Validation par évaluation sur un modèle : méthodes et algorithmes

Jean-Claude Fernandez

DOCUMENT DE PRESENTATION DES TRAVAUX

Pour obtenir

l'Habilitation à diriger des recherches
en Informatique

Présentée le 25 Octobre 1996 devant

l'Université Joseph Fourier

Par

Jean-Claude Fernandez

Jury :

M.	Jacques Voiron	Président
MM.	André Arnold	Rapporteurs
	Pierre Wolper	
	Gérard Boudol	
MM.	Sacha Krakowiak	Examinateurs
	Joseph Sifakis	

Validation par évaluation sur un modèle : Méthodes et Algorithmes

Avant Propos

Ce document se veut une synthèse des activités de recherche que je mène depuis la fin de ma thèseΓen mai 1988Γqui portent sur la vérificationΓetΓdepuis un peu plus d'un anΓsur le test de protocoles.

J'ai commencé mes activités de recherche en 1984Γau sein du projet SpectreΓdirigé par Joseph SifakisΓd'abord au laboratoire de Génie InformatiqueΓpuis à Vérimag. Une des préoccupations majeures du projet est la modélisation et la validation de systèmes critiques. Rentre dans ces préoccupations le développement d'outils de vérification. Mon activité s'est organisée d'abord autour du logiciel ALDÉBARANΓque j'ai réalisé pendant ma thèseΓpuis de la plate-forme de vérification CADP (CAESAR-ALDÉBARAN DISTRIBUTION PACKAGE). CADP est né de la conjonction du compilateur pour le langage LOTOSΓréalisé par Hubert GaravelΓet d'ALDÉBARAN.

La recherche englobant des travaux de réalisation logicielle ne peut s'appuyer que sur une activité collective. Je tiens à associer à ce travail un certain nombre de chercheurs. En particulierΓ Laurent Mounier et Alain Kerbrat dont j'ai dirigé les thèses. Ils ont contribué au développement de la partie vérification de CADPΓen continuant et en amplifiant les résultats obtenus autour d'ALDÉBARAN. Nos activités autour du test résultent d'une forte collaboration entre le projet Pampa de l'Irisa (Claude JardΓThierry Jéron et César Viho) et le projet Spectre.

Pendant cette dizaine d'annéesΓj'étais enseignant-chercheur à l'université Joseph Fourier. En tant qu'enseignantΓje me suis plus particulièrement intéressé au thème *langages et traducteurs*Γ et au thème *architectures logicielles et matérielles*. Ces enseignements reposent sur l'étude des concepts des langages de programmationΓde l'architecture cible et des processus de traduction. Cela va de la formalisation des concepts (donner une sémantique formelleΓsupport de l'analyse de programmes) à l'étude de la machine cible (pour permettre l'efficacité du processus de traduction d'un programme source en code exécutable). Je tiens à souligner les aspects complémentaires entre mes activités d'enseignement et de recherche. Le lien entre compilation et vérification est assez évident ; dans les deux casΓon développe des techniques formelles sur un modèle de programmeΓsoit pour vérifier des propriétésΓsoit pour optimiser le code généré.

Le fil conducteur de ce rapport est l'étude et la mise en œuvre d'algorithmes sur les systèmes de transitions étiquetéesΓqui sont utilisés ici comme modèles de programmes. Les articles qui servent de base à l'élaboration de ce document sont [Fer90ΓBFH+92ΓFer93ΓFJJM92ΓCFG95Γ FJJV96bΓFJJV96aΓFKM93ΓFM95].

Table des matières

1 Introduction

1.1 Le contexte scientifique

L'évolution technologique a conduit au développement de systèmes informatiques de plus en plus complexes et souvent critiques de par leurs fonctions mettant en jeu des vies humaines ou des budgets importants. Des exemples de tels systèmes se trouvent dans des domaines comme l'aéronautiqueΓl'énergieΓl'espaceΓles télécommunicationsΓle transportΓetc. Ces systèmes sont souvent interactifs ou réactifs [HP85]Γc'est-à-dire qu'ils reçoivent de manière continue des entrées et émettent de même des sorties. De plusΓils peuvent aussi être parallèles.

La complexité de ces systèmes pose des problèmes nouveaux et difficiles de mise au point et d'assurance de bon fonctionnement qui mettent en défaut les méthodes traditionnelles de conception et de validation. L'utilisation de méthodes formelles apparaît de plus en plus comme susceptible de contribuer à résoudre ces problèmes.

SchématiquementΓ on peut identifier plusieurs étapes intervenant dans le développement de systèmes :

- *l'expression des besoins*Γqui correspond à la formulation en langage naturel du problème à résoudreΓ
- *la conception* logicielle et matérielleΓqui réalise une solution du problèmeΓ
- *la validation* consistant à s'assurer que la solution retenue résout bien le problème.

L'utilisation de méthodes formelles consiste le plus souvent à :

- insérer une étape de spécification formelle entre l'expression des besoins et la conceptionΓ cette étape visant à formaliser les besoinsΓ
- utiliser cette spécification pour concevoir et/ou valider le système par des techniques formellesΓen s'appuyant sur des outils correspondants.

Une méthode formelle peut donc être caractérisée par :

- un langage ou des langages informatiquesΓayant une syntaxe et une sémantique formelles (il se peutΓen effetΓque des langages différents soient utilisés aux étapes de spécification et de conception)Γ
- des techniques formelles de conception ou de validation de systèmes décrits dans ces langagesΓ
- des outils de mise en œuvre de ces techniques.

Langages et modèles pour développer des systèmes critiques

Les besoins en langages spécialisés aux différentes phases de développement sont donc importants. Il existe un certain nombre de langages dédiés à des domaines d'applications spécifiques. Citons à titre d'exemples le langage VHDL pour le matériel les langages ESTELLE LOTOS et SDL pour la spécification des protocoles de communication les langages ARGOS ESTEREL ou LUSTRE pour la spécification de systèmes réactifs etc.

Plus fondamentalement ces langages se classent en deux catégories bien distinctes :

- Les langages déclaratifs fondés sur des logiques. Ces langages sont soit des extensions du calcul des prédicats comme B Z VDM soit des logiques spécifiques comme les logiques temporelles. Ils sont surtout utilisés en spécification mais peuvent accompagner tout le développement (c'est le cas de B ou de LUSTRE).
- les langages impératifs séquentiels ou parallèles. Ces langages permettent la description de comportements globaux d'un système en terme de compositions de sous-comportements. ARGOS ESTEREL les calculs de processus comme CCS CSP ou les réseaux de Pétri sont des exemples de tels langages.

Les logiciens ont depuis longtemps introduit les notions de sémantique et de modèles associés à des logiques. En informatique les premières sémantiques (dénotationnelles axiomatiques) se sont appliquées à des langages impératifs séquentiels. Ces sémantiques bien adaptées aux systèmes transformationnels se sont avérées peu adaptées aux systèmes réactifs et interactifs qui constituent une grande part des systèmes critiques [HP85].

En revanche le type de sémantique qui paraît à l'expérience bien adapté est la sémantique opérationnelle. La sémantique opérationnelle d'un langage permet d'associer un système de transitions étiquetées à un programme. Un système de transitions étiquetées est constitué d'un ensemble d'états et de transitions entre ces états étiquetées par des actions. Un état est caractérisé par les valeurs des différentes variables du programme et un point de contrôle. Les actions peuvent être des suites atomiques d'instructions (comme par exemple des affectations de valeur à des variables) ou des actions de communications. La sémantique d'un programme parallèle peut s'exprimer en termes d'automates communicants [AN82 Niv79 Arn92] ou de calculs de processus [Mil80 Mil89a]. Un calcul de processus est une algèbre de termes munie d'une sémantique opérationnelle qui permet d'interpréter des termes comme des systèmes de transitions étiquetées et munie d'une égalité sémantique. L'égalité sémantique peut être définie de deux manières. La première définition *syntaxique* ou *structurelle* est un ensemble de lois algébriques sur les termes (axiomes et règles d'inférence). La deuxième définition *sémantique* est une relation d'équivalence sur les systèmes de transitions étiquetées. Il se pose alors les problèmes de complétude et de consistance du système formel vis-à-vis de l'égalité sémantique [Mil89b BFG+91].

Les modèles étudiés ici sont donc les systèmes de transitions étiquetées. Des informations supplémentaires sur les états ou sur les transitions peuvent être utilisées en fonction du type d'analyse :

- pour les propriétés logiques on considère des systèmes de transitions étiquetées avec des prédicats sur les états

- pour les propriétés comportementalesΓon considère des systèmes de transitions étiquetées par des actions internes ou des actions de communicationΓ

- dans le cadre du testΓà cause de la nature asymétrique des communications entre le testeur et l'application à testerΓon considère des systèmes de transitions étiquetées à entrées et sortiesΓ

- lorsque l'on fait de l'analyse de flux de donnéesΓon considère des systèmes de transition dont les états sont des blocs de baseΓc'est-à-dire des séquences élémentaires d'affectations des variables du programme.

La conception

La conception s'appuyant sur des méthodes formelles peut être vue comme une suite de transformations successives de spécificationsΓpartant des spécifications initiales et aboutissant à des spécifications exécutablesΓvoire même compilables. C'est la base des méthodes formelles telles que BΓZ ou VDM.

La validation de programmes

Lorsque l'approche précédente par transformations successives n'est pas appliquéeΓil faut valider la conceptionΓc'est-à-dire comparer les spécifications et le système réalisé. Deux approches sont possibles : la vérification et le test.

La vérification. La vérification consiste à s'assurer du fonctionnement correct du programme par rapport aux spécifications vues comme un ensemble de propriétés. Il existe deux approches principales de la vérification :

- la premièreΓ déductiveΓ syntaxique ou structurelleΓ consiste à prouver une propriété en utilisant la structure du programme. La sémantique axiomatique d'un langage permet d'associer à un programme un ensemble d'axiomes et de règles d'inférence. Prouver une propriété revient à montrer que celle-ci est un théorème en utilisant les axiomes et les règles associés au programme. La mécanisation des preuves consiste à utiliser un démonstrateur de théorème. Cette approche est bien adaptée aux méthodes par transformations successives.

- la deuxième approcheΓappelée vérification par les modèles ou sémantiqueΓconsiste à prouver que l'ensemble des comportements du programme est un modèleΓau sens logique du termeΓpour la propriété. La sémantique opérationnelle d'un langage permet d'associer à un programme un système de transitions étiquetées. La mécanisation de la preuve est une procédure de décision.

On distingue généralement deux types de propriété :

- *Propriétés comportementales* : elles expriment un comportement attendu du système observé à un certain niveau d'abstraction. Une propriété comportementale peut être modélisée par un système de transitions étiquetées. La vérification consiste à comparer le système et la propriété à l'aide d'une relation d'équivalence ou de préordre.

- *Propriétés logiques* : elles décrivent une propriété globale du système comme l'absence d'interblocage l'exclusion mutuelle etc. Les logiques temporelles sont des formalismes bien adaptés pour exprimer ces propriétés. La vérification consiste à montrer que le programme satisfait la formule.

Le test. Historiquement le test a été la première et principale méthode de validation. Il reste une activité essentielle étant données les difficultés de conception et de vérification formelles. L'activité de test comporte essentiellement deux phases :

- La conception et la réalisation d'arbres de test. Un arbre de test décrit les interactions entre la réalisation à tester et le testeur. Pour cela le concepteur utilise la spécification et des *objectifs de test*. Ces derniers expriment une propriété particulière que l'on veut tester. Les arbres de test sont construits à partir de la spécification et des objectifs de test.

- L'exécution de l'arbre de test par un dispositif informatique le *testeur*.

Vérification et test Dans le cadre de la vérification par les modèles on compare deux descriptions formelles l'une pour le programme décrivant l'ensemble des comportements possibles l'autre pour la propriété. Lorsque la réalisation logicielle et matérielle du programme ne correspond pas à une étape de transformation de spécifications formelles elle est vue comme une boite noire avec des entrées que le testeur applique et des sorties qu'il observe. L'arbre de test modélisant ces interactions peut être calculé (par comparaison) à partir d'une spécification et d'un objectif de test. Cet objectif de test peut se formaliser comme une propriété (comportementale ou logique). Il est intéressant de remarquer que les mêmes algorithmes et modèles peuvent être utilisés aussi bien pour la vérification ou le test.

1.2 Les travaux présentés

L'axe principal de mes recherches est la vérification de spécifications formelles et la génération d'arbres de test. Mes travaux obéissent à une logique de développement d'outils appliqués essentiellement dans le domaine des protocoles de communication. La nature expérimentale de ces recherches conduit à amener les outils dans un état où le transfert industriel est possible. Parmi les conditions nécessaires pour y parvenir deux me semblent importantes : la facilité d'utilisation des outils et des performances raisonnables. L'efficacité des outils et donc des algorithmes est ma principale préoccupation. La limitation des outils est généralement due à la taille des modèles traités. Celle-ci est normalement mesurée en nombre d'états et en nombre de transitions. Cependant nous faisons référence dans ce document au nombre d'états ce qui est une approximation à un facteur multiplicatif près dans la mesure où le nombre de successeurs d'un état est de quelques unités dans les exemples que nous avons traités.

Ce mémoire est constitué de trois parties : modèlesΓalgorithmes et plate-forme pour la validation. La partie algorithmique est la plus détailléeΓcar la plus significative de mon travail.

Le chapitre 2 est dédié aux systèmes de transitions étiquetéesΓà la vérification par les modèles et à la génération d'arbres de test. Nous avons mis l'accent sur une application de la vérification à l'optimisation globale de code et sur la partie test :

- l'optimisation globale est une phase de la compilationΓqui est indépendante de la machine cible. Son objectif est de transformer une représentation du programmeΓsoit pour en minimiser la tailleΓsoit pour en améliorer le temps d'exécution. Lorsque les systèmes de transitions étiquetées sont utilisés comme modèles de programmeΓon peut envisager l'adaptation de certains algorithmes de vérification à l'optimisation. CitonsΓcomme exemples d'optimisationΓl'élimination des sous-expressions partiellement redondantesΓla suppression des branchements inutilesΓla propagation des constantesΓle déplacement de code pour optimiser les boucles. La suppression de branchements inutiles peut se formaliser comme un problème d'équivalence sur les systèmes de transitions étiquetées : un branchement est inutile si et seulement si les états atteints sont équivalents. Cela permet de remplacer la construction conditionnelle :
  if `exp` **then** `inst` else `inst'` par `inst` si `inst` et `inst'` sont des instructions équivalentes.

- la conception et la réalisation d'arbres de test sont baséesΓd'une part sur la spécification formelle du système à tester et sur un objectif de testΓetΓd'autre partΓsur une description de l'environnement d'exécution de cet arbre de test. A partir d'une spécification et d'un objectif de testΓdécrits par des systèmes de transitions étiquetéesΓon cherche à construire un arbre de test représentant tous les comportements de la spécification qui satisfont l'objectif de test. Cette relation de satisfactionΓappelée ici *relation de cohérence*Γmet en relation les états de la spécification et les états de l'objectif de test. Pour celaΓnous avons adapté un algorithme de vérification à la construction de cette relation. En utilisant cette relation et les informations sur l'environnement de testΓun arbre de test est produit.

Les systèmes de transitions étiquetées que nous considérons sont finisΓsauf mention contraire.

Le chapitre 3 est consacré aux algorithmes qui sont au cœur de mon travail. Plutôt que de les présenter chronologiquementΓj'ai opté pour une classification qui distingue les algorithmes en fonction de leur utilisation. Pendant ma thèseΓj'ai conçu et réalisé ALDÉBARANΓqui est dédié à la vérification comportementale de systèmes de transitions étiquetées. NotammentΓj'ai programmé une extension d'un algorithme de Paige & TarjanΓbasé sur la notion de *raffinement de partition* : étant donné une partition initiale d'un ensemble et une relation binaire définie sur cet ensembleΓl'algorithme de Paige & Tarjan [PT87] calcule la partition la plus grossière (celle qui identifie le plus d'états)Γincluse dans la partition initialeΓet compatible avec la relation binaire (l'image réciproque d'une classeΓpar la relation binaireΓest une union de classes) ; l'extension proposée consiste à considérer une famille de relations binaires à la place d'une seule relation binaire. Le retour d'expérience comporte deux aspects. PremièrementΓcet algorithme peut fonctionner indépendamment de la vérification. Il est utilisé pour minimiser (optimiser) les automates (ou systèmes de transitions étiquetées) produits par des compilateurs pour les langages LUSTRE ou LOTOS. DeuxièmementΓla taille des systèmes de transitions étiquetées que l'on peut traiter et qui sont représentés explicitement par la relation de transitions étiquetéesΓ

est de quelques centaines de milliers d'états. Deux directions de recherche sont développées pour faire reculer les limites des outils :

- *les algorithmes par analyse globale* [Fer90ΓBFH+92ΓFKM93ΓFer93] (bien adaptés pour la vérification symbolique)Γdont le principe estΓcomme dans Paige & Tarjan [PT87]Γbasé sur le raffinement de partition. Dans [BFH+92]Γon combine l'accessibilité des états et le raffinement de partition. Les instructions mises en jeu dans ces algorithmes sont les opérations ensemblistes sur les ensembles d'états et le calcul des prédécesseurs vis-à-vis de la relation de transitions étiquetées. Cela permet l'utilisation de techniques symboliques pour représenter les systèmes de transitions étiquetées. Cette approche a permis de traiter des systèmes de transitions étiquetées de quelques centaines de millions d'états.

