modules de traitement (c et d)*

de traitement a terminé son travail, il informe l'agent grâce au signal $Free$. Ce dernier peut alors transférer les données à l'agent suivant dans la chaîne de traitement.

L'exemple précédent illustre l'utilisation des automates temporisés pour modéliser le comportement adaptatif d'un agent, en rapport avec le traitement qu'il doit effectuer. Le formalisme permet ainsi une décomposition de l'agent en modules dont on modélise le fonctionnement interne et les communications. De manière similaire, en transposant à l'échelle du système, il est possible de modéliser les agents et la circulation des données entre eux, tout en prenant en compte les contraintes de fonctionnement temps-réel. L'adaptation dynamique de l'architecture à base d'agents peut ainsi être modélisée ce qui permet de vérifier *a priori* son fonctionnement et de le simuler.

13.5. Vérification et Simulation

Le modèle décrit dans la section précédente est ensuite utilisé pour vérifier les propriétés du système par des techniques de *model checking* et de simulation. On montre en particulier que le système ainsi modélisé ne présente pas de blocage. Cela permet par ailleurs d'étudier par la simulation la performance de différentes stratégies d'adaptation. Cette étape permet d'ajuster le modèle initial jusqu'à l'obtention d'un modèle optimisé qui répond au cahier des charges.

Le modèle de contrôleur présenté au paragraphe précédent doit naturellement être instancié en explicitant le ou les critères qui impliquent le choix de l'un ou l'autre des modules de traitement. A titre d'exemple, nous présentons dans ce paragraphe sept stratégies possibles et détaillons les vérifications de propriétés et simulations que l'on peut mener avec chacune. Le contexte particulier dans lequel nous nous plaçons pour cette étude est donné sur la figure 13.6.

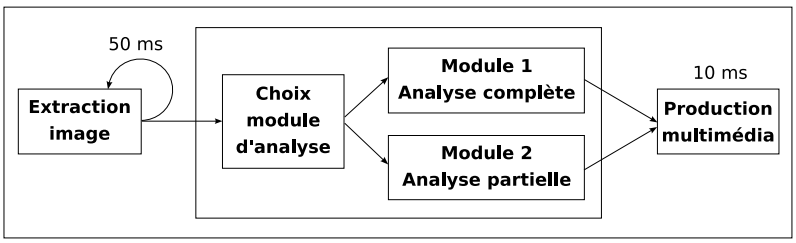


Figure 13.6. *Schéma de la chaîne de traitement simplifiée*

L'agent d'extraction d'image produit environ toutes les 50ms une image qui doit être traitée par l'agent d'analyse, soit avec un module qui fait une analyse complète,

soit avec un module qui ne fait qu'une analyse partielle, mais avec un temps de traitement réduit ($t_{traitement2} < t_{traitement1}$). De même que dans le modèle présenté dans la section 4, ces temps de traitement sont des approximations des temps de traitement réels, et sont encadrés chacun par des bornes min et max.

Le module de choix doit être conçu de manière à concilier deux contraintes potentiellement contradictoires : d'une part analyser toutes les images sans en perdre, c'est-à-dire essayer de respecter au maximum le rythme d'acquisition des données (mesure de qualité temporelle) ; d'autre part maximiser le nombre d'analyses complètes, c'est-à-dire éviter au maximum de devoir dégrader la précision du traitement (mesure de qualité d'analyse).

13.5.1. *Evaluation de stratégies*

Pour commencer, il est nécessaire de définir des stratégies qui pourront être utilisées comme référence. Les autres stratégies pourront alors leur être comparées. On choisit les trois stratégies de référence suivantes : la stratégie 1 utilise toujours le module 1 pour le traitement, de manière à maximiser le nombre d'images pour lesquelles une analyse complète est effectuée ; la stratégie 2 utilise toujours le module 2 pour le traitement de manière à minimiser le nombre d'images perdues ; la stratégie 3 alterne les deux modules, ce qui constitue un premier compromis possible entre analyse fine et respect des contraintes de temps.

A partir de ces stratégies de référence, une première approche peut être d'essayer de minimiser les pertes d'images. Pour ce faire, l'idée est d'anticiper, au moment du choix (t_{choix}), le moment où l'agent recevra une image à traiter alors qu'il a déjà une image en attente dans son *buffer* et qu'il n'a pas terminé l'analyse en cours (t_{perte}). Cela est rendu possible par l'arrivée régulière des images à traiter. Dans la stratégie 4, le module 1 sera alors choisi si et seulement si : $t_{traitement1} < t_{perte} - t_{choix}$. La stratégie 5 est encore plus restrictive et n'autorise un traitement par le module 1 que si l'image suivante peut être traitée sans perte par le module 2 ($t_{traitement1} + t_{traitement2} < t_{perte} - t_{choix}$).

Une autre approche est d'essayer de maximiser le nombre d'analyses complètes. Pour ce faire, on peut relâcher la contrainte précédente et autoriser l'utilisation du module 1 même si cela doit nécessairement conduire à la perte d'une image. Dans la stratégie 6, le module 1 ne sera choisi que si $t_{traitement1} < (t_{perte} - t_{choix}) * coef$, où $coef$ détermine la limite de tolérance admise. Par la suite, la stratégie 6 correspondra à une valeur de $1,25$ pour $coef$.