- *les algorithmes par analyse locale* (pour la *vérification à la volée*) [FJJM92ΓFM95ΓCFG95Γ FJJV96b]Γdont le principe est de ne pas construire explicitement le système de transitions étiquetées. A partir d'une représentation de la spécification (programme LOTOSΓun ensemble de systèmes de transitions étiquetées communicantsΓ...) sous forme d'une fonction quiΓétant donné un étatΓcalcule l'ensemble de ses successeursΓon combine l'exploration du système de transitions étiquetées définie par cette fonctionΓet la validation. Cet approche a permis de traiter des systèmes de transitions étiquetées de l'ordre de quelques millions d'états. Son efficacité est surtout mise en valeur lorsque la propriété à vérifier est fausse ; dans ce casΓdes systèmes de taille supérieure peuvent être traités. Cet algorithme a été développé initialement pour la vérification comportementale [FM90]. Nous avons montré comment l'appliquer à la vérification logique [FJJM92ΓFM95]Γl'optimisation de code [CFG95] et la génération d'arbre de test [FJJV96b].

Le chapitre 4 contient une présentation de CADP (CAESAR-ALDÉBARAN DISTRIBUTION PACKAGE). CADP est dédié à la vérification de protocoles et comprend des compilateursΓdes vérificateurs et des utilitaires de gestion de modèlesΓpermettant un prototypage rapide d'algorithmes de vérification et une connexion avec d'autres environnements. Ceci nous a permis de développer (en collaboration avec le projet Pampa de l'Irisa) le générateur d'arbres de test TGV (Test Généré en utilisant les techniques de Vérification)Γen utilisant CADP.

Les objectifs de CADP sont d'une partΓl'intégration et l'expérimentation d'algorithmes performants pour la validationΓetΓd'autre partΓl'utilisation des différents modules pour spécifier et valider des systèmes critiques. Pour celaΓplusieurs formalismes peuvent être utilisés : le langage LOTOSΓles réseaux de PétriΓles systèmes de transitions étiquetées communicantsΓles systèmes de transitions étiquetées étendus (avec des variables) communicants.

Les systèmes de transitions étiquetées peuvent être représentés :

- *de manière explicite*Γpar la relation de transitions étiquetéesΓ

- *de manière implicite*Γ par un ensemble de fonctions permettant l'accès aux étatsΓ aux actions et aux transitionsΓ

- *de manière algébrique*Γpar une expression de composition de systèmes de transitions étiquetées.

Les algorithmes peuvent utiliser des techniques :

- *énumératives* (ce sont les états qui sont énumérés)Γdont les principales structures de données sont utilisées pour représenter des ensembles d'étatsΓ
- *symboliques*Γqui manipulent des prédicats ou des fonctions.

EnfinΓdans le chapitre 5Γnous présentons un bilan et proposons quelques perspectives de continuation de ces travaux.

Pour terminerΓj'aimerais présenter l'esprit dans lequel j'ai rédigé cette synthèse de mes travaux de recherche sur les algorithmes de validation et leur implantation. J'ai essayé de traduireΓ dans le chapitre 3Γla vision actuelle que j'ai des algorithmes sur lesquels j'ai travailléΓet de présenterΓdans le chapitre 4Γl'environnement dans lequel ils sont implantés. L'existence du chapitre 2 se justifieΓà mon sensΓpar les constatations expérimentales que nous avons faites sur le lien entre algorithmes et représentations du modèle. Plus précisémentΓnous avons remarqué que certaines représentations des systèmes de transitions étiquetées étaient mieux appropriées à certains types d'algorithmes. Je reviendrai sur ce point dans le chapitre 4. Par ailleursΓles aspects sur le testΓprésentés au chapitre 2Γet formalisés dans [FJJV96b]Γsont plus nouveauxΓ et sont donc susceptibles d'évoluer.

2 Modèles et Validation de programme

Ce chapitre vise à présenter les notions de base sur lesquelles se fonde notre travail. Comme nous l'avons dit au chapitre I, nous travaillons sur les systèmes de transitions étiquetées, considérés à la fois comme modèles de programmes ou de propriétés comportementales.

La section 2.1 contient la définition des systèmes de transitions étiquetées ainsi que trois représentations utilisées par les algorithmes présentés au chapitre suivant : la représentation *explicite*, *implicite* et *algébrique*.

La section 2.2 est un bref rappel des relations d'équivalence et de préordre, basées sur les notions de *bisimulation* et de *simulation*, et utilisées pour la comparaison ou la réduction des systèmes de transitions étiquetées. Nous avons appliqué ces notions dans deux cas particuliers, mis en exergue : l'optimisation globale et la génération d'arbres de test. Cette dernière partie est plus détaillée, car elle fait partie de mes préoccupations du moment.

Outre les systèmes de transitions, nous nous sommes intéressés à la vérification logique. Cela nous conduit à présenter, dans la section 2.3, le μ–calcul sous ses aspects syntaxique et sémantique.

2.1 Les modèles : les systèmes de transitions étiquetées

Un système de transitions étiquetées associé à programme représente soit un ensemble de traces d'exécution (*sémantique linéaire*), soit un ensemble d'arbres d'exécution (*sémantique arborescente*). Dans le premier cas, l'égalité sémantique est basée sur la notion d'égalité de langages. Dans le second cas, l'égalité sémantique est basée sur la notion de bisimulation.

Le modèle utilisé : Le modèle que nous utilisons est un système de transitions étiquetées $S = (Q^S, A, \{\xrightarrow{a}_S\}_{a \in A}, q^S_{\text{init}})$ où Q^S est l'ensemble des états, A un ensemble d'actions, $\xrightarrow{a}_S \subseteq Q^S \times Q^S$ la relation de transition étiquetée par des éléments de A, et q^S_{init} l'état initial. Dans la suite, on note $\xrightarrow{a}$ à la place de $\xrightarrow{a}_S$, s'il n'y a pas d'ambiguïté. Ce modèle peut être spécialisé suivant le type de validation :

- pour la vérification comportementale et logique, nous distinguons dans A une action particulière, notée τ, qui désigne toutes les actions internes,
- pour la génération d'arbres de test, l'ensemble A est partitionné entre les entrées, les sorties et les actions internes,
- pour la suppression des branchements, l'ensemble A est partitionné entre les actions visibles et les actions de choix.

Comme il est usuelΓun système de transitions étiquetées est identifié à son état initial. Pour celaΓ nous considérons un univers d'états $\mathcal{Q}$. L'ensemble des états Q^S de $S = (Q^S, A, \{\xrightarrow{a}\}_{a\in A}, q^S_{\text{init}})$, est le sous-ensemble des états accessibles de $\mathcal{Q}$ à partir de q^S_{init}Γ via la relation de transitions étiquetées.

On appelle *successeurs immédiats* (resp. *prédécesseurs immédiats*) d'un état p l'ensemble $\{q \mid \exists a.\ p \xrightarrow{a} q\}$ (resp. $\{q \mid \exists a.\ q \xrightarrow{a} p\}$). La transition $p \xrightarrow{a} q$ est une transition *entrante* de q et *sortante* de p.

Un système de transitions étiquetées est *déterministe* si pour tout état p et toute action aΓp a au plus une transition sortante étiquetée par a.

Notations : A^* désigne l'ensemble des mots finis sur AΓuv la concaténation du mot v après le mot uΓϵ le mot vide. Un langage λ est un sous-ensemble de A^*. Comme il est usuelΓ$a \in A$ désigne le langage ne comportant que le mot aΓsi λ_1, λ_2 sont deux langages alors $\lambda_1 + \lambda_2$Γ$\lambda_1\lambda_2$ et λ_1^* désignent respectivement l'unionΓla concaténation et l'itération des langages. On peut étendre aux motsΓpuis aux langagesΓla relation de transitions étiquetées dans un système de transitions étiquetées : pour $u, v \in A^*, \lambda \subseteq A^*$Γ

$$
\begin{array}{lll}
q & \xrightarrow{\epsilon} & q \\
p & \xrightarrow{uv} & q - \exists p' \cdot p \xrightarrow{u} p' \wedge p' \xrightarrow{v} q \\
p & \xrightarrow{\lambda} & q - \exists u \in \lambda \cdot p \xrightarrow{u} q
\end{array}
$$

On utilise la fonction prédécesseurΓpour une action a et un sous-ensemble d'états X :

$$pre_a(X) \quad = \quad \{p \mid \exists p' \,.\, p' \in X \text{ et } p \xrightarrow{a} p'\}$$

Par ailleursΓon note $\mathcal{A}\text{ct}(p) = \{\alpha \mid \exists p' \,.\, p \xrightarrow{\alpha} p'\}$ l'ensemble des actions possibles dans p.

Les équivalences : Les équivalences qui nous intéressent sont celles définies sur les états. Comparer deux systèmes de transitions étiquetées consiste à comparer les deux états initiaux en utilisant l'union des relations de transitions étiquetées. Une relation d'équivalence $\sim$ peut servir à minimiser un système de transitions étiquetées S. Pour celaΓon considère le système de transitions étiquetées *quotient*Γ$S/\sim$: l'ensemble des états du quotient est l'ensemble des classes d'équivalence de Q^SΓl'état initial du quotient est la classe contenant l'état initial de SΓet deux classes sont en relation $C_1 \xrightarrow{a} C_2$ si et seulement si il existe un état q_1 de C_1 et q_2 de C_2 vérifiant $q_1 \xrightarrow{a} q_2$. Bien entendu cette définition n'a d'intérêt que si la relation d'équivalence est *compatible* avec la relation de transitions étiquetées. Une relation d'équivalence $\sim$ est compatible avec la relation de transitions étiquetées si et seulement siΓpour toute action a et toutes classes C_1 et C_2 de $\sim$Γsoit $pre_a(C_1) \bigcap C_2 = \emptyset$Γsoit $C_2 \subseteq pre_a(C_1)$. Cette propriété de compatibilité caractérise les relations d'équivalence basées sur la notion de bisimulation.

Représentation des systèmes de transitions étiquetées Nous nous sommes intéressés à trois types de représentation de systèmes de transitions étiquetées ; ce choix correspond à ceux faits dans ALDÉBARAN :

- *la représentation explicite* est définie comme un couple comprenant l'état initial et la relation de transitions étiquetées définie en extension
- *la représentation implicite* est définie comme un couple comprenant l'état initial et la fonction qui étant donné un état calcule l'ensemble de ses successeurs
- *la représentation algébrique* est définie syntaxiquement de la manière suivante :

$$t ::= S \mid t||_G t \mid \textbf{hide } G \textbf{ in } t$$

où S est un système de transitions étiquetées représenté de manière explicite G est un sous-ensemble de l'ensemble des actions A $||_G$ est un opérateur de composition parallèle paramétré par un ensemble de synchronisations G et **hide** l'opérateur d'abstraction (qui renomme en τ toutes les actions de G). Le système de transitions étiquetées associé à ces constructions est défini par des règles de sémantique opérationnelle. Il n'est pas utile dans la suite de ce document de décrire une sémantique particulière pour l'opérateur $||_G$. On peut par exemple choisir une sémantique opérationnelle "à la LOTOS". Il est important de souligner qu'il s'agit d'une sémantique d'*entrelacement* voir figure 2.1.

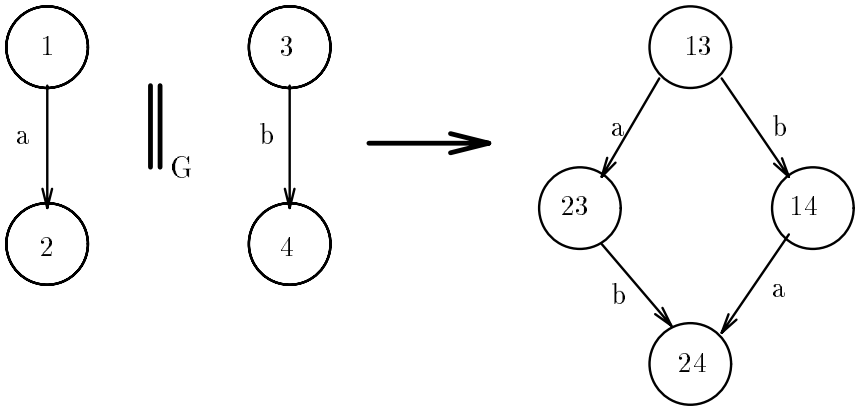

Figure 2.1: Exemple d'entrelacement : si les actions a et b ne font pas partie de G on entrelace les transitions étiquetées par a et b

2.2 Propriétés comportementales

2.2.1 La vérification comportementale arborescente

Cette partie présente les notions de simulation et de bisimulation qui ont conduit à l'étude des algorithmes et à la réalisation d'ALDÉBARAN [Fer88 Mou92 Ker94]. La vérification comportementale est basée sur la comparaison entre deux niveaux de description. Cette comparaison peut être formalisée soit par une relation d'équivalence soit par une relation de préordre. La première notion définit une égalité sémantique entre ensembles de comportements alors que la seconde définit une inclusion. D'un point de vue mathématique à toute relation de préordre R est associée de manière canonique la relation d'équivalence $R \cap R^{-1}$.

Dans le cadre de la sémantique arborescente, les relations de préordre (resp. d'équivalence) sont basées sur la notion de *simulation* (resp. de *bisimulation*) [Par81, Mil80], et sont paramétrées par un critère d'abstraction [Bou85]. Un critère d'abstraction sur A est une partition partielle de A^*, c'est-à-dire un ensemble de langages (ou *actions abstraites*) deux à deux disjoints. La définition de la relation de simulation stipule que deux états q_1 et q_2 appartiennent à la relation si et seulement si pour tout état q_1', atteignable à partir de q_1 par une transition étiquetée par a, il existe un état q_2', atteignable à partir de q_2 par une transition étiquetée par a, tel que q_1' et q_2' appartiennent à la relation. Pour la bisimulation, la réciproque est exigée. Cette notion peut être étendue en utilisant une relation de transitions étiquetées par des langages au lieu d'actions.

Diverses relations de préordre (resp. d'équivalence) ont été proposées. Elles présentent un certain nombre d'avantages :

- en faisant varier la notion d'observabilité des actions du programme, nous pouvons obtenir des équivalences plus ou moins fines. Le choix de la relation dépend de la propriété que l'on veut vérifier,
- il existe des algorithmes de décision efficaces pour diverses relations de bisimulation [PT87, KS90, GV90].

Les relations de simulation et de bisimulation

Une simulation (resp. bisimulation) sur un système de transitions étiquetées, paramétrée par un critère d'abstraction Λ sur A, est une relation binaire ρ sur les états vérifiant $\rho = \mathcal{G}(\rho)$ (resp. $\rho = \mathcal{H}(\rho)$), où :

$$(p,q) \in \mathcal{G}(\rho) \text{ ssi } \forall \lambda \in \Lambda \quad \forall p' \quad . (p \xrightarrow{\lambda} p' \Longrightarrow \exists q'.\, q \xrightarrow{\lambda} q' \ \wedge \ (p',q') \in \rho)$$

(resp.

$$\begin{aligned}(p,q) \in \mathcal{H}(\rho) \text{ ssi } \forall \lambda \in \Lambda \quad & \forall p' \quad . (p \xrightarrow{\lambda} p' \Longrightarrow \exists q'.\, q \xrightarrow{\lambda} q' \ \wedge \ (p',q') \in \rho) \wedge \\ & \forall q' \quad . (q \xrightarrow{\lambda} q' \Longrightarrow \exists p'.\, p \xrightarrow{\lambda} p' \ \wedge \ (p',q') \in \rho))\end{aligned}$$

Remarquons que la définition de la bisimulation se déduit de celle de la simulation en ajoutant "ρ symétrique". Par ailleurs, on calcule la plus grande relation ρ qui satisfait la définition ci-dessus et qui est incluse dans une relation initiale ρ_{init}. La plus grande relation de simulation (resp. de bisimulation), incluse dans une partition initiale ρ_{init}, définie par $\bigcap_{n \in \mathbb{N}} \mathcal{G}^n(\rho_{\text{init}})$ (resp. $\bigcap_{n \in \mathbb{N}} \mathcal{H}^n(\rho_{\text{init}})$) est une relation de préordre (resp. d'équivalence).

Exemples de critères d'abstraction:

- *simulation, bisimulation forte* : le critère d'abstraction est $\{a \mid a \in A\}$ où a désigne le langage formé du mot a.

- *bisimulation faible* ou *équivalence observationnelle* : cette équivalence proposée par Milner [Mil80] prend en compte les actions internes. Le critère d'abstraction est

  $\{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in A \text{ et } a \neq \tau\}$

- *préordre de sûreté, bisimulation* $\tau^* a$: Contrairement aux deux précédentes elle ne s'intéresse qu'aux actions visibles. Dans la définition de la bisimulation on ne considère que les actions $a \neq \tau$. Le critère d'abstraction est:

  $\{\tau^* a \tau^* \mid a \in A \text{ et } a \neq \tau\}$

Une *bisimulation de branchement* [GW89] est une relation binaire ρ symétrique telle que $\rho = \mathcal{B}(\rho)$ où

$$(p,q) \in \mathcal{B}(\rho) \text{ ssi} \quad \forall \alpha \in A \forall p' \,.\, p \xrightarrow{\alpha} p' \Longrightarrow \alpha = \tau \ \wedge \ (p',q) \in \rho \ \vee$$
$$\exists q_0, ..., q_n, q' .\, q = q_0, q_i \xrightarrow{\tau} q_{i+1}, q_n \xrightarrow{\alpha} q' \ \wedge \ (p,q_i) \in \rho \ \wedge \ (p',q') \in \rho$$

Cette bisimulation prend en compte les actions internes et préserve la structure de branchement des systèmes de transitions étiquetées considérés.

Ces relations sont ordonnées par la relation d'inclusion $\subseteq$. Remarquons que si $\rho_1 \subseteq \rho_2$ alors toute classe d'équivalence de ρ_2 est une union de classes d'équivalence de ρ_1. La bisimulation forte est incluse dans la bisimulation de branchement qui est incluse dans l'équivalence observationnelle ou dans la bisimulation $\tau^* a$. On trouvera dans [vG90 Mou92] une classification de certaines équivalences comportementales.

Méthodologie de vérification

Ce qui suit est le fruit d'expériences que nous avons menées en utilisant CADP. Les relations de comparaison présentées ci-dessus sont décidables pour les systèmes de transitions étiquetées finis. En pratique celles que nous utilisons le plus souvent sont le préordre de sûreté les bisimulations forte de branchement faible et $\tau^* a$. Cependant la validation peut être impossible à cause de la taille des systèmes de transitions étiquetées.

En fonction de la représentation des systèmes de transitions étiquetées plusieurs solutions peuvent être envisagées pour prouver $p_1 \sim_1 p_2$; en voici quelques exemples :

- lorsque les systèmes de transitions étiquetées sont représentés de manière explicite l'idée la plus simple qui vient à l'esprit est d'utiliser la hiérarchie entre les relations d'équivalence : prouver $p_1' \sim_2 p_2'$ avec $p_1 \sim_1 p_1'$ $p_2 \sim_1 p_2'$ et $\sim_2$ incluse dans $\sim_1$. Ce processus de remplacer une équivalence par une autre peut être réalisé en prenant en considération la taille des systèmes de transitions étiquetées ou le temps de comparaison (mis par un outil de vérification) de ces systèmes. Par exemple la minimisation de p_i en p_i' peut résulter de la minimisation par la bisimulation forte suivie de la minimisation de branchement. Ce processus d'enchaînement de minimisations peut être automatisé

- une autre idée consiste à tenir compte de la structure des systèmes de transitions étiquetées. Considérons par exemple une représentation algébrique. Supposons que p_1 soit de la

forme $q_1||_{G_2}\cdots||_{G_n}q_n$. On peut minimiser sous réserve que l'équivalence considérée soit une congruence chacun des composants (pour chaque q_i on obtient q_i') par rapport à une relation $\sim_2$ incluse dans $\sim_1$ et montrer que $q_1'||_{G_2}\cdots||_{G_n}q_n' \sim_1 p_2$. Par exemple $\sim_1$ peut être l'équivalence observationnelle et $\sim_2$ la bisimulation de branchement

- on peut aussi chercher à généraliser l'approche précédente en calculant un sous-terme de $q_1'||_{G_2}\cdots||_{G_n}q_n'$ (soit par exemple $q_1'||_{G_2}q_2'$) en minimisant le résultat (on obtient q_{12}") et en substituant celui-ci au sous-terme initial (q_{12}"$||_{G_3}\cdots||_{G_n}q_n'$) (calculer un sous-terme signifie lui associer un système de transitions étiquetées). On peut réitérer ce processus de composition-réductions jusqu'à l'élimination de l'opérateur de composition parallèle. Cependant il est bien connu que la taille d'un système de transitions étiquetées associé à un sous-terme peut être plus grande que la taille du système de transitions étiquetées associé au terme. Le traitement de ce problème a déjà été abordé et consiste à prendre en compte le "reste" du terme lorsque l'on calcule un sous-terme [GS90, Kri96].

Les définitions des notions de simulation et de bisimulation peuvent servir dans d'autres contexte que la vérification : l'optimisation globale et le test.

2.2.2 Application à l'optimisation de code : suppression des branchements inutiles

P. Caspi et A. Girault [CGP94, Gir94] ont proposé un algorithme de répartition de programmes synchrones représentés par des systèmes de transitions étiquetées. Un des problèmes rencontrés lors de la répartition de code par réplication (sur chaque site) est son optimisation. En effet cette réplication suivie d'une projection des actions sur leur site de calcul peut effacer des actions qui contribuaient à distinguer les états dans le système de transitions étiquetées initial. Cela conduit à des programmes non-minimaux vis-à-vis des branchements. Intuitivement on associe un état à chaque instruction du programme dans la phase de construction du système de transitions étiquetées. A l'instruction conditionnelle **if** `exp` **then** `inst1` else `inst0` on associe l'état $q : c_0q_0 + c_1q_1$ de la manière suivante : q_0 (resp. q_1) est associé à `inst0` (resp. `inst1`) c_0 (resp c_1) représente l'évaluation de `exp` à faux (resp. vrai) c_i étiquette la transition allant de q à q_i et $+$ représente le branchement. Un branchement est inutile si q_0 est équivalent à q_1.

A. Girault a formalisé cette équivalence [Gir94, CFG95] et l'a programmée dans le répartiteur de code `oc` (produit par les compilateurs ARGOS, ESTEREL, LUSTRE ou SIGNAL) appelé `oc2rep`. L'ensemble des actions du système de transitions étiquetées est partitionné en un ensemble d'actions visibles V et un ensemble de conditions C. Ce dernier est lui-même partitionné en C_0 et C_1. Intuitivement C_1 contient les branchements "vrai" et C_0 les branchements "faux". On peut caractériser syntaxiquement les systèmes de transitions étiquetées de la manière suivante pour $a \in V$ et $c_0, c_1 \in C$

$$q \ ::= \ nil \mid x \mid a.q \mid c_0.q + c_1.q \mid \text{rec } x.q$$

(où rec $x.q$ est la définition récursive de processus). Ces systèmes de transitions étiquetées sont déterministes et vérifient la propriété qu'un état exécute exclusivement une action visible ou un branchement.

A partir de cette caractérisation nous pouvons définir l'équivalence que nous avons appelée équivalence de test [CFG95].

Une relation binaire R sur Q^s est une équivalence de test si et seulement si

$$
\begin{array}{rl}
(p,q) \in R \Rightarrow & \forall \alpha, \forall p_0, p \xrightarrow{\alpha} p_0 \Rightarrow \\
& \left\{ \begin{array}{l} \alpha \in C \wedge (p_0, q) \in R \quad \vee \\ \exists q_0, q_1, q \xrightarrow{C^*} q_0 \xrightarrow{\alpha} q_1 \wedge (p, q_0) \in R \wedge (p_0, q_1) \in R \end{array} \right. \\
& \forall \alpha, \forall q_0, q \xrightarrow{\alpha} q_0 \Rightarrow \\
& \left\{ \begin{array}{l} \alpha \in C \wedge (p, q_0) \in R \quad \vee \\ \exists p_0, p_1, p \xrightarrow{C^*} p_0 \xrightarrow{\alpha} p_1 \wedge (p_0, q) \in R \wedge (p_1, q_0) \in R \} \end{array} \right.
\end{array}
$$

Remarque Nous nous sommes inspirés de la bisimulation de branchement pour définir cette équivalence de test. Elle en diffère dans la mesure où les conditions ne peuvent pas être renommées en τ : si $\tau.p_1 + \tau.p_0$ et $\tau.p_0 + \tau.p_1$ sont équivalents pour l'équivalence de branchement $c_1.p_1 + c_0.p_0$ et $c_0.p_1 + c_1.p_0$ ne sont généralement pas équivalents pour l'équivalence de test.

2.2.3 La génération automatique d'arbres de test

Plusieurs méthodes de test sont utilisées pour le développement des protocoles. Parmi celles-ci le test de conformité a pour objectif de démontrer l'adéquation d'une réalisation aux spécifications de référence. Dans la pratique la conformité est testée au moyen d'une suite de tests. Chaque test appelé ici arbre de test consiste en une procédure finie d'interactions entre le testeur et la réalisation à tester et doit aboutir à un verdict. La norme ISO 9646 [ISO92] dans le cas particulier du test de conformité propose trois sortes de verdict : *Pass* (réussi) *Fail* (raté) et *Inconclusive* (inconcluant). Il est possible de réessayer le test inconcluant pour tenter d'obtenir l'un des deux autres verdicts. Une réalisation est non conforme à la spécification s'il existe un arbre de test dont l'exécution conduit au verdict *Fail*.

Actuellement les tests de conformité sont produits manuellement. L'analyse de l'existant montre qu'environ 5 à 20 % des tests manuels sont erronés. L'étude appelée projet EGT (Etude pour la Génération de Tests) que nous avons menée durant une année pour le compte de la DGA (Direction Générale pour l'Armement) a montré que la génération automatique de tests apportait un gain en productivité et en qualité.

La génération de test reste un problème difficile d'un point de vue théorique et pratique. Les experts du test utilisent une méthodologie basée sur les étapes suivantes :

- identification d'une architecture de test qui est une description de l'environnement dans lequel la réalisation est testée
- écriture en langage naturel d'objectifs de test qui permettent de tester un aspect particulier de la réalisation
- dérivation manuelle des tests (sous forme TTCN norme ISO 9646).

Nous avonsΓen collaboration avec le projet Pampa de l'IrisaΓdéveloppé des méthodes et un outil appelé TGV. Le résultat du projet EGT sur un protocole réel a montré le bien-fondé de notre approche. Notre travail a consisté à étudier les concepts suivants :

- l'architecture de testΓ
- la spécificationΓ
- les objectifs de testΓ
- la notion de cohérence entre spécification et objectif de testΓ
- les arbres de testΓ
- les verdictsΓ
- les temporisations (timersΓen anglais)Γ
- la conformité d'une réalisation à une spécification.

Les modèles utilisés

Les modèles utilisés pour la spécificationΓl'objectif de test et l'arbre de test sont des *systèmes de transitions étiquetées à entrées et sorties*. En effetΓla distinction entre les entrées et les sorties nous paraît fondamentale : les interactions entre le testeur et la réalisation forment un processus où le testeur émet des messages et observeΓen retourΓles réactions de la réalisation.

Un système de transitions étiquetées à entrées et sorties est un système de transitions étiquetées $X = (Q^{\mathrm{X}}, A, \{\xrightarrow{a}_{\mathrm{X}}\}_{a \in A}, q^{\mathrm{X}}_{\mathrm{init}})$Γdont l'ensemble des actions A est partitionné entre entréesΓsorties et actions internes.

Contrôlabilité et observabilité Nous supposeronsΓ pour simplifierΓ que la réalisation à tester communique directement avec l'environnement de testΓque l'on appelle *testeur*. La communication est asymétrique : si on contrôle les sorties du testeurΓon ne peut qu'observerΓen retourΓ les sorties de la réalisation. Les entrées de la réalisation correspondent aux sorties du testeur et sont appelées *actions contrôlables*. InversementΓles sorties de la réalisation correspondent aux entrées du testeur et sont appelées *actions observables*. Pour observer la présence ou l'absence d'interblocage ou de bouclesΓon associe à chaque action observableΓune temporisation qui est gérée par le testeur.

Un système de transitions étiquetées à entrées et sorties satisfait la condition de *contrôlabilité* si pour tout état q tel qu'il existe une transition $q \xrightarrow{o} q'$ étiquetée par l'action contrôlable oΓalors l'état q n'a pas d'autre transition sortante. FormellementΓ$\forall a, q''. q \xrightarrow{a} q'' \Rightarrow a = o \wedge q'' = q'$.

Autrement ditΓun système de transitions étiquetées à entrées et sorties qui satisfait la condition de contrôlabilité a son ensemble d'états partitionnés de fait : les états qui ont une transition sortanteΓétiquetée par une action contrôlableΓet les états dont les transitions sortantes sont étiquetées par des actions observables.

Architecture de test

Elle comporte d'une part les dispositifs utilisés pour tester la réalisation et d'autre part l'interface de test qui permet l'accès à la réalisation sous test. La notion d'interface de test correspond à celle de *Points de Contrôle et d'Observation* (PCOs) que l'on trouve dans la norme ISO 9646. L'ISO propose dans cette norme quatre méthodes pour développer des architectures de test de conformité : une locale et trois externes. Nous nous sommes restreints au test local avec un seul testeur qui interagit avec la réalisation IUT (*Implementation Under Test*) via des PCOs figure 2.2.

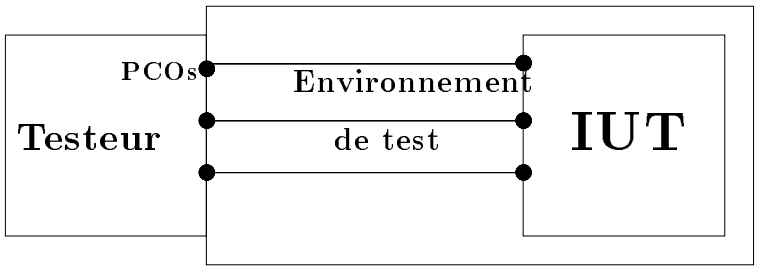

Figure 2.2: Environnement de test

L'architecture de test influence la génération de test et peut conduire à l'entrelacement de séquences d'actions observables observées sur des PCOs distincts. Nous avons pris en compte l'architecture de test en transformant la spécification. Cet aspect est détaillé dans [FJJV96a] section 4.5.

Spécification La spécification S est modélisée par un système de transitions étiquetées à entrées et sorties $(Q^{\mathrm{S}}, A, \{\xrightarrow{a}_{\mathrm{S}}\}_{a \in A}, q^{\mathrm{S}}_{\mathrm{init}})$.

Objectifs de test Un objectif de test est une propriété particulière que doit satisfaire la réalisation. Il peut comprendre un comportement qui spécifie un ordonnancement des interactions et des contraintes sur l'état de l'application (par exemple : "le réseau est saturé").

Nous avons choisi de modéliser les objectifs de test par des systèmes de transitions étiquetées à entrées et sorties déterministes acycliques et qui satisfont la condition de contrôlabilité. Ils ont un ensemble d'états accepteurs noté `Accept`. Un état accepteur n'a pas de successeur.

Dans ce qui suit nous notons $OT = (Q^{\mathrm{OT}}, A, \{\xrightarrow{a}_{\mathrm{OT}}\}_{a \in A}, q^{\mathrm{OT}}_{\mathrm{init}})$ le système de transitions étiquetées à entrées et sorties associé à l'objectif de test. L'objectif de test est défini par le couple formé de OT et de `Accept` où `Accept` $\subseteq Q^{\mathrm{OT}}$.

Cohérence entre l'objectif de test et la spécification Un objectif de test est un critère pour sélectionner un comportement de la spécification en vue de générer un arbre de test. Il faut donc définir une relation de cohérence entre l'objectif de test et la spécification. Celle-ci est définie de telle sorte que :

- l'ensemble des comportementsΓdéfini par l'objectif de testΓdoit être inclus d'une certaine manière dans l'ensemble des comportements de la spécificationΓ
- de tout état de la spécification en relation avec un état d'acceptationΓil existe une séquence conduisant à l'état initial de la spécification.

Nous nous sommes inspirés de la définition de la simulation pour définir la relation de cohérence. FormellementΓ soit Q^{OT} l'ensemble des états de l'objectif de testΓ Q^{S} l'ensemble des états de la spécification. Une relation $R \subseteq Q^{\mathrm{OT}} \times Q^{\mathrm{S}}$ est une relation de cohérence si et seulement si $R \subseteq \mathcal{F}(R)$ où :

$$
\begin{array}{lll}
\mathcal{F}(R) = \{(p^{\mathrm{OT}}, p^{\mathrm{S}}) \mid & & \\
p^{\mathrm{OT}} \xrightarrow{\alpha}_{\mathrm{OT}} q^{\mathrm{OT}} & \Longrightarrow & \exists q^{\mathrm{S}}, q_1^{\mathrm{S}}, \exists \sigma \in (A \setminus \{\alpha\})^* \cdot p^{\mathrm{S}} \xrightarrow{\sigma}_{\mathrm{S}} q_1^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \wedge \\
 & & (q^{\mathrm{OT}}, q^{\mathrm{S}}) \in R \wedge (p^{\mathrm{OT}}, q_1^{\mathrm{S}}) \in R \wedge \\
p^{\mathrm{OT}} \in \texttt{Accept} & \Longrightarrow & \exists \sigma \in A^* \cdot p^{\mathrm{S}} \xrightarrow{\sigma}_{\mathrm{S}} q_{\mathrm{init}}^{\mathrm{S}}\}
\end{array}
$$

L'objectif de test est cohérent vis-à-vis de la spécification si et seulement si il existe une relation $R \subseteq \mathcal{F}(R)$Γvérifiant $(q_{\mathrm{init}}^{\mathrm{OT}}, q_{\mathrm{init}}^{\mathrm{S}}) \in R$. Si l'objectif de test est cohérent vis-à-vis de la spécificationΓ alors il est possibleΓà partir de la spécification et de l'objectif de testΓde construire un arbre de test *valide* dans le sens où un verdict **Fail** correspond à une réalisation non conforme.

Arbre de Test Un arbre de test est un système de transitions étiquetées à entrées et sortiesΓ déterministeΓacyclique et qui satisfait la condition de contrôlabilité. Il est construit à partir de la spécification et de l'objectif de testΓsi la cohérence est vérifiée. Cette construction est réalisée par un algorithme linéaireΓ qui parcourt en profondeur d'abord un produit synchrone entre l'objectif de test et la spécification. L'idée de base de l'algorithme est de vérifier la cohérence pendant le parcours et de synthétiser l'arbre de test. Dans [FJJV96b]Γnous avons formellement défini par des règles :

- la construction du produit synchroneΓ
- la vérification de la condition de contrôlabilité et la synthèse de l'arbre de testΓà partir du produit synchroneΓ
- et la décoration de certaines transitions par des verdicts et des temporisations.