Finalement, pour que l'analyse des données soit suffisamment bonne, on peut considérer qu'il est nécessaire d'effectuer une analyse complète au moins une fois

toutes les n images. Pour cela, le contrôleur de la figure 13.5 doit être légèrement modifié. Il fonctionnera en deux étapes : il commence par vérifier qu'il y a eu moins de $n-1$ images traitées depuis le dernier appel au module 1 ; si la limite est atteinte, le module 1 est appelé ; si ce n'est pas le cas, le contrôleur choisit le module en utilisant la stratégie 4.

13.5.2. *Résultats*

L'utilisation de techniques de *model checking* permet de prouver automatiquement que certaines propriétés considérées comme importantes pour le système resteront toujours valides quelle que soit l'évolution du système. Ainsi, pour chaque stratégie, on a pu vérifier formellement au moyen d'outils existants comme UPPAAL [LAR 98] que certaines propriétés sont respectées par le système. Les propriétés sont exprimées par des formules de logique temporelle puis soumises au moteur de vérification d'UPPAAL. Dans un premier temps, il faut s'assurer qu'il n'y a pas de deadlocks dans le modèle : A[ ] not deadlock. L'utilisation de ces techniques permet par ailleurs de vérifier des propriétés exprimant la qualité du système modélisé, par exemple :

– qu'aucune image n'est perdue : A[ ] $pertes$ == 0 ;

– que la proportion d'appels au module 1 est supérieure à un seuil donné :
A[ ] $(nb1 * 100/(nb1 + nb2 + pertes)) > 50)$.

De manière complémentaire, l'utilisation de la simulation permet d'obtenir une évaluation empirique des performances du système en termes de qualité logique et temporelle, en fonction des caractéristiques des modules de traitement et du type de stratégie appliquée. En simulant le fonctionnement du système pendant une centaine de cycles, on a pu évaluer expérimentalement les proportions d'images perdues et d'images analysées complètement en fonction des temps de traitement $t_{traitement1}$ et $t_{traitement2}$, comme illustré dans la figure 13.7.

Grâce à ces expérimentations, il est possible d'évaluer l'efficacité de chaque stratégie pour assurer une certaine qualité de traitement, ce qui permet d'envisager un niveau supplémentaire dans le contrôle de l'agent d'analyse des images, correspondant à une « méta-stratégie » qui adapterait dynamiquement la stratégie de choix en fonction des contraintes et des objectifs assignés.

De même, en se plaçant au niveau du système dans son ensemble, il est possible de modéliser différentes stratégies pour la réorganisation adaptative de l'architecture de traitement des données puis de tester ces stratégies par la vérification et la simulation.

La figure 13.10 montre une comparaison de différentes stratégies pour ce critère, pour les mêmes conditions que dans la figure 13.7. On peut y observer, en particulier, qu'en fonction des temps caractéristiques de traitement des modules et en fonction

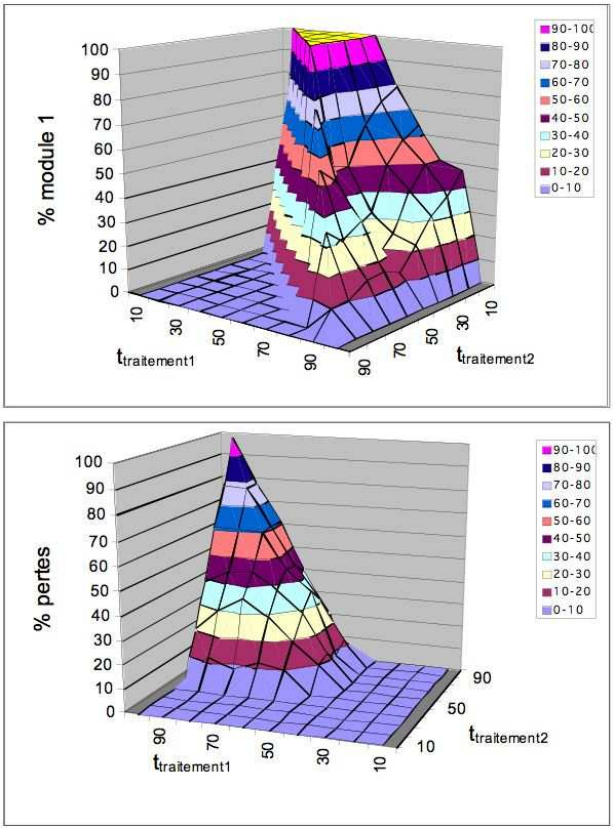


Figure 13.7. *Le pourcentage d'images analysées avec le module 1 (en haut) et pourcentage d'images perdues (en bas) obtenus avec la 4ème stratégie pour différentes valeurs de temps de traitement pour les modules 1 et 2*

des contraintes temps-réel fixées, les agents doivent adapter leur stratégie de contrôle. Cette façon de faire les autorise à adapter leur comportement dynamiquement lorsque les conditions extérieures changent. Ceci peut être modélisé par les automates de la figure 13.11.