L'arbre de test est construit de la manière suivante :

Produit synchrone. On considère le système de transitions étiquetées à entrées et sortiesΓ $P = (Q^{\mathrm{P}}, A, \{\xrightarrow{a}_{\mathrm{P}}\}_{a \in A}, (q_{\mathrm{init}}^{\mathrm{OT}}, q_{\mathrm{init}}^{\mathrm{S}}))$Γproduit synchrone de OT et de SΓoù $Q^{\mathrm{P}} \subseteq Q^{\mathrm{OT}} \times Q^{\mathrm{S}}$ est l'ensemble des états accessibles via la relation de transition du produit. Un état du produit est un couple $(p^{\mathrm{OT}}, p^{\mathrm{S}})$Γcomposé d'un état p^{OT} de l'objectif de test et d'un état p^{S} de la spécification. Une transition $(p^{\mathrm{OT}}, p^{\mathrm{S}}) \xrightarrow{a}_{\mathrm{P}} (q^{\mathrm{OT}}, q^{\mathrm{S}})$ est dans le produit si et seulement si :

- le couple $(p^{\mathrm{OT}}, p^{\mathrm{S}})$ appartient à Q^{P}Γ
- $p^{\mathrm{S}} \xrightarrow{a}_{\mathrm{S}} q^{\mathrm{S}}$ et $p^{\mathrm{OT}} \xrightarrow{a}_{\mathrm{OT}} q^{\mathrm{OT}}$ ouΓ

- $p^{\mathrm{S}} \xrightarrow{a}_{\mathrm{S}} q^{\mathrm{S}}$ et $p^{\mathrm{OT}} = q^{\mathrm{OT}}\Gamma$
  si l'action a n'est pas possible dans $p^{\mathrm{OT}}\Gamma$et si $(p^{\mathrm{OT}}, p^{\mathrm{S}}) \notin \mathbf{Accept} \times \{q^{\mathrm{S}}_{\mathrm{init}}\}$.

Synthèse de l'arbre de test. On restreint les comportements du produit synchrone en synthétisantΓà partir des feuilles $(q^{\mathrm{OT}}, q^{\mathrm{S}}_{\mathrm{init}})$ (avec $q^{\mathrm{OT}} \in$ `Accept`) un système de transitions étiquetées à entrées et sortiesΓdéterministeΓacyclique et en imposant la condition de contrôlabilité.

Plus précisémentΓun arbre de test est un système de transitions étiquetées à entrées et sorties $T = ((Q^{\mathrm{T}}, A, \{\xrightarrow{a}_{\mathrm{T}}\}_{a \in A}, (q^{\mathrm{OT}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}}))\Gamma$où :

- $\perp$ est un état particulier n'appartenant pas à Q^{P} et servant par exemple à contruire des transitions étiquetées par **Inconclusive** ou **Fail**Γ
- $Q^{\mathrm{T}} \subseteq Q^{\mathrm{P}} \cup \{\perp\}\Gamma$
- les feuilles $(p^{\mathrm{OT}}, q^{\mathrm{S}}_{\mathrm{init}}) \in Q^{\mathrm{T}}\Gamma$où p^{OT} appartient à l'ensemble des états accepteursΓ
- à partir de tout état de l'arbre de testΓil existe une séquence d'exécution conduisant à une feuilleΓ
- chaque état de l'arbre de test vérifie la condition de contrôlabilité.

Dans l'arbre de testΓnous distinguons trois parties : le *préambule*Γle *corps* et le *postambule*.

postambule Un état $(p^{\mathrm{OT}}, p^{\mathrm{S}})$ appartient à un postambule si et seulement si p^{OT} est un état accepteurΓ

préambule Un état appartient au préambule si et seulement si il est de la forme $(q^{\mathrm{OT}}_{\mathrm{init}}, p^{\mathrm{S}})\Gamma$

corps l'arbre de test Les autres états appartiennent au corps.

Verdicts. La signification des verdicts est la suivante :

Pass C'est un verdict qui signifie que l'état initial a été atteint par l'intermédiaire d'un postambuleΓaprès un verdict **(Pass)**. La dernière transition du postambule est étiquetée par un verdict `Pass`.

(Pass) Ce verdict signifie qu'une séquence de la spécification a atteint un état accepté par l'objectif de test. Après cet étatΓil faut trouver une séquence conduisant à l'état initial (postambule). Ce verdict étiquette une transition reliant le corps de l'arbre de test au postambule.

Inconclusive Ce verdict signifie qu'une entrée du testeurΓcorrespondant à une sortie de la spécificationΓn'est pas décrite dans l'arbre de test ou ne conduit pas à un verdict **(Pass)**. On rajouteΓdans l'arbre de testΓune nouvelle transitionΓentrante de $\perp\Gamma$ étiquetée par l'entrée correspondanteΓet on lui associe le verdict **Inconclusive**.

Fail Toute transition étiquetée par une entrée du testeurΓqui ne correspond pas à une sortie de la spécification. Cette transition est virtuelle et correspond dans la pratique à "otherwise Fail" en TTCN.

Temporisations Les temporisations doivent assurer la finitude de l'application d'un arbre de test sur une réalisation. Tout blocage ou boucle divergente (τ^{ω}) doit être détecté : lorsque le testeur émet une sortieΓil attend en retour un certain nombre d'entrées. A l'inverseΓsi

les délais des temporisations sont suffisantsΓchaque expiration d'une temporisation doit correspondre à un blocage ou à une boucle divergente. A chaque entrée i est associé une temporisation t_i. Trois opérations sont possibles sur les temporisations : **Start**(t_i)Γ **Cancel**(t_i) et **Timeout**(t_i). Les opérations sur les temporisations décorent certaines transitions étiquetées.

Start (t_i) est réalisé dans la transition qui atteint un état où i est possibleΓ

Cancel (t_i) est réalisé à la réception de i ou quand celle-ci n'est plus possible (à cause d'un choix par exemple)Γ

Timeout (t_i) représente l'observation d'un blocage sur l'attente de i. Une transitionΓ entrante de $\perp$Γétiquetée par **Timeout**(t_i) est rajoutée en alternative à l'état où une transition étiquetée par i est possible.

Conformité d'une réalisation à une spécification

La réalisation du système n'est connue que par l'intermédiaire de ses interactions avec l'environnement. Pour définir formellement la notion de conformitéΓl'enchaînement des interactions entre la réalisation et l'environnement (qui peut être le testeur) est modélisé par un système de transitions étiquetées à entrées et sortiesΓnoté $R = (Q^{\mathrm{R}}, A', \{\xrightarrow{a}_{\mathrm{R}}\}_{a \in A'}, q^{\mathrm{R}}_{\mathrm{init}})$Γoù $A \subseteq A'$. Le testeur a donc une *vue externe* de la réalisation. Par oppositionΓla spécification modélise une *vue interne* sans hypothèse sur la manière dont elle interagit avec son environnement.

La relation de conformité choisie est semblable à celle définie dans [Tre95ΓPha94]. Elle stipule queΓpour toute séquence d'exécution σ de la spécificationΓtelle que $q^{\mathrm{S}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathrm{S}} q^{\mathrm{S}}$Γs'il existe un état q^{R} vérifiant $q^{\mathrm{R}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathrm{R}} q^{\mathrm{R}}$Γalors l'ensemble des actions observables de q^{R} est inclus dans l'ensemble des actions observables de q^{S}.

Lors de l'application d'un arbre de test sur une réalisationΓle testeur n'émet et ne reçoit que les interactions décrites par l'arbre de test. Il est impossibleΓdu point de vue du testeurΓde revenir en arrière et donc d'observer un branchement non-déterministe de la réalisation. Ce qui fait que deux implantations ayant les même traces ne seront pas distinguables. A contrarioΓune réalisation n'est pas conforme à une spécificationΓsi un comportement de la réalisation n'est pas dans la spécification. Cette non-conformité peut-être détectée en appliquant un arbre de test.

Discussion

Je voudrais souligner dans ce paragrapheΓles points qui sont susceptibles d'évoluer. En effetΓla manière dont nous avons formalisé les différents conceptsΓa été influencée par les nombreuses discussions que nous avons eues avec les spécialistes du test des protocolesΓet par les contraintes techniques (et de temps) du projet Egt.

La notion de cohérence ne prend pas en compte la condition de contrôlabilité. A partir d'une spécification et d'un objectif de testΓil est possible de produire plusieurs arbres de test. La méthode implantée dans Tgv ne génère qu'un seul arbre de test.

Nous avons modélisé l'architecture de test en déterminantΓpour chaque action observableΓle PCO sur lequel elle est observée. Cela revient à partitionner l'ensemble des actions observablesΓ

de telle sorte que chaque classe est identifiée à un PCO. L'architecture de test a été prise en compte par transformation de la spécification en entrelaçant les actions observables qui sont observées sur des PCOs distincts. Nous étudions la prise en compte de l'architecture de test après réduction de l'ensemble des comportements de la spécification par l'objectif de test sur le produit synchrone.

Nous avons modélisé les objectifs de test par des systèmes de transitions étiquetées à entrées et sorties déterministes acycliques et satisfaisant la condition de contrôlabilité. Ce choix repose sur le fait que dans le cadre de l'expérience que nous avons conduite les objectifs de test étaient relativement simples.

2.3 La vérification logique

Le principe est de modéliser les propriétés par des formules de logiques temporelles et de les évaluer sur un système de transitions étiquetées [Pnu85, MP82, QS83, Sif82, CES83]. Nous distinguons la logique linéaire qui exprime des propriétés sur les séquences d'exécution de la logique arborescente qui exprime des propriétés sur les états. On trouvera dans [Arn92] une classification de différentes logiques.

Notre contribution est marginale : elle est motivée par le fait que les algorithmes par analyse locale s'adaptaient au cadre de la vérification logique et par le fait qu'il n'y avait aucun vérificateur logique dans CADP. Nous avons programmé dans CADP EVALUATOR un évaluateur de formules du μ-calcul arborescent.

2.3.1 Le μ-calcul arborescent

Nous considérons le μ-calcul arborescent (propositionnel) qui est un sur-ensemble du calcul propositionnel. On rajoute au calcul propositionnel un nombre fixé d'opérateurs qui permettent d'exprimer l'évolution du système de transitions étiquetées.

La sémantique d'une formule du μ-calcul arborescent est un ensemble d'états. Les opérateurs temporels s'interprètent à l'aide :

- de combinaisons booléennes des opérateurs suivants sur les systèmes de transitions étiquetées

$$\begin{aligned} pre_a(X) &= \{p \mid \exists p'.p \xrightarrow{a} p' \wedge p' \in X\} \\ \widetilde{pre}_a(X) &= \{p \mid \forall p'.p \xrightarrow{a} p' \Rightarrow p' \in X\} \end{aligned}$$

- d'opérateurs de plus petit et de plus grand point fixe.

Au cours de notre travail nous nous sommes intéressés au μ-calcul propositionnel [Koz83] dont nous rappelons la syntaxe et la sémantique.

Syntaxe

$$\varphi ::= \text{true} \mid \text{false} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X \mid \neg\varphi \mid < a > \varphi \mid [a]\varphi \mid \nu X.\varphi \mid \mu X.\varphi \mid$$

Sémantique

Une formule est interprétée sur un système de transitions étiquetéesΓ
$S = (Q^{\mathrm{S}}, A, \{\xrightarrow{a}\}_{a \in A}, q^{\mathrm{S}}_{\mathrm{init}})$Γ par l'intermédiaire d'une fonction ρ qui associe un sous-ensemble d'états de Q^{S} à chaque variable de la formule. La sémantique d'une formule φΓ notée $[\![\varphi]\!]_\rho$ représente l'ensemble des états satisfaisant φ.

$$
\begin{array}{lcll}
[\![\mathrm{true}]\!]_\rho & = & Q^{\mathrm{S}} & \\
[\![\mathrm{false}]\!]_\rho & = & \emptyset & \\
[\![\varphi_1 \wedge \varphi_2]\!]_\rho & = & [\![\varphi_1]\!]_\rho \cap [\![\varphi_2]\!]_\rho & \\
[\![\varphi_1 \vee \varphi_2]\!]_\rho & = & [\![\varphi_1]\!]_\rho \cup [\![\varphi_2]\!]_\rho & \\
[\![X]\!]_\rho & = & \rho(X) & \\
[\![\neg\varphi]\!]_\rho & = & Q^{\mathrm{S}} \setminus [\![\varphi]\!]_\rho & \\
[\![< a > \varphi]\!]_\rho & = & \{q \mid \exists q' \,.\, q \xrightarrow{a} q' \wedge q' \in [\![\varphi]\!]_\rho\} & = \quad pre_a([\![\varphi]\!]_\rho) \\
[\![[a]\varphi]\!]_\rho & = & \{q \mid \forall q' \,.\, q \xrightarrow{a} q' \Rightarrow q' \in [\![\varphi]\!]_\rho\} & = \quad \widetilde{pre}_a\,([\![\varphi]\!]_\rho) \\
[\![\mu X.\varphi]\!]_\rho & = & \bigcap \{R \mid [\![\varphi]\!]_{\rho[R/X]} \subseteq R\} & \\
[\![\nu X.\varphi]\!]_\rho & = & \bigcup \{R \mid R \subseteq [\![\varphi]\!]_{\rho[R/X]}\} &
\end{array}
$$

2.4 A propos de modèles...

Les systèmes de transitions étiquetées sont largement utilisés pour modéliser les systèmes critiques. Il existe d'autres modèles : les réseaux de PétriΓles algèbres de processus communicantsΓ les structures d'événementsΓles automates étendusΓetc. Il existe aussi d'autres modes de communication commeΓpar exempleΓla communication par file. On peut définirΓpour chacun de ces formalismesΓune sémantique opérationnelle qui associe un système de transitions étiquetées à un modèle. L'inconvénient de cette traduction est que l'on peut être confronté au problème de la taille du système de transitions étiquetées : l'ensemble des états est infini ou trop grand pour être analysé par un outil de vérification. L'avantage de cette traduction est que les algorithmes de validation que nous utilisons (sur les systèmes de transitions étiquetées) sont efficaces en temps (complexité au plus cubique) et en mémoire (complexité linéaire).

Différents problèmes de validation s'expriment en terme de calcul de point fixe sur l'ensemble des états. Au chapitre suivantΓnous décrivons un ensemble d'algorithmes qui mettent en œuvre ces calculs.

3 Algorithmes

3.1 Introduction

Cette partie est consacrée aux algorithmes de validation sur les systèmes de transitions étiquetées. Dans le cas finiΓnous notons n le cardinal de l'ensemble des états et m celui de la relation de transitions étiquetées.

Ces algorithmes doivent déciderΓétant donné une spécification $S = (Q^S, A, \{\xrightarrow{a}\}_{a\in A}, q^S_{\text{init}})$Γune propriété Φ_{init} et une relation de comparaison $\mathcal{R}$Γsi la spécification satisfait la propriétéΓc'est-à-dire quelle est la valeur booléenne de $(q^S_{\text{init}}, \Phi_{\text{init}}) \in \mathcal{R}$. $\mathcal{R}$ est un point fixe d'un opérateur monotone f (cf. chapitre précédent). Comme nous l'avons indiqué en introductionΓ ces algorithmes peuvent être classés en deux catégories : les algorithmes par analyse globale et les algorithmes par analyse locale.

Les algorithmes par analyse globale C'est dans cette catégorie que se trouvent les algorithmes ayant la meilleure complexité en temps et en mémoire. Nous nous sommes intéressés uniquement à la vérification comportementale. Dans ce casΓ Φ_{init} est un système de transitions étiquetées. Le principe de ces algorithmes est décrit ci-dessous. Pour montrer que q^S_{init} et Φ_{init} sont équivalentsΓon calcule la plus grande relation d'équivalence sur l'union des ensembles d'états des deux systèmes de transitions étiquetéesΓpuis on vérifie si q^S_{init} et Φ_{init} sont équivalents. Plusieurs algorithmes par analyse globale sont possibles. Parmi ceux-ciΓ nous avons étudié les algorithmes basés sur le *raffinement de partition* : étant donné une partition initialeΓ raffiner la partition courante jusqu'à ce qu'elle soit compatible avec la relation de transitions étiquetées.

Nous avons montré que ces algorithmes pouvaient se combiner avec la représentation symbolique des ensembles d'états [FKM93]. Cette représentation est appelée ainsi parce que les ensembles d'états sont caractérisés par des prédicats ou des fonctions booléennes. Ces ensembles sont manipulés en compréhension. Parmi les techniques de représentation symboliqueΓ nous utilisons les *Diagrammes Binaires de Décision* (BDDs) [Bry86]. Ce formalisme procure une forme canoniqueΓ à l'ordre des variables prèsΓ pour les formules de logique propositionnelle et permet ainsi de représenter et de manipuler les ensembles par l'intermédiaire de leurs fonctions caractéristiques. Les BDDs ont été utilisés pour la vérification de circuits séquentielsΓ représentés par des automates à entrés et sorties [BCM+89Γ CBM89]Γ et pour la vérification de programmes synchronesΓ voir par exemple [RHR91]. Ce formalisme a été ensuite utilisé dans la mise en œuvre de la vérification comportementaleΓ voirΓpar exemple [EFT91ΓBdS92ΓFKM93].

Les algorithmes par analyse locale Ils ne nécessitent pas la construction préalable du système de transitions étiquetées. L'exploration de ce dernier peut se combiner avec la vérification de la propriété ou la construction d'un arbre de test. Le principe de ces algorithmes est le suivant : à partir du but initial $(q^S_{\text{init}}, \Phi_{\text{init}}) \in \mathcal{R}$Γon décompose un but $(q, \Phi) \in \mathcal{R}$ en

sous-buts $(q', \Phi') \in \mathcal{R}$ où q' est un successeur de q et Φ' est un successeur ou une sous-formule de Φ suivant le langage d'expression de la propriété. Ces algorithmes explorent un graphe composé dont la racine est $(q^{\mathrm{S}}_{\mathrm{init}}, \Phi_{\mathrm{init}})$ et synthétisent la valeur $(q^{\mathrm{S}}_{\mathrm{init}}, \Phi_{\mathrm{init}}) \in \mathcal{R}$ à la racine. La valeur associée à chaque état (q, Φ) est une combinaison des valeurs associées à ses successeurs (q', Φ'). C'est une valeur booléenne pour la vérification comportementale, la vérification logique, et la vérification de la cohérence d'un objectif de test vis-à-vis d'une spécification. Dans le cas de la synthèse d'un arbre de test, cette valeur représente un arbre.

Ces algorithmes peuvent être efficaces lorsque le parcours est partiel, ce qui est le cas lorsque la propriété à vérifier est fausse. Nous avons montré que l'algorithme à la volée pour les équivalences comportementales [FM90] peut s'adapter à :

- la vérification logique [FJJM92, FM95].
- la suppression des branchements inutiles [CFG95],
- la génération d'arbres de test [FJJV96b, FJJV96a].