13.5.3. *Retour sur la conception*

Comme nous l'avons vu dans les sections précédentes, les différentes stratégies choisies ne sont pas équivalentes par rapport à la correction logique (avec quelle fréquence l'analyse complète est-elle faite ?) et temporelle (quel pourcentage de données

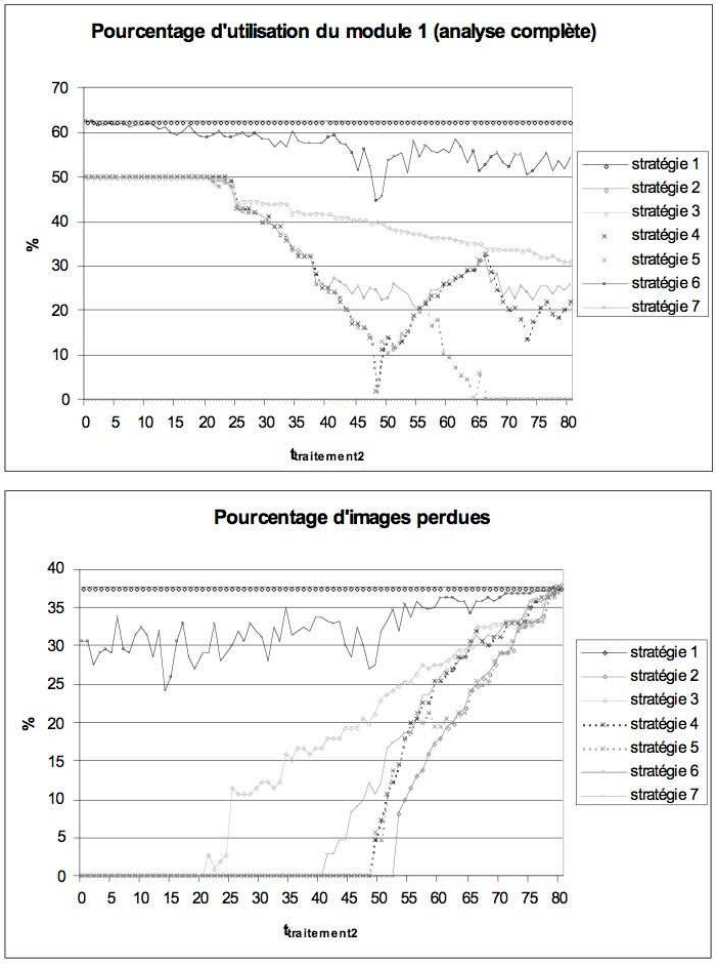


Figure 13.8. *Comparaison des 7 stratégies pour $t_{traitement1} = 80ms$, par rapport au pourcentage d'analyses complètes et au pourcentage d'images perdues, en faisant varier $t_{traitement2}$; le caractère irrégulier de certaines courbes reflète le non-déterminisme du système sous-jacent*

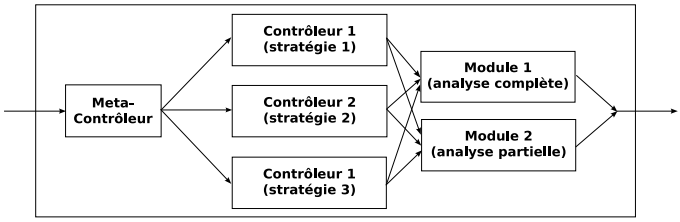


Figure 13.9. *La stratégie de choix du module de traitement est choisie dynamiquement par un méta-contrôleur, en fonction des paramètres des modules de traitement et des contraintes temps-réel*

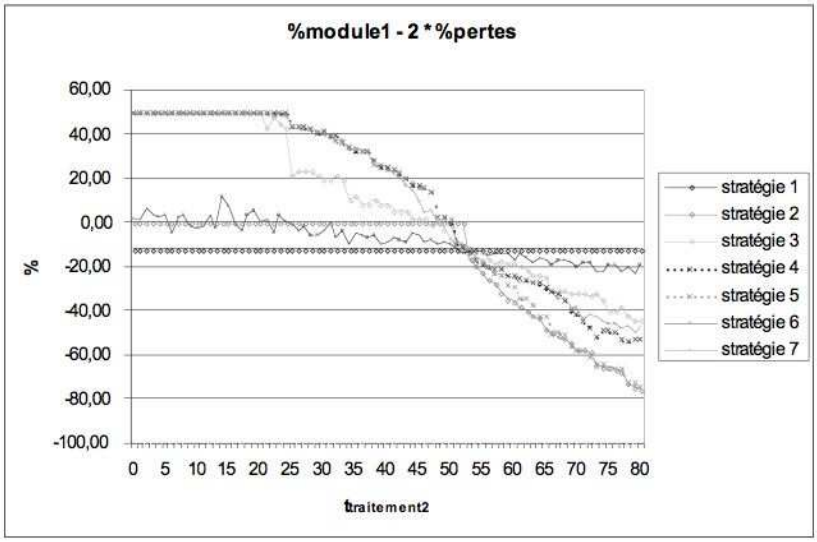


Figure 13.10. *Comparaison de sept stratégies pour $t_{traitement1} = 80ms$, $\alpha = 1$ et $\beta = 2$; pour les valeurs de $t_{traitement2}$ inférieures à 20, les meilleures stratégies sont 3, 4, 5 et 7 ; pour les valeurs de $t_{traitement2}$ entre 20 et 50, les meilleures stratégies sont 4 et 5 ; pour les valeurs autour de 50, toutes les stratégies sont pratiquement équivalentes, mais la meilleure est la 2 (elle appelle toujours le module 2) ; enfin, pour les valeurs au-dessus de 55, la meilleure stratégie est la 1 (elle appelle toujours le module 1)*