Ce chapitre est organisé de la manière suivante. Dans la première partie, section 3.2, nous rappelons le principe de base des algorithmes basés sur le raffinement de partition, pour le calcul de plus grandes bisimulations (sous-section 3.2.1). Puis, nous présentons un algorithme qui combine le raffinement de partition et le calcul de l'accessibilité d'une classe (sous-section 3.2.2). Une classe est accessible si elle contient un état accessible. Dans le cas où les ensembles d'états sont représentés en compréhension, ce calcul d'accessibilité n'est pas un problème trivial. Un des problèmes de cet algorithme est la terminaison lorsque l'espace des états est infini. Nous proposons une heuristique permettant d'assurer, dans certains cas, la terminaison de cet algorithme (sous-section 3.2.3). Dans la deuxième partie, section 3.3, nous présentons un algorithme local, dont le principe est le parcours en profondeur d'abord d'un graphe composé (appelé aussi produit synchrone, dans le cas de la bisimulation et du test), définie sur le produit cartésien des états et des propriétés. Nous montrons comment nous avons adapté cet algorithme au calcul de bisimulation, à la génération d'arbres de test, à la détection des branchements inutiles, et à la vérification de formules du μ-calcul.

3.2 Algorithmes par analyse globale

3.2.1 Algorithmes de raffinement de partition

La comparaison et la réduction de systèmes de transitions étiquetées pour une relation de bisimulation sont basées sur un même principe. Il s'agit de raffiner une partition initiale de l'ensemble des états par rapport à la relation de transitions étiquetées. Une relation d'équivalence définie sur un ensemble peut être caractérisée soit comme une partition de cet ensemble, soit comme un ensemble de couples appartenant à cette relation. Suivant la définition choisie, on peut déduire deux sortes d'algorithmes : ceux qui raffinent une partition jusqu'à la rendre compatible avec la relation de transitions étiquetées, et ceux qui construisent l'ensemble des couples.

Remarquons que, à partir de la définition de la bisimulation, nous pouvons déduire une propriété caractéristique : si C et C' sont deux classes de la relation de bisimulation, alors, pour

tout a, soit $pre_a(C') \bigcap C = \emptyset$, soit $C \subseteq pre_a(C')$. Cette propriété est connue indifféremment sous le nom de *compatibilité* d'une relation d'équivalence avec la relation de transitions étiquetées, ou de *stabilité* d'une partition vis-à-vis de la relation de transitions étiquetées. Notons Raffiner(ρ, a, C, C') la fonction qui remplace dans la partition ρ la classe C par ses deux sous-classes $C \bigcap pre_a(C')$ et $C \setminus pre_a(C')$. A partir de cette définition, il vient un premier algorithme consistant à raffiner toutes les classes non stables par rapport à la partition courante :

– Soit ρ_{init} *la partition initiale*
begin
  $\rho := \rho_{\text{init}}$
  while $\exists a, C, C'.(C \bigcap pre_a(C') \neq \emptyset \wedge C \not\subseteq pre_a(C'))$ **loop**
    $\rho :=$ *Raffiner*(ρ, a, C, C')
  end loop
end

Cet algorithme permet de comparer deux systèmes de transitions étiquetées. Pour cela, on considère le système de transitions étiquetées obtenu en faisant l'union des états et l'union des relations de transitions étiquetées. Lorsque la partition est stable par rapport à la relation de transitions étiquetées, les deux états initiaux sont dans la même classe si et seulement si les systèmes sont équivalents.

Cet algorithme permet de minimiser un système de transitions étiquetées. Lorsque la partition est stable, on construit le système de transitions étiquetées quotient. (Rappel : l'ensemble des états du quotient est l'ensemble des classes d'équivalence et il existe une transition $C \xrightarrow{a} C'$ entre deux classes C et C' si et seulement si $q \xrightarrow{a} q'$ pour un état q de C et un état q' de C').

Note bibliographique Sur la base de cet algorithme, Paige & Tarjan [PT87] ont proposé un algorithme de complexité mémoire $O(m+n)$ et de complexité en temps de $O(m \log n)$, dans le cas où l'ensemble des actions A est réduit à une seule étiquette (ce qui revient à traiter une relation binaire et non pas une famille de relations binaires, étiquetées par un élément de A). Cet algorithme a été adapté aux systèmes de transitions étiquetées [Fer90] pour la calcul de la bisimulation forte, et constitue un élément fondamental de ALDÉBARAN [Fer88]. La complexité mémoire est identique à celle de Paige & Tarjan et la complexité en temps est $O(cm \log n)$ où c est le nombre maximal de successeurs d'un état par une action. Kanellakis & Smolka [KS90] ont proposé un algorithme de complexité mémoire $O(m+n)$ et de complexité en temps $O(mn)$. Ils ont proposé une optimisation qui réduit la complexité en temps à $O(k^2 n \log n)$, où k est le nombre maximal de successeurs d'un état. Cet algorithme est programmé dans certains outils de vérification.

Remarque L'idée de l'algorithme de Paige & Tarjan est de procéder de manière analogue au cas des automates déterministes [AHU74] : on mémorise les classes C qui sont décomposées et lorsqu'elles sont utilisées pour décomposer d'autres classes, alors on n'utilise que la plus petite des sous-classes. Dans le cas où le système de transitions étiquetées est déterministe, l'algorithme est linéaire en temps.

Le calcul de la partition stable pour la bisimulation $\tau^* a$ ou l'équivalence observationnelle utilise l'algorithme de bisimulation forte. Pour cela on modifie la relation de transitions étiquetées en construisant la fermeture transitive de cette relation. Cette phase est coûteuse en temps puisque le calcul de la fermeture transitive d'une relation est cubique. Groote & Vaandrager [GV90] ont proposé un algorithme pour la bisimulation de branchement dont la complexité mémoire est $O(m)$ et la complexité en temps est $O(mn)$. Pour des raisons d'efficacité on peut décomposer le calcul de la partition stable pour l'équivalence observationnelle en un calcul pour l'équivalence de branchement (plus efficace) suivi d'un raffinement de partition pour l'équivalence observationnelle (voir le paragraphe méthodologie de la vérification pour la vérification comportementale chapitre précédent).

Implantation Ces algorithmes globaux sont implantés dans les outils de vérification depuis quelques années. Ils servent de briques de base dans le processus de vérification d'applications complexes. Ils permettent de réduire les composants dans l'approche compositionnelle (où le système de transitions étiquetées est représenté sous forme algébrique). Ces algorithmes sont efficaces lorsque le graphe est déjà construit ce qui permet de traiter des graphes de l'ordre du million d'états sur une station de travail. Enfin ces algorithmes peuvent se combiner avec une représentation symbolique des ensembles d'états et de la relation de transitions étiquetées. Cet aspect est abordé au paragraphe suivant.

3.2.2 Raffinement de partition et accessibilité

Nous avons proposé un algorithme [BFH90] appelé *génération de modèle minimal* qui permet de combiner l'accessibilité des ensembles d'états et la réduction par bisimulation. Cet algorithme se combine avec la représentation symbolique des états en particulier avec les BDDs. Dans certains cas l'algorithme permet d'obtenir un quotient fini d'un système infini.

Le principe de l'algorithme est le suivant [BFH90, BFH+92] :

Accessibilité La notion de classe accessible est définie à partir de l'accessibilité des états appartenant à une classe stable (par rapport à la partition courante). Si X est stable par rapport à la partition courante ρ et si il existe une transition d'un état de X vers un état de Y alors tous les états de X ont un successeur dans Y :

$$\forall p, q \in X, p \cap pre_a(Y) \neq \emptyset \text{ si et seulement si } q \cap pre_a(Y) \neq \emptyset$$

Cette propriété est logiquement équivalente à :

$$X \cap pre_a(Y) = \emptyset \text{ ou } X \subseteq pre_a(Y)$$

Prenons l'exemple de la figure 3.1. La classe Z n'est pas accessible puisqu'elle est atteinte par la transition reliant q à r et que seul p est accessible dans Y.

- la classe contenant l'état initial est toujours accessible
- on ne calcule l'accessibilité de classes qu'à partir de classes stables ;

Stabilité Soit X une classe accessible ;

- on raffine la classe XΓ
- si la classe X est inchangéeΓalors sa stabilité est établieΓ
- sinonΓles sous-classes de X remplacent X dans la partition couranteΓet les classes qui précèdent X par la relation de transitions étiquetées deviennent non stables.

Terminaison Un invariant de l'algorithme est : l'ensemble des classes stables est inclus dans l'ensemble des classes accessibles. L'algorithme termine lorsque toutes les classes accessibles sont stables.

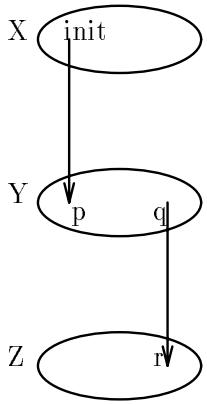

Figure 3.1: Stabilité et accessibilité

Algorithme Nous rappelons brièvement l'algorithme de génération de modèle minimal.

Pour celaΓnous utilisons :

- Les ensembles $\rho\Gamma\pi$ et σ. ρ est la partition couranteΓπ est l'ensemble des classes de ρ qui sont accessiblesΓet σ est le sous-ensemble des classes stables. On a l'invariant :

  $\{\sigma \subseteq \pi \subseteq \rho\}$.

- La fonction Raffiner qui remplace une classe X par ses sous-classes stables par rapport à la partition courante.

- Les fonctions pre_ρ et $post_\rho$ qui associent à une classeΓles classes prédécesseurs et successeurs.

---

begin
ρ:=ρ_{init}; π:=$\{[q_{init}]_\rho\}$; σ:=$\emptyset$;
while $\pi \neq \sigma$ **loop**
choisir et supprimer X *de* $\pi \setminus \sigma$
let $\pi' = Raffiner(X, \rho)$
if $\pi' = \{X\}$ **then**
$\sigma := \sigma \cup \{X\}$; $\pi := \pi \cup post_\rho(X)$
else
$\pi := \pi \setminus \{X\}$
if $\exists Y \in \pi'$ *tel que* $q_{init} \in Y$ **then** $\pi := \pi \cup \{Y\}$ **end if**
σ :=$\sigma \setminus pre_\rho(X)$;
$\rho := (\rho \setminus \{X\}) \cup \pi'$
end if

end loop
end

L'efficacité de la méthode dépend des calculs symboliques effectués sur les ensembles. Cet algorithme a été appliqué dans une version du compilateur Lustre [HRR91].

Il a été aussi appliqué dans le cadre de la vérification comportementale [Ker94]Γen considérant soit des systèmes de transitions étiquetées communicantsΓsoit des Réseaux de Pétri Interprétés. A partir de la description d'une spécification sous forme de Réseaux de Pétri InterprétésΓun modèle minimal est généré vis-à-vis d'une relation de bisimulation. Dans l'implantation que nous avons faiteΓnous nous sommes restreints aux Réseaux de Pétri booléens saufs. Le codage des états et le codage de la relation de transitions étiquetées sont basés sur les BDDs. Nous étudions actuellement l'extension des BDDs pour représenter des domaines finis. Les résultats expérimentaux ont confirmé l'intérêt de cette approcheΓen montrant que l'on pouvait traiter des applications plus importantes.

Nous avons expérimenté cet algorithme dans le cas où les données manipulées sont des ensembles de n-uplets d'entiers approchés par des polyèdres. Un polyèdre peut être caractérisé soit comme l'ensemble des solutions d'un système de contraintes linéairesΓsoit comme l'enveloppe convexe d'un système générateur fini (un ensemble de pointsΓet un ensemble de rayons). Le fait de disposer des deux caractérisations permet de réaliser efficacement l'union et l'intersection de deux polyèdres [Hal79]. Dans le premier casΓon fait l'union des systèmes générateursΓetΓdans le second casΓl'union des systèmes de contraintes. CependantΓla différence ensembliste de deux polyèdres est en général une union de polyèdres. Il faut alors considérer des ensembles de polyèdres pour représenter des classes d'équivalence. Nous pensons néanmoins que cette approche peut être intéressante lorsque l'on considère des sous-classes de polyèdres [TY96].

Note bibliographique Des algorithmes analogues ont été étudiés dans le cadre des systèmes temporisés [LY92ΓACH+92]. Leur complexité dépend d'une part de la taille du quotientΓetΓ d'autre partΓde l'efficacité des opérations sur les ensembles représentés symboliquement.

3.2.3 Approximation

L'algorithme de génération de modèle minimal peut s'appliquer à des spécifications dont le domaine des données est infini ou très grand. Dans [Fer93]Γje me suis intéressé à la terminaison de l'algorithme de génération de modèle minimal lorsque le quotient est fini. En effetΓmême dans ce casΓl'algorithme peut ne pas terminer. Considérons l'exemple suivant :

```
x := 0 ;
do
   x= 0  ⟶ x:= x-1 ||
   x := x+1
od
```

Supposons que nous voulions construire le modèle minimal modulo la bisimulation forte en raffinant la partition initialeΓcomposée des classes $\{x \mid x \in \mathbb{Z} \text{ et } x = 0\}$ et $\{x \mid x \in \mathbb{Z} \text{ et } x \neq 0\}$. Ces classes sont notées respectivement $(x = 0)$ et $(x \neq 0)$.

Après le premier pas de l'algorithmeΓnous obtenons trois classes :

$$C_0 = (x = 0) \quad C_3 = (x = -1) \quad C_4 = (x \neq 0) \text{ et } (x \neq -1).$$

Au pas nΓnous obtenons $n-1$ classes inaccessibles $(x = -j)$Γpour $j = 2, n$. La partition courante est raffinée par rapport à des classes inaccessibles : au pas $n+1$Γla classe $\bigwedge_{i=0}^{n}(x \neq -i)$ est raffinée par rapport à la classe $(x = -n)$ qui est inaccessible.

J'ai proposé une solution partielle dans le cadre de l'interprétation abstraite [CC77]. Considérons un domaine concret D_cΓc'est-à-dire le treillis des parties de l'ensemble des états du programmeΓ et un domaine abstrait D_aΓmuni de la relation d'ordre $\sqsubseteq_{D_a}$Γde la borne supérieure $\sqcup_{D_a}$Γde la borne inférieure $\sqcap_{D_a}$Γde plus petit élément $\bot_{D_a}$ et de plus grand élément $\top_{D_a}$. Le domaine abstrait D_a est relié au domaine concret D_c par une connexion de Galois (α, γ)Γc'est-à-dire que α est une application de D_c dans D_a et γ est une application de D_a dans D_c vérifiant ;

$$\alpha(x_c) \sqsubseteq_{D_a} y_a \text{ si et seulement si } x_c \subseteq \gamma(y_a)$$

Par exempleΓle domaine concret de l'ensemble des parties de $\mathbb{Z}$Γ$\mathcal{P}(\mathbb{Z})$Γest relié au domaine des intervalles L_I par la connexion de Galois (α, γ) telle que $\alpha(X)$ est le plus petit intervalle contenant X et $\gamma(Y)$ est l'ensemble des éléments de $\mathbb{Z}$ appartenant à l'intervalle Y.

J'utilise aussi des opérateurs d'élargissement [CC76] et de rétrécissement qui permettent d'extrapoler la limite d'une suite au bout d'un petit nombre de pas de calcul. L'opérateur d'élargissement vérifie les deux propriétés suivantes :

1. la borne supérieure de deux éléments x et y est plus petite que x élargi par y (noté $x \bigtriangledown y$)Γ
2. pour toute suite croissanteΓ$(x_i)_i$Γla suite $(y_i)_i$ définie par $y_0 = x_0, y_{i+1} = y_i \bigtriangledown x_{i+1}$Γest stationnaire à partir d'un certain i.

Par exemple dans le domaine abstrait L_I des intervalles muni de la relation d'ordre $\sqsubseteq_{L_I}$ de la borne supérieure $\sqcup_{L_I}$ de la borne inférieure $\sqcap_{L_I}$ de plus petit élément $\bot_{L_I}$ et de plus grand élément $\top_{L_I}$ nous pouvons choisir [CC76] l'opérateur d'élargissement $\bigtriangledown$:

$$\begin{array}{lcl} \bot \bigtriangledown X & = & X \bigtriangledown \bot = X \\ [l_0, u_0] \bigtriangledown [l_1, u_1] & = & [\text{si } l_1 < l_0 \text{ alors } -\infty \text{ sinon } l_0, \text{si } u_1 > u_0 \text{ alors } +\infty \text{ sinon } u_0]. \end{array}$$

De manière générale étant donné un opérateur d'élargissement $\bigtriangledown$ si F est monotone sur D_a son plus petit point fixe peut être approché supérieurement par la suite :

$$\begin{array}{lcll} X_0 & = & \bot_{D_a} & \\ X_{i+1} & = & X_i & \text{si } F(X_i) \sqsubseteq_{D_a} X_i \\ & = & X_i \bigtriangledown F(X_i) & \text{sinon} \end{array}$$

La combinaison de l'interprétation abstraite et de la génération de modèle minimal est basée sur une approximation supérieure de l'ensemble des états accessibles et une approximation inférieure de la fonction de raffinement (on raffine plus que nécessaire). Dans l'algorithme raffiner une classe X par une classe Y consiste à remplacer X par $X \bigcap pre_a(Y)$ et par $X \setminus pre_a(Y)$. Il se peut que l'opération de différence ensembliste ne puisse être abstraite dans le domaine abstrait. C'est le cas lorsque l'abstraction du domaine des entiers est celui des intervalles et plus généralement l'abstraction d'un produit cartésien d'entiers est celui des polyèdres. Considérons alors une approximation inférieure de la fonction raffiner qui consiste lorsque c'est possible à remplacer $X \setminus pre_a(Y)$ par ses composants i.e. des éléments images du domaine abstrait. Nous définissons ainsi la notion de *base*. Une base est un ensemble d'éléments du domaine abstrait tel que chaque élément du domaine concret est la borne supérieure d'un nombre fini d'images d'éléments de la base. Par exemple si nous considérons à nouveau des classes représentées par des polyèdres la différence ensembliste produit un ensemble de classes chacune correspondant à un polyèdre.