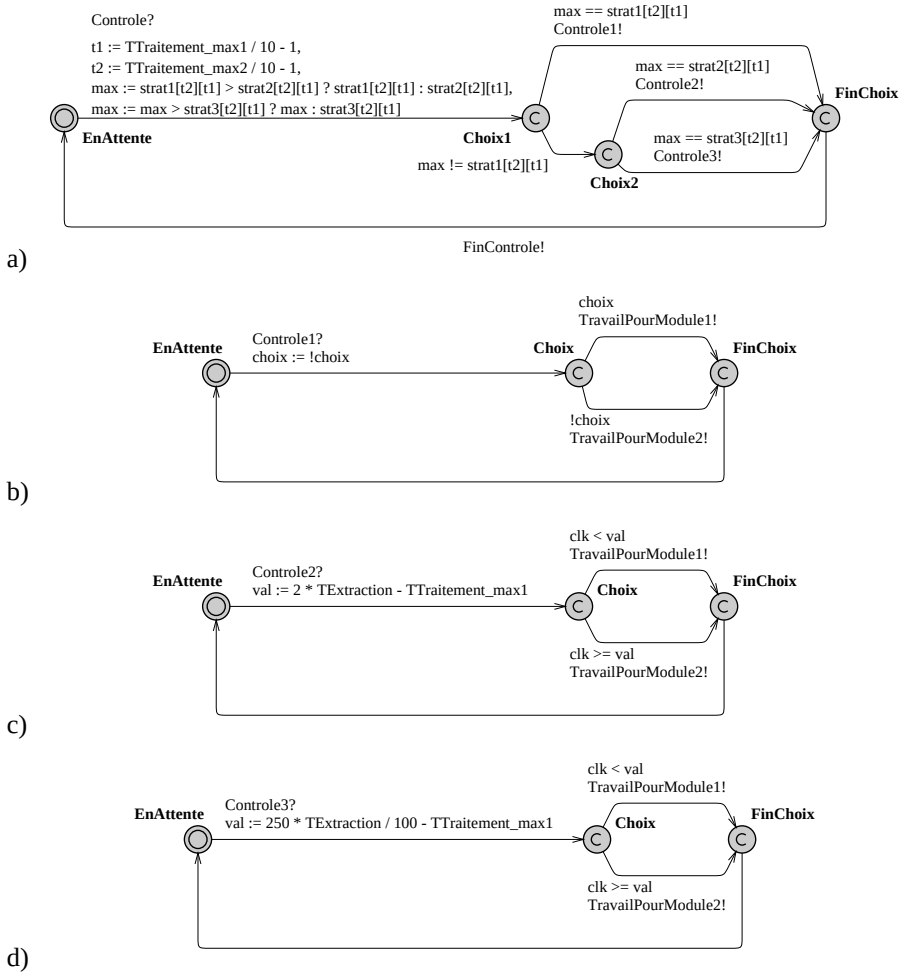


Figure 13.11. *Le méta-contrôleur modélisé par un automate temporisé (a) ; la meilleure stratégie (max) est calculée en fonction des temps caractéristiques de traitement des modules $t1$ et $t2$; le module du contrôleur correspondant est alors choisi parmi les différentes stratégies possible : le contrôleur b) correspond à la stratégie 3, le c) à la stratégie 4 et le d) à la stratégie 6*

est perdu ?). En fonction des valeurs de $t_{traitement1}$ et $t_{traitement1}$, la « meilleure » stratégie n'a pas toujours été la même, ce qui signifie qu'aucune stratégie n'est universelle. Evidemment, la « meilleure » dépend des critères que nous avons pu retenir pour évaluer son efficacité.

Dans ce contexte, ce qui semble intéressant, c'est la possibilité de changer de stratégie lorsque les conditions changent : ou bien parce que les temps de traitement caractéristiques du module ont changé (à cause d'une répartition différente de la charge du processeur) ou bien parce que les contraintes temps réel ont été modifiées.

Ceci peut être pris en compte en ajoutant un module supplémentaire, un méta-contrôleur, qui serait chargé de passer dynamiquement d'un contrôleur (associé à une stratégie particulière) à un autre, en fonction des conditions d'exécution. Ceci est illustré dans la figure 13.9. A nouveau, différentes stratégies peuvent être proposées pour ce méta-contrôleur. Nous allons expliquer uniquement comment l'une d'elles pourrait être conçue. On peut considérer, par exemple, le critère défini par la formule suivante :

$$efficacite = \alpha * \%module1 - \beta * \%pertes$$

où α et β correspondent respectivement aux poids affectés à la qualité logique et à la qualité temporelle.

13.6. Génération automatique de code

Une fois que le modèle du système de traitement de données a été validé à la fois formellement et expérimentalement, il reste encore à le transformer en un modèle exécutable.

Pour ce faire, une première idée pourrait conduire à implémenter chaque automate sous forme d'un thread, puisqu'il s'agit de modèles de processus concurrents. Pour un même agent, modélisé par plusieurs automates, cela pourrait cependant conduire à une multiplication des problèmes de synchronisation et se traduire par une baisse sensible de performances, ce qui peut être gênant pour un système réactif. La solution retenue est de ne considérer qu'un seul thread par agent, plutôt qu'un thread par automate. Le système entier est alors implanté sous forme d'une application multi-threadée.