Etant donné un opérateur d'élargissement $\bigtriangledown$ sur D_a et une base $\beta = \{B_i \mid i \in I\}$ au lieu de calculer l'approximation précédente nous calculons les solutions du système d'équations ci-dessous où $F_i(X) = F(X) \sqcap_{D_a} B_i$:

$$\begin{array}{lcll} X_i^0 & = & \bot_{D_a} & \\ X_i^{k+1} & = & X_i^k & \text{si } F_i(\bigsqcup_{j=1}^{n}{}_{D_a} X_j^k) \sqsubseteq_{D_a} X_i^k \\ & = & (X_i^k \bigtriangledown F_i(\bigsqcup_{j=1}^{n}{}_{D_a} X_j^k)) \sqcap_{D_a} B_i & \text{sinon} \end{array}$$

Lorsque Acc est la fonction d'accessibilité sur le domaine concret certains éléments de la base peuvent être prouvés inaccessibles. Cela permet de ne pas les inclure dans les classes de la partition courante servant au raffinement. Nous obtenons ainsi des conditions pour calculer une approximation inférieure de la partition finale.

Revenons à l'exemple. Soit $Acc_{L_I} = \alpha \circ Acc \circ \gamma$ définie sur L_I. Après la seconde étape de l'algorithme la classe $(x \neq 0)$ est raffinée en la classe $(x = -1)$ et la classe $(x \neq -1)$ et $(x \neq 0)$. La base est $\beta_2 = \{[0,0], [-\infty,-2], [-1,-1], [1,+\infty]\}$. Soit $Acc^0_{L_I}, Acc^1_{L_I}$ $Acc^2_{L_I}$ et $Acc^3_{L_I}$ les fonctions définies par :

$$\begin{array}{lcl} Acc^0_{L_I}(X) & = & Acc_{L_I}(X) \sqcap_{L_I} [0,0] \\ Acc^1_{L_I}(X) & = & Acc_{L_I}(X) \sqcap_{L_I} [1,+\infty] \\ Acc^2_{L_I}(X) & = & Acc_{L_I}(X) \sqcap_{L_I} [-\infty,-2] \end{array}$$

$$Acc^3_{L_I}(X) \quad = \quad Acc_{L_I}(X) \sqcap_{L_I} [-1,-1]$$

Nous calculons la plus petite solution du système d'équations : $X_i = X_i \triangledown Acc^i_{L_I}(\bigsqcup_{j=1}^{n}{}_{L_I} X_j)$ qui est $X^2_0 = [0,0], X^2_1 = [1,\infty], X^2_2 = \bot_{L_I}, X^2_3 = [-1,-1]$. Le résultat final est

$$\rho_1 = (x = 0) \quad (x = -1) \quad (x \geq 1) \quad (x < -1)$$

Ces techniques peuvent s'appliquer à des systèmes de transitions étiquetées étendus dont les transitions sont étiquetées par des tripletsΓcomprenant une partie conditionΓune partie communication et une partie affectation. Dans le cas où les variables manipulées sont des entiers et les affectations sont des transformations linéairesΓnous pouvons représenter chaque classe de la partition initiale par un polyèdre. Le calcul de la fonction *pre* peut se projeter sur un ensemble fini de polyèdres.

3.3 Algorithmes par analyse locale

3.3.1 Introduction

L'approche par analyse locale est motivée par le calcul des propriétés "à la volée"Γc'est-à-dire sans mémoriser le système de transitions étiquetées avant de le vérifier [Hol85ΓCVWY90ΓJJ91Γ FM90]. Le principe est de parcourir *en profondeur d'abord* un graphe composéΓproduit du système de transitions étiquetées (de la spécification) et de la propriété à vérifier. La figure 3.2 représente le schéma d'un parcours en profondeur d'abord d'un graphe $(Q, \longrightarrow, s_{\text{init}})$. Les états du graphe composé sont de la forme (q, Φ)Γoù Φ est la propriété que doit vérifier l'état de la spécification q. Cette section se veut une synthèse de nos articles.

- Dans [FJJM92]Γnous avons présenté de manière uniforme la vérification comportementaleΓla vérification par automate de Büchi déterministeΓet le test de borne sur les files. L'algorithme qui parcourt en profondeur d'abord le graphe composéΓpeut être paramétré par des fonctions spécifiques à chaque type d'analyse.
- Dans [FM95]Γnous avons utilisé un algorithme en profondeur d'abord pour résoudre un système d'équations booléennesΓexprimant des problèmes de vérification comportementale et logique. La différence avec l'algorithme précédent est l'ajout de la détection des composantes fortement connexes maximalesΓen utilisant une variante non-récursive de l'algorithme de Tarjan [Tar72]) (ce qui permet la détection des boucles de τΓou de branchementΓetc).
- Dans [CFG95]Γl'algorithme qui calcule l'équivalence de testΓpour la suppression des branchements inutilesΓest donné par un système formel.
- Dans [FJJV96b]Γnous avons proposé un algorithme qui parcourt en profondeur d'abord le produit synchrone entre l'objectif de test et la spécificationΓpour vérifier la relation de cohérence et synthétiser un arbre de test.

Structures de données

- Une pile Γ qui représente la séquence d'exécution courante. Un élément de la pile est un couple (s, S) où s est un état de Q et S l'ensemble des successeurs de s restant à explorer.
- Un ensemble V (*Visité*)Γpour mémoriser les états visités autres que ceux de la séquence courante.

Algorithme

```
procedure schema_dfs (s_init : state) is
begin
   Γ := ∅ ; V := ∅
   empiler ((s_init Γ post (s_init ))ΓΓ)
   while (Γ ≠ ∅) loop
        (sΓS) := top (Γ)
        if S ≠ ∅ then
             choisir et supprimer un élément s' de S
             if s' ∉ (V ∪ Γ) then
                  empiler (s'Γpost (s'))ΓΓ)
             end if
        else (*S = ∅ *)
             depiler (Γ) ; V := V ∪ {s}
        end if
   end loop
end
```

Figure 3.2: Schéma itératif de parcours en profondeur d'abord

Ces algorithmes calculent un point fixe $\mathcal{R} = f(\mathcal{R})$ et vérifient que $(q^{\mathrm{S}}_{\mathrm{init}}, \Phi_{\mathrm{init}}) \in \mathcal{R}$. f est un opérateur monotone correspondant soit à une définition d'une relation de comparaison (dans le cas comportemental)Γsoit à une définition d'une relation de satisfaction (dans le cas logique). Nous présentons l'idée générale commune à tous ces algorithmes.

3.3.2 Algorithmes

Dans ce qui suit nous nous intéressons au calcul de la simulation forte de la bisimulation forte de la suppression des branchements inutiles de la cohérence entre l'objectif de test et la spécification et de la satisfaction d'une formule du μ-calcul. A partir de là il est facile d'en déduire le calcul pour d'autres bisimulations ou pour la construction d'un arbre de test.

Le schéma de la figure 3.2 peut être adapté à ces différents calculs en associant des informations à chaque état et en spécialisant la fonction *post*. Pour éviter toute confusion les lettres s s' désignent un état du graphe composé lorsque cet état n'est pas défini par un couple et les lettres p, p', q les états de la spécification ou de la propriété. L'algorithme utilise une pile pour stocker les éléments de la séquence courante du graphe composé et un ensemble d'états visités V. Pour la vérification comportementale la vérification logique et la vérification de la cohérence (dans le cadre du test) l'algorithme calcule un ensemble $\mathcal{C}$ de *racines de cycle* identifiées comme telles pendant le parcours en profondeur : lorsqu'un successeur du sommet de pile est lui-même dans la pile ce successeur est une racine de cycle et est stocké dans $\mathcal{C}$. Parmi cet ensemble un sous-ensemble $\mathcal{S}$ de racines de composantes fortement connexes maximales est calculé (uniquement pour le μ-calcul). Pour ce faire l'algorithme dû à Tarjan [Tar72] associe à chaque état s un numéro en profondeur d'abord DFN(s) et le plus petit des numéros HEAD(s) associé aux états accessibles à partir de s. Lorsque DFN(s) est égal à HEAD(s) s est une racine de composante fortement connexe maximale. Ceci peut être adapté à l'algorithme de la figure 3.2 en associant à chaque état s les entiers DFN(s) et HEAD(s). DFN(s) est initialisé avec la profondeur de la pile au moment où s est empilé. HEAD(s) initialisé à DFN(s) est mis à jour chaque fois qu'un successeur s' de s est rencontré en affectant à HEAD(s) la plus petite des valeurs HEAD(s) et HEAD(s'). La profondeur de la pile initialisée à 0 est incrémentée (resp. décrémentée) à chaque opération empiler (resp. depiler). Pour chaque état du graphe composé (p, Φ) formé d'un état p de la spécification et d'une propriété Φ nous notons $X_{p,\Phi}$ la valeur booléenne associée à cet état. Pour un état s X_s est une combinaison booléenne des X'_s pour $s' \in post(s)$ (voir 3.3.4). L'algorithme calcule les conditions pour que $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}} = \texttt{true} \Rightarrow (q^{\mathrm{S}}_{\mathrm{init}}, \Phi_{\mathrm{init}}) \in \mathcal{R}$ (resp. $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}} = \texttt{false} \Rightarrow (q^{\mathrm{S}}_{\mathrm{init}}, \Phi_{\mathrm{init}}) \notin \mathcal{R}$) Pour le μ-calcul une information booléenne supplémentaire $\sigma(s)$ indiquant la nature du point fixe est associée à chaque racine s : si $s = (p, \nu X \,.\, \Phi)$ (resp. $s = (p, \mu X \,.\, \Phi)$) alors $\sigma(s) = \texttt{true}$ (resp. $\texttt{false}$).

En résumé à chaque état s du graphe composé sont associées les informations suivantes :

- la valeur booléenne $s \in \Gamma$ notée $\Gamma(s)$
- la valeur booléenne $s \in V$ notée $V(s)$
- la valeur booléenne $s \in \mathcal{C}$ notée $\mathcal{C}(s)$
- la valeur booléenne X_s
- la valeur booléenne $s \in \mathcal{S}$ pour le μ-calcul et notée $\mathcal{S}(s)$
- la valeur booléenne $\sigma(s)$ pour le μ-calcul
- les entiers HEAD(s) et DFN(s) pour le μ-calcul.

Ces valeurs sont calculées par l'algorithme (en profondeur d'abord) et l'initialisation de ces informations est faite par nécessité : $\Gamma(s)\Gamma V(s)\Gamma\mathcal{C}(s)$ et $\mathcal{S}(s)$ sont implicitement initialisées à `false`. Ces informations sont mises à jour lorsque s est empiléΓ($\Gamma(s)$ vaut `true`)Γdépilé ($\Gamma(s)$ vaut `false` et $V(s)$ vaut `true`)Γun successeur du sommet de pile ($\mathcal{C}(s)$ vaut `true`). Par ailleursΓ si($\mathcal{C}(s)$ vaut `true` et DFN(s)=HEAD(s))Γalors $\mathcal{S}(s)$ vaut `true`.

L'objectif de l'algorithme est de déterminer la valeur $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}}$.

- InitialementΓla pile contient l'état initial du graphe composéΓet l'ensemble des états visités est videΓ

- *exploration du graphe composé* : (paramétré par la fonction *post* quiΓsuivant le type de validationΓcalcule les états successeurs d'un état du graphe composé). On examine les successeurs du sommet de pile. On détermine ceux qui ne sont pas encore visitésΓceux qui sont dans la pile (racines de cycle ou de composante fortement connexe du graphe composé). On empile les états qui doivent être visités et on initialiseΓen fonction du type de validationΓ$X_{p,\Phi}$Γlorsque $(p,\Phi)\in\mathcal{C}$: par exempleΓpour la vérification comportementaleΓ on calcule généralement un plus grand point fixe (la plus grande relation de bisimulation) ; dans ce casΓon initialise $X_{p,\Phi}$ à `true`.

- *synthèse de la valeur associée à l'état s* : Lorsque tous les successeurs du sommet de pile sont examinésΓl'état s en sommet de pile est rangé dans l'ensemble VΓla valeur X_s associée est calculée (en fonctions des valeurs associées au successeurs)Γet on dépile.

Un ou plusieurs parcours peuvent être nécessaires :

- un seul parcours est suffisantΓdans le cadre de la vérification comportementaleΓlorsqu'un des deux systèmes de transitions étiquetées est déterministeΓet pour la génération d'arbres de test. A la fin de l'algorithmeΓ$X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}}$ = `true` (resp. `false`) si $q^{\mathrm{S}}_{\mathrm{init}}$ satisfait Φ_{init} (resp. si $q^{\mathrm{S}}_{\mathrm{init}}$ ne satisfait pas Φ_{init}).

- à la fin de l'algorithmeΓpour le calcul de la bisimulation dans le cas généralΓle résultat $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}}$ = `false` est toujours valide et signifie $q^{\mathrm{S}}_{\mathrm{init}}$ ne satisfait pas Φ_{init}. En revancheΓle résultat $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}}$ = `true` est valide uniquement si aucune racine de cycle n'a changé de valeur. On rajoute une variable *Stable*Γinitialisée à `true`Γet devenant `false` si une racine passe de la valeur initiale `true` à `false`. Dans ce casΓun autre parcours est nécessaire. Entre deux parcours consécutifsΓon mémorise les valeurs X_s = `false`. L'ensemble des racines dont la valeur devient `false` est croissant. En effetΓde par le fait que f est un opérateur monotone et que l'on calcule un plus grand point fixeΓX_s = `false` est stable. Lorsque *Stable* = `true` et $X_{q^{\mathrm{S}}_{\mathrm{init}},\Phi_{\mathrm{init}}}$ = `true`Γalors $q^{\mathrm{S}}_{\mathrm{init}}$ satisfait Φ_{init}.

- pour le μ-calculΓon applique une idée analogue en initialisant les racines correspondants aux plus petits points fixes (resp. plus grands points fixes) à `false` (resp. `true`). La restriction que nous avons mise sur notre algorithme est que toute composante fortement connexe maximale du graphe composé a des racines de même type. Là encoreΓon mémorise entre deux parcours les valeurs X_s = `false` (resp. X_s = `true`) dans le cas d'un plus grand (resp. plus petit) point fixe.

Il nous reste à voir comment est définie la fonction *post*Γ sous-section 3.3.3Γ et quelle est l'expression booléenne associée à X_sΓsous-section 3.3.4.

3.3.3 Définition de la fonction *post* du graphe composé

Soit $S = (Q^{\mathrm{S}}, A, \{\xrightarrow{a}\}_{a\in A}, q^{\mathrm{S}}_{\mathrm{init}})$ le système de transitions étiquetées de la spécification et soit $\mathcal{L}$ le langage de propriétés. $\mathcal{L}$ désigne

- soit les systèmes de transitions étiquetées $P = (Q^{\mathrm{P}}, A, \{\xrightarrow{a}\}_{a\in A}, q^{\mathrm{P}}_{\mathrm{init}})$Γdans le cas de la vérification comportementaleΓ
- soit le μ-calculΓ
- soit les objectifs de testΓavec un ensemble distingué d'états d'acceptation noté `Accept`.

Soit Φ_{init} la propriété à prouver à l'état initial $q^{\mathrm{S}}_{\mathrm{init}}$. Le graphe composé $\mathcal{G} = (Q, \longrightarrow, q_{\mathrm{init}})$ est défini par :

- $q_{\mathrm{init}} = (q^{\mathrm{P}}_{\mathrm{init}}, \Phi_{\mathrm{init}})\Gamma$
- la fonction *post* : $p \longrightarrow q$ si et seulement si $q \in post(p)$.

Vérification comportementale L'ensemble $Q \subseteq Q^{\mathrm{S}} \times Q^{\mathrm{P}}$ des états du graphe composé est construit par induction. C'est le plus petit ensemble vérifiant :

- $(q^{\mathrm{S}}_{\mathrm{init}}, q^{\mathrm{P}}_{\mathrm{init}}) \in Q\Gamma$
- si $(p^{\mathrm{S}}, p^{\mathrm{P}}) \in Q$ alors$\Gamma post(p^{\mathrm{S}}, p^{\mathrm{P}}) \subseteq Q\Gamma$où :

Préordre	$post(p^{\mathrm{S}}, p^{\mathrm{P}})$	$=$	$\{(q^{\mathrm{S}}, q^{\mathrm{P}}) \mid \exists\lambda\,.\, p^{\mathrm{S}} \xrightarrow{\lambda} q^{\mathrm{S}} \wedge p^{\mathrm{P}} \xrightarrow{\lambda} q^{\mathrm{P}}\}$	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \neq \emptyset$ et si $\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \subseteq \mathcal{A}\mathrm{ct}(p^{\mathrm{P}})$
	$post(p^{\mathrm{S}}, p^{\mathrm{P}})$	$=$	$\emptyset$	sinon
Bisimulation	$post(p^{\mathrm{S}}, p^{\mathrm{P}})$	$=$	$\{(q^{\mathrm{S}}, q^{\mathrm{P}}) \mid \exists\lambda\,.\, p^{\mathrm{S}} \xrightarrow{\lambda} q^{\mathrm{S}} \wedge p^{\mathrm{P}} \xrightarrow{\lambda} q^{\mathrm{P}}\}$	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \neq \emptyset$ et si $\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) = \mathcal{A}\mathrm{ct}(p^{\mathrm{P}})$
	$post(p^{\mathrm{S}}, p^{\mathrm{P}})$	$=$	$\emptyset$	sinon
Suppression des Branchement	$post(p^{\mathrm{S}}, q^{\mathrm{S}})$	$=$	$\{(p'^{\mathrm{S}}, q'^{\mathrm{S}})\}$	si $p^{\mathrm{S}} \xrightarrow{a} p'^{\mathrm{S}}$ et $q^{\mathrm{S}} \xrightarrow{a} q'^{\mathrm{S}}$
		$=$	$\{(p^{\mathrm{S}}_0, q^{\mathrm{S}}_0), (p^{\mathrm{S}}_1, q^{\mathrm{S}}_1)\}$	si $p^{\mathrm{S}} \xrightarrow{c_0} p^{\mathrm{S}}_0$ et $p^{\mathrm{S}} \xrightarrow{c_1} p^{\mathrm{S}}_1$ et si $q^{\mathrm{S}} \xrightarrow{c_0} q^{\mathrm{S}}_0$ et $q^{\mathrm{S}} \xrightarrow{c_1} q^{\mathrm{S}}_1$
		$=$	$\{(p^{\mathrm{S}}_0, q^{\mathrm{S}}), (p^{\mathrm{S}}_1, q^{\mathrm{S}})\}$	si $p^{\mathrm{S}} \xrightarrow{c_0} p^{\mathrm{S}}_0$ et $p^{\mathrm{S}} \xrightarrow{c_1} p^{\mathrm{S}}_1$
		$=$	$\{(p^{\mathrm{S}}, q^{\mathrm{S}}_0), (p^{\mathrm{S}}, q^{\mathrm{S}}_1)\}$	si $q^{\mathrm{S}} \xrightarrow{c_0} q^{\mathrm{S}}_0$ et $q^{\mathrm{S}} \xrightarrow{c_1} q^{\mathrm{S}}_1$
		$=$	$\emptyset$	sinon

Génération d'arbres de test L'ensemble des états est défini de manière analogue. La fonction *post* est définie comme suit :

$$
\begin{array}{lcl}
post((p^{\mathrm{OT}}, p^{\mathrm{S}})) & = & \{((q^{\mathrm{OT}}, q^{\mathrm{S}})) \mid \exists \alpha \,.\, p^{\mathrm{OT}} \xrightarrow{\alpha}_{\mathrm{OT}} q^{\mathrm{OT}} \wedge p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}}\} \\
 & & \cup \{(p^{\mathrm{OT}}, q^{\mathrm{S}}) \mid \exists \alpha \,.\, \neg(p^{\mathrm{OT}} \xrightarrow{\alpha}_{\mathrm{OT}}) \wedge p^{\mathrm{S}} \xrightarrow{\alpha}_{\mathrm{S}} q^{\mathrm{S}} \wedge (p^{\mathrm{OT}}, p^{\mathrm{S}}) \notin \mathbf{Accept} \times \{q^{\mathrm{S}}_{\mathrm{init}}\} \\
 & = & \emptyset \text{ sinon}
\end{array}
$$

Vérification logique La fonction *post* est définie par induction sur la structure des formules :

$$
\begin{array}{lcl}
post(p, \mathrm{true}) & = & \emptyset \\
post(p, \mathrm{false}) & = & \emptyset \\
post(p, \varphi_1 \vee \varphi_2) & = & post(p, \varphi_1) \cup post(p, \varphi_2) \\
post(p, \varphi_1 \wedge \varphi_2) & = & post(p, \varphi_1) \cup post(p, \varphi_2) \\
post(p, < a > \varphi) & = & \begin{cases} \{(p', \varphi) \mid p \xrightarrow{a} p'\} \\ \emptyset \text{ si} \neg(p \xrightarrow{a}) \end{cases} \\
post(p, [a]\varphi) & = & \begin{cases} \{(p', \varphi) \mid p \xrightarrow{a} p'\} \\ \emptyset \text{ si} \neg(p \xrightarrow{a}) \end{cases} \\
post(p, \sigma X.\varphi) & = & post(p, \varphi[\sigma X.\varphi / X]) \text{pour } \sigma \in \{\mu, \nu\}
\end{array}
$$

3.3.4 Evaluation du graphe composé

Nous donnons ici les régles de calcul pour synthétiser la valeur X_s associée à un état $s\Gamma$ en fonction des valeurs $X_{s'}$ associées aux successeurs s' de s. Dans le cas présentΓ X_s est une combinaison booléenne des $X_{s'}$.

Vérification comportementale

Préordre	$X_{p^{\mathrm{S}},p^{\mathrm{P}}}$	$=$	$\bigwedge_{a \in A} \bigwedge_{p^{\mathrm{S}} \xrightarrow{a} q^{\mathrm{S}}} \bigvee_{p^{\mathrm{P}} \xrightarrow{a} q^{\mathrm{P}}} X_{q^{\mathrm{S}},q^{\mathrm{P}}}$	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \neq \emptyset$ et si $\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \subseteq \mathcal{A}\mathrm{ct}(p^{\mathrm{P}})$
		$=$	`true`	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) = \emptyset$
		$=$	`false`	sinon
Bisimulation forte	$X_{p^{\mathrm{S}},p^{\mathrm{P}}}$	$=$	$\bigwedge_{a \in A} \bigwedge_{p^{\mathrm{S}} \xrightarrow{a} q^{\mathrm{S}}} \bigvee_{p^{\mathrm{P}} \xrightarrow{a} q^{\mathrm{P}}} X_{q^{\mathrm{S}},q^{\mathrm{P}}} \wedge \bigwedge_{p^{\mathrm{P}} \xrightarrow{a} q^{\mathrm{P}}} \bigvee_{p^{\mathrm{S}} \xrightarrow{a} q^{\mathrm{S}}} X_{q^{\mathrm{S}},q^{\mathrm{P}}}$	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) \neq \emptyset$ et si $\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) = \mathcal{A}\mathrm{ct}(p^{\mathrm{P}})$
		$=$	`true`	si$\mathcal{A}\mathrm{ct}(p^{\mathrm{S}}) = \mathcal{A}\mathrm{ct}(p^{\mathrm{P}}) = \emptyset$
		$=$	`false`	sinon
Suppresion des Branchements	$X_{p^{\mathrm{S}},p^{\mathrm{S}}}$	$=$	`true`	
	$X_{p^{\mathrm{S}},q^{\mathrm{S}}}$	$=$	$X_{p'^{\mathrm{S}},q'^{\mathrm{S}}}$	si $p^{\mathrm{S}} \xrightarrow{a} p'^{\mathrm{S}}$ et $q^{\mathrm{S}} \xrightarrow{a} q'^{\mathrm{S}}$
		$=$	$X_{p_0^{\mathrm{S}},q_0^{\mathrm{S}}} \wedge X_{p_1^{\mathrm{S}},q_1^{\mathrm{S}}}$	si $p^{\mathrm{S}} \xrightarrow{c_0} p_0^{\mathrm{S}}$ et $p^{\mathrm{S}} \xrightarrow{c_1} p_1^{\mathrm{S}}$ et si $q^{\mathrm{S}} \xrightarrow{c_0} q_0^{\mathrm{S}}$ et $q^{\mathrm{S}} \xrightarrow{c_1} q_1^{\mathrm{S}}$
		$=$	$X_{p_0^{\mathrm{S}},q^{\mathrm{S}}} \wedge X_{p_1^{\mathrm{S}},q^{\mathrm{S}}}$	si $p^{\mathrm{S}} \xrightarrow{c_0} p_0^{\mathrm{S}}$ et $p^{\mathrm{S}} \xrightarrow{c_1} p_1^{\mathrm{S}}$ et si $\neg(q^{\mathrm{S}} \xrightarrow{c_0})$ et $\neg(q^{\mathrm{S}} \xrightarrow{c_1})$
		$=$	$X_{p^{\mathrm{S}},q_0^{\mathrm{S}}} \wedge X_{p^{\mathrm{S}},q_1^{\mathrm{S}}}$	si $q^{\mathrm{S}} \xrightarrow{c_0} q_0^{\mathrm{S}}$ et $q^{\mathrm{S}} \xrightarrow{c_1} q_1^{\mathrm{S}}$ et si $\neg(p^{\mathrm{S}} \xrightarrow{c_0})$ et $\neg(p^{\mathrm{S}} \xrightarrow{c_1})$
		$=$	`false`	sinon

La valeur initiale associée aux racines de cycles est `true`.

Génération d'arbres de test (équations de vérification de la cohérence)

$$
\begin{array}{lcl}
X_{p^{\mathrm{OT}},q^{\mathrm{S}}_{\mathrm{init}}} & = & \texttt{true} \text{ si } \quad p^{\mathrm{OT}} \in \texttt{Accept} \\
X_{p^{\mathrm{OT}},p^{\mathrm{S}}} & = & \texttt{false} \text{ si} post((p^{\mathrm{OT}},p^{\mathrm{S}})) = \emptyset \\
 & = & \bigvee_{a\in A} \bigvee_{p^{\mathrm{S}} \xrightarrow{a} q^{\mathrm{S}}} X_{p^{\mathrm{OT}},q^{\mathrm{S}}} \quad \text{si} \quad p^{\mathrm{OT}} \in \texttt{Accept} \text{ et } p^{\mathrm{S}} \neq q^{\mathrm{S}}_{\mathrm{init}} \\
X_{p^{\mathrm{OT}},p^{\mathrm{S}}} & = & \bigwedge_{a\in A} \bigwedge_{p^{\mathrm{OT}} \xrightarrow{a} q^{\mathrm{OT}}} Y_{q^{\mathrm{OT}},p^{\mathrm{S}},a} \\
Y_{q^{\mathrm{OT}},p^{\mathrm{S}},a} & = & \bigvee_{p^{\mathrm{S}} \xrightarrow{a} q^{\mathrm{S}}} X_{q^{\mathrm{OT}},q^{\mathrm{S}}} \vee \bigvee_{b\neq a} \bigvee_{p^{\mathrm{S}} \xrightarrow{b} q^{\mathrm{S}}} Y_{q^{\mathrm{OT}},q^{\mathrm{S}},a}
\end{array}
$$

La valeur initiale associée aux racines de cycles est `false` (En effet l'on cherche à atteindre obligatoirement l'état initial par l'intermédiaire d'un postambule).

Vérification logique

$$
\begin{array}{rclrcl}
X_{p^{\mathrm{S}},\mathrm{true}} & = & \texttt{true} & X_{p^{\mathrm{S}},\mathrm{false}} & = & \texttt{false} \\
X_{p^{\mathrm{S}},<a>\varphi} & = & \bigvee_{p^{\mathrm{S}} \xrightarrow{a} p'^{\mathrm{S}}} X_{p'^{\mathrm{S}},\varphi} & X_{p^{\mathrm{S}},<a>\varphi} & = & \texttt{false} \text{ si} \neg(p \xrightarrow{a}) \\
X_{p^{\mathrm{S}},[a]\varphi} & = & \bigwedge_{p^{\mathrm{S}} \xrightarrow{a} p'^{\mathrm{S}}} X_{p'^{\mathrm{S}},\varphi} & X_{p^{\mathrm{S}},[a]\varphi} & = & \texttt{true} \text{ si} \neg(p \xrightarrow{a}) \\
X_{p^{\mathrm{S}},\varphi_1\wedge\varphi_2} & = & X_{p^{\mathrm{S}},\varphi_1} \wedge X_{p^{\mathrm{S}},\varphi_2} & X_{p^{\mathrm{S}},\varphi_1\vee\varphi_2} & = & X_{p^{\mathrm{S}},\varphi_1} \vee X_{p^{\mathrm{S}},\varphi_2} \\
X_{p^{\mathrm{S}},\nu X.\varphi} & = & X_{p^{\mathrm{S}},\varphi[\nu X.\varphi/X]} & X_{p^{\mathrm{S}},\mu X.\varphi} & = & X_{p^{\mathrm{S}},\varphi[\mu X.\varphi/X]}
\end{array}
$$

La valeur initiale associée à une racine de cycle de la forme $(p, \nu x.\phi)$ (resp. de la forme $(p, \mu x.\phi)$) est `true` (resp. `false`).

3.3.5 Remarques sur l'implantation

Nous avons décrit les informations minimales devant être mémorisées (une valeur booléenne peut être mémorisée sur un bit).

D'autres solutions sont envisageables suivant le compromis temps/mémoire :

- une autre solution a été retenue pour le calcul de la suppression des branchements dans [CFG95]. A chaque couple d'états l'est associée la liste des hypothèses dont il dépend. On obtient ainsi une complexité mémoire quadratique et une complexité en temps linéaire.
- dans le cas du μ-calcul l'on pourrait identifier l'pour chaque état l'la composante fortement connexe maximale à laquelle il appartient. Dans ce cas l'lorsqu'on détecte une racine de composante fortement connexe maximale non stable l'on la parcourt de nouveau.

3.3.6 Remarques sur la complexité

Soit m (resp. n) le nombre de transitions (resp d'états) du produit.

La complexité en mémoire est $O(n)$. En effetΓla relation de transitions étiquetées n'est jamais mémorisée.

La complexité en temps d'un parcours est $O(m)$. La complexité globale en temps dépend du problème traité :

- pour la bisimulationΓdans le cas généralΓla complexité est $O(mn)$ (au plus n parcours)Γ
- pour la bisimulationΓdans le cas où un des deux systèmes est déterministeΓla complexité est $O(m)$Γ(un seul parcours)Γ
- Pour les bisimulations autres que la bisimulation forteΓle calcul du langage fait augmenter la complexité. Dans la réalisation de l'algorithmeΓnous avons choisi de ne traiter que des propriétés qui sont sans action interne.
- Pour la génération d'arbres de testΓla complexité est $O(m)$.

3.3.7 Note Bibliographique

L'application des algorithmes basés sur les parcours en profondeur d'abordΓdans les domaines de la vérificationΓ remonte à quelques années. Cette approche a été proposée par G. Holzmann [Hol85]. Dans [Lar92]Γl'objectif est de vérifier un sous-ensemble du μ-calcul qui ne contient que des plus grands points fixes. Dans [JJ91ΓCVWY90]Γc'est la vérification de formules de logique temporelle linéaire qui est développéeΓainsi que la vérification des automates de Büchi déterministes et le test de borne des files. Nous nous sommes intéressésΓdans [FM90ΓFM91]Γà la vérification comportementale.

Plus récemmentΓ[And92Γ VWL94Γ AV95] ont posé le problème de la vérification comme un problème de résolution d'équations booléennes.

Par ailleursΓ[Cle90ΓCS91ΓLar92ΓBVW94ΓAnd92] se sont intéressés aux algorithmes par analyse locale pour la vérification logique. Le problème consiste à trouver des sous-logiques pour lesquelles l'algorithme de vérification est linéaire.

3.4 Bilan

Nous tirons quelques conclusions sur l'utilisation des algorithmes.

Algorithmes par analyse globale

L'utilisation la plus fréquente de l'extension de l'algorithme de Paige & Tarjan est utilisée pour la minimisation. Il peut être utilisé en sortie d'un compilateur produisant des systèmes de transitions étiquetées. Il permet aussiΓlors d'une approche compositionnelleΓde minimiser au fur et à mesure les résultats intermédiaires.

L'algorithme de génération de modèle minimal a été expérimenté avec les BDDsΓpour la vérification dans le domaine des protocolesΓet pour la compilation du langage Lustre. L'algorithme

fonctionne indépendamment de la nature des données. En effet, il prend en paramètre une relation d'équivalence et une représentation symbolique du système de transitions étiquetées. Une version modifiée de cet algorithme fonctionne dans KRONOS pour des *régions*, qui sont des formes particulières de contraintes linéaires [TY96]. Actuellement, il fonctionne dans CADP avec

- des réseaux de Pétri saufs dont les variables sont booléennes,
- des compositions de systèmes de transitions étiquetées communicants,
- un système de transitions étiquetées. Il est peu utilisé dans ce cas, car moins efficace que l'algorithme de Paige & Tarjan.

Nous n'avons pas évoqué tous les travaux concernant les algorithmes sur les systèmes de transitions étiquetées. Emerson et Lei [EL86] décrivent un algorithme par analyse globale pour le μ-calcul alterné. Arnold et Crubillé [AC88] ont proposé un algorithme linéaire pour calculer un point fixe. On peut trouver d'autres travaux sur le μ-calcul dans [CS91] et sur les systèmes de transitions [VL92].

Algorithmes par analyse locale

Nous avons vu qu'ils permettent de résoudre plusieurs types de problèmes et de traiter plusieurs formes intermédiaires. Les paramètres de cet algorithme sont la fonction *post* et les fonctions ensemblistes relatives aux états.

L'expérience pour la vérification comportementale nous a montré que le temps d'exécution n'atteignait jamais le cas le plus défavorable. En effet, pour la plupart des exemples, l'algorithme termine en une ou deux itérations. Pour le μ-calcul à la volée, le bilan est plus mitigé. A notre avis, l'algorithme est trop général et il faut le spécialiser pour des sous-ensembles du μ-calcul.

Lorsque la propriété n'est pas satisfaite, la production de diagnostic est plus aisée avec un algorithme par analyse locale.

On peut mémoriser plus ou moins d'informations sur les états visités. Par exemple, dans le cas de la vérification comportementale, on mémorise les racines de cycles et les états du graphe composé représentant des états non-équivalents.

4 Une plate-forme pour la validation

Il existe un grand nombre d'outils pour la vérification. Je ne citerai que deux outils développés par des équipes avec qui nous avons eu des collaborations : MEC [Arn89] développé à Bordeaux et AUTO/AUTOGRAPH [RdS90]Γdéveloppé à Sophia-Antipolis.