13.6.1. *Synchronisation d'automates et optimisation*

Une première étape consiste donc à faire le produit synchronisé des automates modélisant un même agent puis à transformer l'automate ainsi obtenu en squelette d'application. Le compilateur que nous avons développé réalise ce produit synchronisé. Si l'on fait, par exemple, le produit synchronisé des quatre automates modélisant

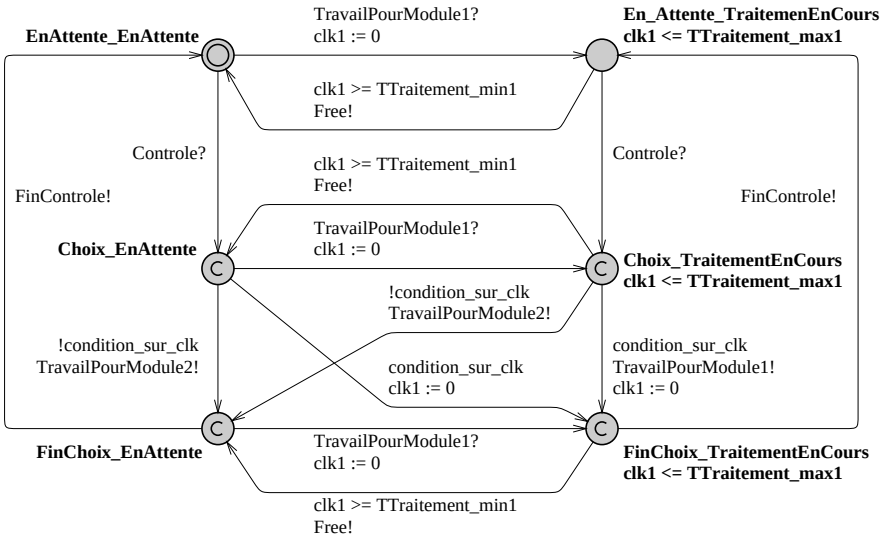


Figure 13.12. *Automate obtenu en effectuant le produit synchronisé de deux des quatre automates de la figure13.5, correspondant au contrôleur et au premier module de traitement*

l'agent de traitement reproduit à la figure figure13.5, l'automate résultant comporte 84 états et 388 transitions !

Lorsque nous réalisons le produit synchronisé d'automates, nous distinguons cependant deux types de signaux. Les signaux « locaux » ou « internes » qui n'apparaissent que dans les automates dont on fait le produit synchronisé, correspondent à une synchronisation entre ces automates. Les signaux « globaux » ou « externes » qui apparaissent également dans d'autres automates, correspondent à une synchronisation avec ces autres automates.

Les transitions qui apparaissent dans le produit synchronisé et qui sont étiquetées avec des signaux locaux n'ont aucune chance d'être franchies puisque ces signaux ne sont présents dans aucun autre automate. Les transitions correspondantes peuvent donc être supprimées (voir figure13.13.a). Dans l'exemple précédent, on n'a plus alors que 84 états et 120 transitions.

Suite à la suppression de ces transitions, certains états deviennent inaccessibles, et peuvent également être supprimés ainsi que les transitions éventuelles qui en partent. D'autres états correspondent à des interblocages (deadlocks) parce qu'ils n'ont que des transitions entrantes ou en boucle. Si l'on s'est assuré au départ que le système n'avait

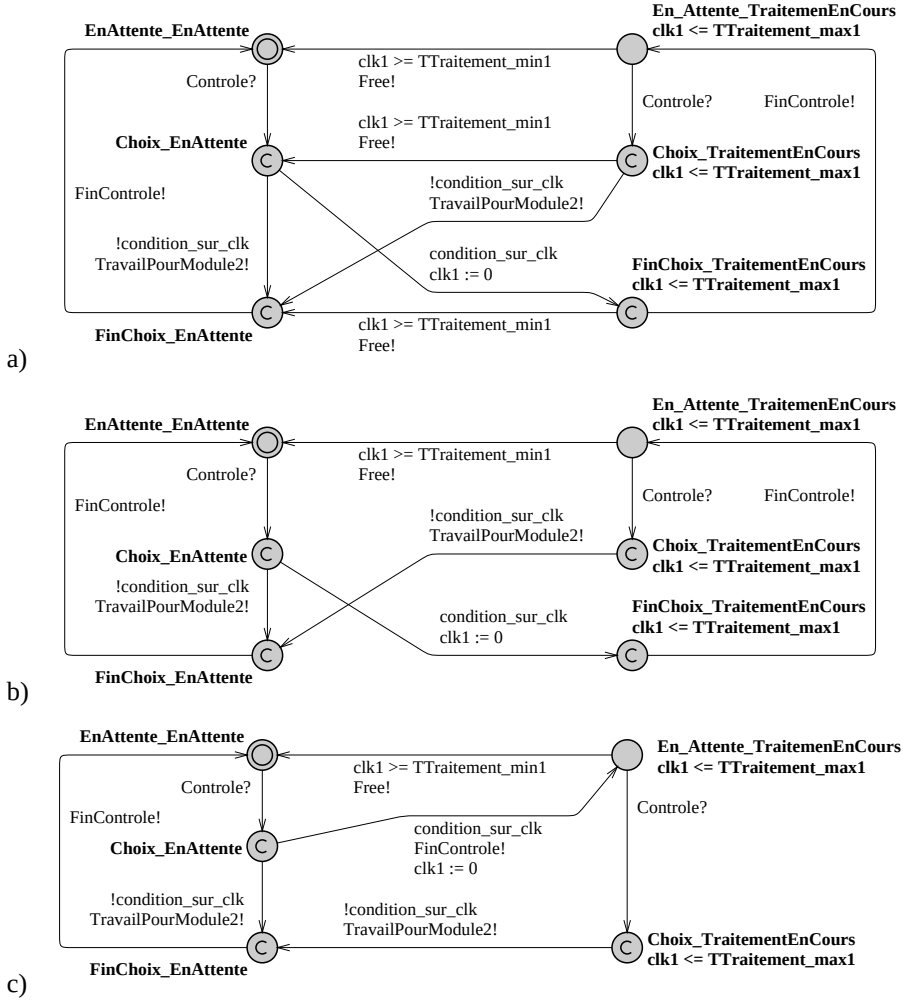


Figure 13.13. *Produits synchronisés optimisés après : a) suppression des signaux locaux ; b) suppression des transitions inutiles sortant d'états instantanés ; c) compression d'états instantanés*

pas d'interblocage, on peut également supprimer ces états ainsi que les transitions qui y arrivent. Par extension, tous les états qui ne sont pas atteignables à partir de l'état initial (problème de couverture dans un graphe orienté) peuvent être supprimés. Toujours pour le même exemple, on n'a plus alors que 11 états et 16 transitions.

Le compilateur effectue ensuite une optimisation liée aux propriétés des états instantanés, pour minimiser la taille de l'automate. Cette optimisation se justifie par le fait que, lorsque l'on fusionne deux états dont l'un seulement est un état instantané, seules les transitions sortant de l'état instantané sont autorisées. Il est facile de voir en effet que la fusion de deux états dont l'un au moins est un état instantané, conduit à un état qui est également un état instantané.

Par ailleurs, puisqu'il est interdit de laisser le temps s'écouler dans un état instantané, les seules transitions autorisées quand on se trouve dans l'un de ces états sont les transitions permettant de quitter l'état. Ce sont donc celles qui, dans l'automate de départ, permettaient de sortir de l'état instantané. Les autres transitions n'ont aucune chance d'être franchies et peuvent donc être supprimées (voir figure 13.13.b). De la même manière que précédemment, cela peut conduire, dans certains cas, à des états qui ne sont plus connectés au reste de l'automate et peuvent eux aussi être supprimés.

Un dernier type d'optimisation correspond aux états instantanés qui n'ont qu'une seule transition sortante, ce qui signifie qu'il n'y a qu'une seule possibilité pour poursuivre l'exécution de l'automate. Dans certaines conditions, on peut alors fusionner cet état avec l'état suivant, en fusionnant également la transition sortante avec chacune des transitions entrantes (voir figure 13.13.c).

En particulier, si la transition sortante porte un signal, aucune des transitions entrantes ne doit porter de signal, car une transition ne peut pas porter simultanément deux signaux. La nouvelle garde est alors la conjonction des gardes portées par les transitions et les affectations sont combinées en séquence. Pour le produit synchronisé des quatre automates de la figure 13.5, on obtient finalement un automate à 8 états et 13 transitions (figure 13.14).

13.6.2. *Implantation*

A l'issue de l'étape de l'étape précédente, chaque agent est modélisé par un automate unique que l'on peut alors traduire sous forme exécutable en plusieurs étapes. Nous nous sommes appuyés, pour la génération de code, sur la plate-forme JADE [BEL 99] qui propose un ensemble de bibliothèques et de services pour l'implantation de systèmes à base d'agents qui se conforment à la norme FIPA [FIP 05].

Le compilateur que nous avons développé analyse le fichier XML produit par UPPAAL et génère automatiquement, à partir de chacun des automates, le squelette d'un agent.

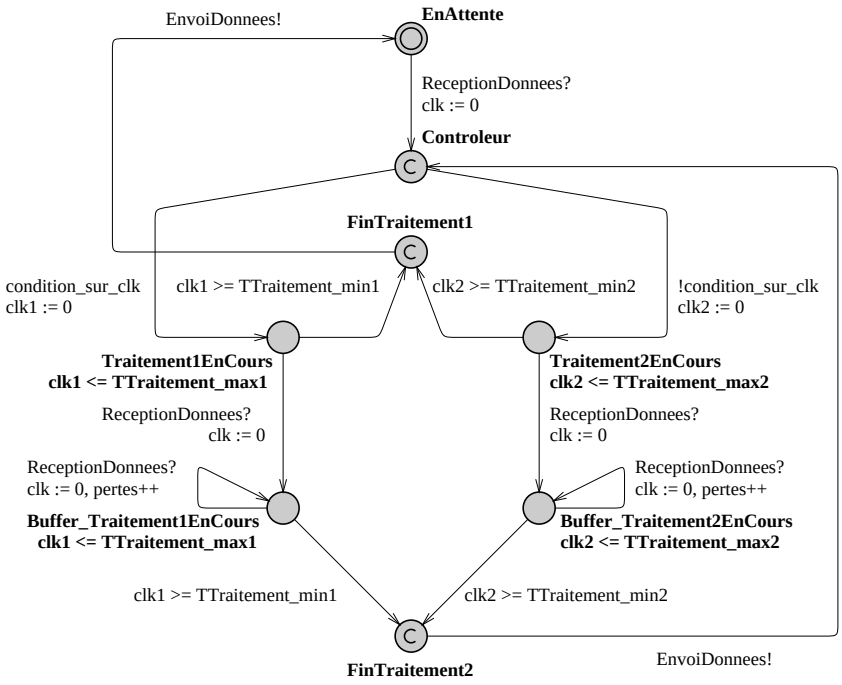


Figure 13.14. *Automate obtenu en effectuant puis en optimisant le produit synchronisé des quatre automates de la figure 13.5*

A chaque automate peut être associé un ensemble de variables discrètes. Ces déclarations étant spécifiques à un automate, le compilateur les transforme en autant d'attributs pour l'agent correspondant. De même, des horloges peuvent être associées aux automates, dont l'avancement sera calculé par rapport à l'horloge interne de l'agent. Chaque affectation d'horloge correspond à l'initialisation de l'origine des temps par rapport à cette horloge interne et chaque lecture correspond à la mesure du temps écoulé depuis cette origine.

Dans le même temps, des variables discrètes et horloges globales peuvent être définies. Puisque l'on souhaite que l'application finale soit répartie (des agents qui fonctionnent sur des machines différentes) et décentralisée (sans contrôle centralisé de la part d'un agent particulier), ces données globales sont potentiellement problématiques.

La présence de telles données globales dans le modèle peut se justifier de différentes manières : dans un objectif de mise au point et de simulation, il peut par exemple

être utile de compter le nombre de fois où un automate est passé par une place donnée ; par ailleurs, du fait que le formalisme des automates temporisés ne permet pas le passage de données avec les signaux, il peut être utile d'utiliser des variables globales à cet effet (l'automate émetteur modifie la variable globale lors de l'envoi du signal, l'automate récepteur lit cette même variable à la réception du signal).

Dans le premier cas, il s'agit d'informations liées à la mise au point du modèle, et qui peuvent donc être supprimées sans difficulté lors du passage à l'implantation. Dans le second cas, il s'agit d'un envoi de message entre agents, pour lequel on substitue la modification de variable globale par un paramètre transmis avec le message. Au moment de l'implantation, nous n'autorisons que ce deuxième type d'utilisation de variable globale.

Notre compilateur identifie tout d'abord l'ensemble des données globales (horloges et variables) présentes dans le modèle et alerte le concepteur des problèmes potentiels liés à ces données. Le concepteur a alors le choix de modifier le modèle ou de continuer la phase d'implantation. Dans ce dernier cas, le compilateur identifie tous les signaux de synchronisation entre les automates.

Puisque chaque automate modélise un agent, les signaux peuvent être assimilés à des messages échangés entre les agents. Dans le cas de notre implantation avec la plate-forme JADE, il s'agit de messages asynchrones de type ACL (*Agent Communication Language*). Lorsque les arcs porteurs de ces signaux sont également associés à des affectations de variables globales, ces variables sont transmises comme paramètres du message, avant d'être lues par le destinataire.

De manière plus globale, l'automate temporisé est transcrit sous forme d'un automate à états finis. Les places dans lesquelles il est nécessaire de laisser s'écouler le temps (places associées à un invariant de type $t \leq t_{max}$, avec une transition sortante associée à une garde de type $t \geq t_{min}$) sont supposées correspondre à un traitement. Cela signifie que l'agent n'est pas censé attendre dans la place un temps compris entre t_{min} et t_{max} mais qu'il est censé effectuer un traitement dont on a estimé la durée entre t_{min} et t_{max}. L'appel au module de traitement doit être ajouté manuellement à l'issue de la phase de génération automatique.

13.6.3. *Une boîte de réception*

On peut observer tout de suite qu'un agent réel qui est en train d'effectuer son traitement monopolise toutes ses ressources pour faire avancer son traitement, n'est plus attentif aux signaux extérieurs et ne devrait donc quitter l'état correspondant que quand il a terminé son traitement. Cela signifie que quand un agent se trouve dans une place où il effectue un traitement, il ne devrait pas réagir à la réception de signaux.

Cette remarque est cohérente avec l'implantation sous JADE puisque ce ne sont pas les agents qui gèrent par eux-mêmes la réception des message. C'est en effet la plate-forme qui reçoit les messages et les place dans les boîtes de réception des agents correspondants. Il apparaît donc nécessaire de modifier la modélisation en séparant la boîte de réception des messages du reste de l'agent.

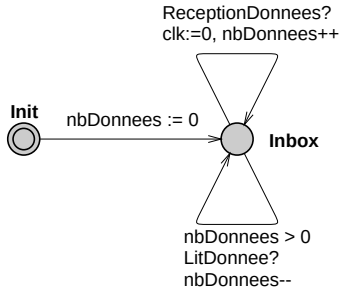


Figure 13.15. *Modélisation de la boîte de réception des messages : la boîte peut recevoir des messages (ReceptionDonnees ?) ; l'agent peut lire les messages dans la boîte (LitDonnee)*

Du coup, il n'est plus nécessaire pour l'agent de gérer la réception des messages et l'on peut donc fusionner deux par deux les places correspondant aux traitements par l'agent. L'automate se simplifie tel qu'il apparaît sur la figure13.16.

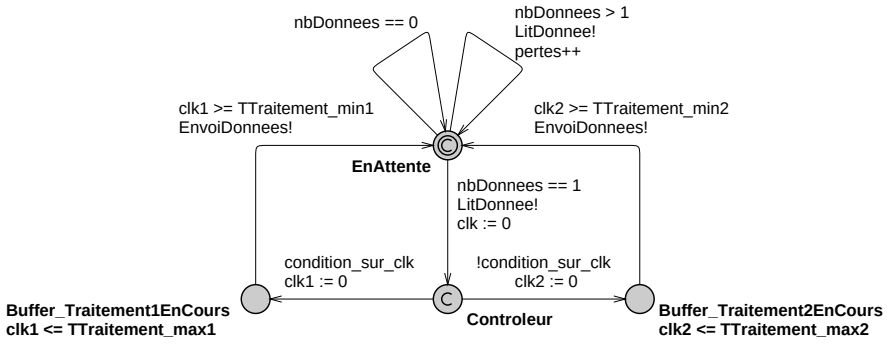


Figure 13.16. *Automate de l'agent d'analyse des images simplifié après introduction de l'automate Inbox*

13.6.4. *Validation*

Une fois l'implantation finalisée, il est nécessaire de s'assurer que le comportement du système implanté correspond bien au comportement du modèle. Pour cela, nous avons conçu un outil qui enregistre les dates de tous les changements d'états chez les agents. Une interface nous permet ensuite de visualiser l'évolution du système de la même manière qu'avec le simulateur d'UPPAAL.

Il est ainsi apparu très rapidement que le comportement du système implanté était très différent. La raison principale tient au modèle de communication de JADE qui est lui-même très différent du modèle des automates temporisés. L'échange de signaux dans les automates temporisés s'effectue en effet de manière synchrone et instantanée alors que la communication agent s'effectue par échange asynchrone de messages et nécessite un temps d'acheminement à travers le réseau.

Pour modéliser un modèle asynchrone, il est nécessaire d'ajouter un automate de type messager qui intercepte le signal émis, attend pendant une durée variable comprise entre deux bornes min et max, puis réémet le signal vers son destinataire initial. La figure13.17 illustre la modélisation correspondante avec les automates temporisés. Les bornes t_{ach_min} et t_{ach_max} ont pu être évaluées grâce à l'enregistrement des changements d'états chez tous les agents.

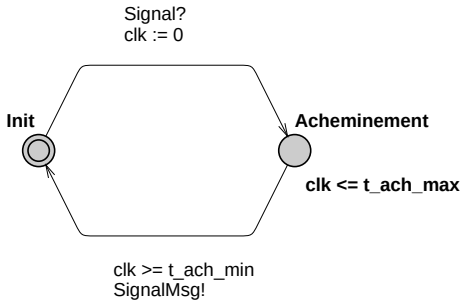


Figure 13.17. *Modèle de communication asynchrone integrant un automate messager*

Grâce à cette nouvelle modélisation, on parvient finalement à un comportement du système implanté qui est qualitativement identique au comportement du modèle. Il aurait naturellement été plus efficace, du point de vue du fonctionnement en temps-réel, d'adopter une plate-forme plus adaptée que la plate-forme JADE, notamment en termes de communication entre agents. Le travail réalisé permet cependant de démontrer la possibilité offerte par les automates temporisés d'adapter la modélisation en

fonction des caractéristiques de la plate-forme utilisée pour l'implantation, et donc d'avoir une modélisation réellement pertinente par rapport au système final implanté.

13.7. Conclusion

L'étude de cas développée dans ce chapitre présente un exemple de modélisation d'un système d'agents, à la fois sous l'angle du traitement de données et sous l'angle de la reconfiguration dynamique de la chaîne de traitement. L'approche proposée permet de contrôler le compromis de qualité entre des contraintes antagonistes, en choisissant de manière adaptative les modules de traitement à utiliser par un agent et les connexions entre les agents.

Le formalisme des automates temporisés permet ainsi de modéliser de façon modulaire un contrôleur d'agent, pour raisonner et prendre des décisions en fonction des objectifs assignés. Le formalisme se prête bien par ailleurs à l'utilisation de techniques de *model checking* et de simulation, ce qui permet d'obtenir une connaissance théorique du comportement du système et de tester *a priori* différents comportements d'agents. Même si le *model checking* est limité à la vérification de systèmes de petite taille, les techniques de simulation restent applicables quelle que soit la taille du système.

Enfin, le formalisme peut constituer le support de la génération semi-automatique du squelette d'un premier prototype. Le passage à l'implantation a permis d'illustrer le besoin de prendre en compte, dans la modélisation du système, les spécificités de la plate-forme sur laquelle le système doit être développé et sur laquelle il doit s'exécuter.

13.8. Bibliographie

[ALU 94] ALUR R., DILL D. L., « A Theory of Timed Automata », *Theoretical Computer Science*, vol. 126, n°2, p. 183-236, 1994.

[ATT 97] ATTOUI A., *Les systèmes multi-agents et le temps-réel*, Eyrolles, Paris, 1997.

[BEL 99] BELLFEMINE F., POGGI A., RIMASSA G., « JADE - A FIPA-compliant agent framework », *Proceedings of Practical Applications of Intelligent Agents and Multi-Agent Technology*, 1999.

[FIP 05] FIPA, « Foundation for Intelligent Physical Agents », http ://www.fipa.org, 1996-2005.

[HUT 02] HUTZLER G., GORTAIS B., JOLY P., ORLAREY Y., ZUCKER J.-D., « J'ai dansé avec machine ou comment repenser les rapports entre l'homme et son environnement », *Proceedings of JFIADSMA'2002*, Hermès Science Paris, 2002.

[HUT 04] HUTZLER G. KLAUDEL H., WANG D. Y., « Towards Timed Automata and Multi-Agent Systems », *Proceedings of FAABS'2004, LNCS 3228*, Springer-Verlag, 2004.

[LAR 98] LARSEN K. G., PETTERSSON P., YI W., « UPPAAL in a Nutshell », *Springer International Journal of Software Tools for Technology Transfer*, vol. 1/2, n˚1, p. 134-152, 1998.

Index