Ce chapitre est composé de trois parties. Dans la section 4.1Γnous faisons une brève présentation de CADP. Puis nous présentons plus en détailΓdans la section 4.2Γla partie à laquelle nous avons contribué : la vérification comportementale et la génération d'arbres de test. EnfinΓdans la section 4.3Γnous concluons sur quelques perspectives d'évolution. Dans la présentation qui suitΓj'ai gardé le terme "graphe"Γdans la mesure où il apparaît dans la distribution de CADP. MaisΓil est employé dans un sens équivalent à "système de transitions étiquetées".

4.1 Présentation générale de CADP

CADP est un environnement pour le développement des protocoles de communication. Il comporte plusieurs outils qui intègrent différentes techniques de vérification. La première version est issue de la mise en commun des compilateurs pour LOTOSΓCAESAR et CAESAR.ADTΓet du vérificateur comportemental ALDÉBARAN. Plusieurs logiciels composent CADP : ALDÉBARANΓ BCGΓ CAESARΓ CAESAR.ADTΓ OPEN/CAESARΓ EVALUATORΓ EXP.OPENΓ TGV et XTL. Tous ces composants sont accessibles par l'intermédiaire d'une interface graphique développée dans le cadre du projet EUCALYPTUS. Nous avons plus particulièrement contribué à ALDÉBARANΓ EXP.OPEN et TGV.

4.1.1 Langage et compilateurs

A l'heure actuelleΓ CADP prend en entrée des descriptions pouvant être écrites dans plusieurs formalismes :

- des programmes écrits en LOTOS. CAESAR et CAESAR.ADT traduisent ces descriptions vers un programme C qui permet de générer le système de transitions étiquetées correspondant au programme LOTOS. Cela correspond en fait à une représentation implicite du système de transitions étiquetées.
- des systèmes de transitions étiquetéesΓ
- des systèmes de transitions étiquetées communicantsΓdes réseaux de Pétri.

Tout compilateur produisant des systèmes de transitions étiquetées peut utiliser la partie vérification. De mêmeΓla sortie de CAESAR peut être imprimée en utilisant d'autres formats.

4.1.2 Présentation des outils

ALDÉBARAN permet de minimiser ou de comparer des systèmes de transitions étiquetées modulo plusieurs relations d'équivalence et de préordre : la bisimulation forteΓla bisimulation observationnelleΓla bisimulation de délaiΓla bisimulation de branchementΓla bisimulation $\tau^* a$Γ le préordre de sûreté et l'équivalence de sûreté. Lorsque deux systèmes de transitions étiquetées ne sont pas équivalentsΓun diagnostic sous forme de séquences d'exécution est produit.

BCG est à la fois un format pour la représentation explicite compacte des systèmes de transitions étiquetées et un ensemble de bibliothèques. Les outils suivants permettent de traiter ce format :

- BCG_IO est un traducteur du format BCG vers ou depuis d'autres formats (comme par exemple les formats ALDEBARANΓFC2ΓMECΓAUTOΓetc)Γ
- BCG_OPEN est une passerelle entre le format BCG et l'environnement OPEN /CAESARΓ
- BCG_DRAW permet de visualiser des graphes en deux dimensionsΓ
- BCG_EDIT est un éditeur graphique interactif.

CAESAR est un compilateur qui traduit un programme LOTOS vers un système de transitions étiquetées. La traduction est réalisée en plusieurs étapes. PremièrementΓla description est traduite vers une algèbre de processus simplifiée. Puis un réseau de Pétri est généré. Un système de transitions étiquetées peut être produit à partir de ce réseau par analyse d'accessibilité.

CAESAR.ADT est un compilateur qui traduit la partie données de Lotos vers des bibliothèques de types et de fonctions C. Chaque sorte LOTOS (resp. chaque opération) est traduite en un type C (resp. une fonction C). CAESAR.ADT produit aussi des fonctions de comparaisonΓd'impression et d'itération sur le domaine de valeurs. L'utilisateur doit différencier les constructeurs des autres opérateurs et ne pas définir les constructeurs par un système d'équations.

EVALUATOR est un évaluateur de formules du μ-calcul. Cet évaluateur est basé sur OPEN/CAESAR.

OPEN/CAESAR est un ensemble de modules qui permet de combiner des algorithmes de parcours de système de transitions étiquetées et des algorithmes de validation.

- le module *graphe* fournit des primitives permettant le parcours et l'accès aux constituants d'un graphe (états et transitions étiquetées).
- le module *stockage* fournit les structures de données qui permettent le stockage complet ou partiel du grapheΓ
- le module exploration contient l'algorithme de parcours du graphe qui effectue la validation.

Les structures de données et les fonctions des deux premiers modules sont accessibles à l'écrivain d'algorithmes. Le troisième module utilise ces deux premiers modules et permet au concepteur d'algorithmes de vérification de programmer facilement lui-même ses algorithmes de parcours.

EXP.OPEN convertit une représentation algébrique d'un système de transitions étiquetées en une représentation implicite.

TGV génère à partir de deux systèmes de transitions étiquetées (l'un pour la spécification et l'autre pour l'objectif de test) un arbre de test écrit en TTCN (Tree and Tabular Comnined Notation). Cette génération est réalisée en plusieurs étapes. Tout d'abordΓle système de transitions étiquetées est transformé en un système de transitions étiquetées représentant le comportement observable de la spécification dans l'environnement de test. Cette transformation tient compte de l'architecture de test. D'autres transformations comme l'abstractionΓla déterminisation et la minimisation sont appliquées. Un parcours en profondeur d'abordΓconstruit l'arbre de test contenant les informations pour générer du TTCN.

XTL est un langage fonctionnel conçu pour faciliter l'implémentation de divers opérateurs de logique temporelleΓévalués sur un système de transitions étiquetées au format BCG. Le langage contient des types prédéfinis tels que les booléensΓles caractères et les entiers et permet la définition de types et de fonctions. Il offre des primitives d'accès aux états et aux transitions étiquetées.

4.2 Outils pour la vérification et le Test

Dans cette partieΓnous faisons le lien entre les implantations et les algorithmes présentés au chapitre précédent.

Représentation des systèmes de transitions étiquetées

Un système de transitions étiquetées peut être représenté :

de façon explicite Cette représentation comporte un descripteur et un ensemble de transitions étiquetées. Le descripteur contient le nombre de transitions et un nombre N qui représente la borne supérieure du nombre d'états. Chaque transition est de la forme (étatΓactionΓ état)Γoù état est codé par un entier strictement inférieur à N et action est une chaîne de caractères.

de façon implicite Cette représentation est constituée d'un ensemble de fonctions qui permettent le parcours de la relation de transitions étiquetéesΓle stockage des étatsΓla comparaison entre les étatsΓet la comparaison entre les actions (voir OPENÆSAR).

de façon algébrique Cette représentation consiste en un terme. L'ensemble des termes est défini comme suit :

- un système de transitions étiquetées est un termeΓ
- la composition parallèle de plusieurs termes est un terme. Cette composition parallèle permet de synchroniser plusieurs termes sur un ensemble d'actions.
- l'application d'un opérateur d'abstraction sur un terme est un terme ; un tel opérateur prend en argument un ensemble d'actions et un terme. Il transforme ce dernier en renommant en τ toutes les actions qui appartiennent à l'ensembleΓ
- l'application d'un opérateur de renommage sur un terme est un terme. Un tel opérateur est une substitution uniforme d'actions.

4.2.1 La vérification comportementale : ALDÉBARAN

Les systèmes de transitions en entrée d'ALDÉBARAN sont représentés de façon explicite ou algébrique.

Les algorithmes

Les algorithmes peuvent utiliser des techniques énumératives ou symboliques. Les techniques énumératives sont appelées ainsi parce que les états sont énumérés.

Raffinement de Partition L'extension de l'algorithme de Paige & Tarjan est programmée pour la représentation explicite. Il sert pour la minimisation et pour la comparaison de systèmes de transitions étiquetées. Les équivalences sont la bisimulation forte et observationnelle. La comparaison consiste à calculer une forme normale des deux systèmes de transitions étiquetées et de comparer ces formes normales modulo la bisimulation forte. Dans le cas de la bisimulation forteΓla forme normale est le quotient du système initial modulo la bisimulation forte. Dans le cas de l'équivalence observationnelleΓla forme normale est obtenue en calculant la fermeture transitive de la relation de transitions étiquetées par τΓpuis en minimisant le système de transitions étiquetées ainsi obtenuΓpar la bisimulation forte. Inutile de dire queΓpour des systèmes de transitions étiquetées d'une centaine de milliers d'étatsΓcette approche n'est pas performante. Cet algorithme est actuellement surtout utilisé pour la minimisation et la comparaison modulo la bisimulation forte. Ce dernier aspect peut être utile pour tester des compilateurs générant des systèmes de transitions étiquetéesΓen comparant les systèmes de transitions étiquetées générés par deux versions différentes.

Génération minimale basée sur les techniques symboliques Il s'agit de générerΓà partir d'un système de transitions étiquetéesΓun système de transitions étiquetées minimal. Cet algorithme utilise des techniques symboliquesΓcomme les BDDsΓet il est intéressant surtout pour des représentations algébriques. Les bisimulations considérées sont la bisimulation forteΓla bisimulation $\tau^* a$ et la bisimulation de branchement. ActuellementΓ cette génération est effective aussi pour des Réseaux de Pétri booléens saufs (produitsΓpar exempleΓpar le compilateur CÆSAR)Γet des systèmes de transitions étiquetées représentés sous forme algébrique.

Ce travail a été réalisé par Alain KerbratΓdans le cadre de sa thèse [Ker94].

La Vérification comportementale à la volée Le système de transitions étiquetéesΓassocié au programmeΓ est représenté sous forme algébrique ; celui associé à la propriété est représenté sous forme explicite. Pour des raisons d'efficacitéΓnous avons restreint la forme des propriétés. Le système de transitions étiquetées représentant la propriété n'a pas de τ transition. A notre avisΓcette restriction n'est pas génante. Différentes relations ont été implantées : la bisimulation forteΓla bisimulation de branchementΓla τ^*a-bisimulationΓle préordre de sûreté et l'équivalence de sûreté.

Ce travail a été réalisé par Laurent Mounier dans le cadre de sa thèse [Mou92].

Evolution d'Aldébaran

Pendant ma thèse [Fer88]Γj'ai conçu et réalisé la première version d'ALDÉBARAN. Les objectifs initiaux étaient les suivants :

- implanter un algorithme de minimisation de systèmes de transitions étiquetées efficaceΓ basé sur l'extension de Paige & TarjanΓ
- proposer des procédures de décision pour l'équivalence observationnelle et l'équivalence d'acceptation (qui n'est plus utilisée à l'heure actuelleΓcar non maintenue)Γ
- proposer une approche compositionnelleΓ c'est-à-dire utilisant la représentation algébriqueΓpour la vérification comportementaleΓen proposant des stratégies de composition-réduction. Divers opérateurs de composition pouvaient être définisΓainsi qu'un opérateur de renommageΓd'abstraction et de restriction.

Le logiciel ALDÉBARAN atteint une taille critique (plusieurs milliers de lignes de code). Rajouter un nouveau module réalisant un algorithmeΓdevient de plus en plus difficile. Nous nous orientons vers un environnement plus ouvertΓ où sont bien séparés la représentation des donnéesΓ les bibliothèques de gestion de systèmes de transitions étiquetéesΓet le ou les modules réalisant l'algorithme de vérification.

4.2.2 Vérification du μ-calcul

EVALUATOR vérifie des formules du μ-calcul sur des systèmes de transitions étiquetées représentés de manière expliciteΓimplicite ou algébrique. EVALUATOR est un module d'exploration relié aux bibliothèques de OPEN /CAESAR et de EXP.OPEN. Nous avons créé ce dernier module pour pouvoir convertir une représentation algébrique en une représentation impliciteΓ sous forme de fonction. Ce module détermine les communications entre les systèmes de transitions étiquetées de la représentation algébrique et génère la fonction quiΓpour un étatΓcalcule ses successeurs. L'intérêt de cette démarche est de pouvoir ainsi associer à un algorithme un module logicielΓqui est indépendant de la représentation choisie.

4.2.3 La génération d'arbres de test

TGV est le fruit d'une collaboration entre le projet Pampa et Vérimag. L'algorithme a été développé par les deux équipes. TGV a été programmé à Rennes en utilisant les librairies OPEN/CAESAR et en s'inspirant d'un module développé à Grenoble pour mettre en œuvre l'algorithme présenté dans [FJJM92].

Il s'agit de générer un arbre de testÀ partir d'un objectif de test et d'une spécification formelle. Nous présentons la méthode de génération telle qu'elle existe. Cette génération est réalisée par différents outilsfigure 4.1.

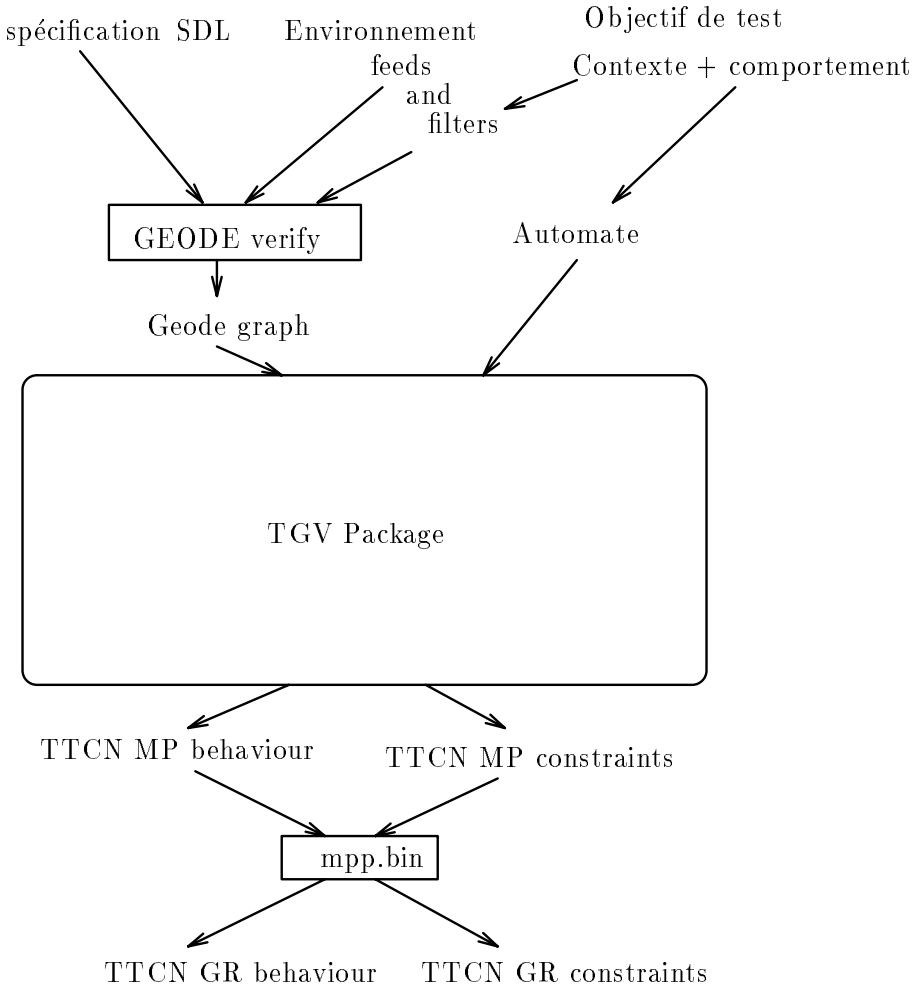

Figure 4.1: Vue externe de Tgv

Etape 1 Le graphe est généré par l'outil GEODE de VERILOG qui a pour entrées la spécification et un ensemble de messages pour fermer la spécification. En effetune spécification GEODE peut être ouverte dans le sens où certaines variables ne sont pas affectées. Les mécanismes de

"feeds" et de "filters" permettent à la fois de fermer les variables (en leur associant des valeurs) et de restreindre des comportements.

Etape 2 Le graphe est transformé en un système de transitions étiquetées au format ALDÉBARAN. Nous faisons l'hypothèse *d'environnement raisonnable* : chaque fois que l'environnement émet un message, il attend la stabilisation du protocole : aucun autre message n'est envoyé avant que tous les messages attendus ne soient reçus. Les entrées du protocole sont transformées en sorties et vice-versa. Les réceptions concurrentes (du point de vue du testeur) sont entrelacées. Les actions internes sont renommées en τ.

Etape 3 Le système de transitions étiquetées est minimisé par ALDÉBARAN en utilisant l'équivalence $\tau^* a$.

Etape 4 L'arbre de test est généré en utilisant l'algorithme présenté au chapitre précédent.

Etape 5 L'arbre de test est transformé en utilisant l'outil `mpp.bin` développé par VERILOG.

Evolution de TGV

A court terme, deux objectifs sont poursuivis : la connexion de TGV à LOTOS et le fonctionnement à la volée pour une spécification LOTOS ou SDL.

4.3 Evolution

A l'heure actuelle, un système de transitions étiquetées est représenté de manière explicite, implicite ou algébrique. Pour modéliser un système critique, on peut soit le modéliser par un système de transitions étiquetées, soit le décrire dans un langage comme LOTOS ou SDL. La connexion entre SDL et CADP, initialisée pour l'étude sur le test, est encore expérimentale.

Une étude a été réalisée [Boz96] pour pouvoir traiter des systèmes de transitions étiquetées étendus communicants. Un état d'un système de transitions étiquetées étendu est un point de contrôle et un ensemble de variables. L'ensemble des points de contrôle est fini et le domaine de chaque variable est fini. Les transitions sont étiquetées par un triplet constitué d'une garde (une expression booléenne), d'une action de communication, et d'un ensemble d'affectations. La communication est réalisée par rendez-vous binaire ou n-aire.

Cette étude porte aussi sur la représentation symbolique des données à domaine fini. Outre des connexions à différentes librairies de diagrammes de décision (TiGeR, BDDS, Berkeley BDDs, Colorado BDDs), une nouvelle librairie a été proposée.

L'orientation générale pour la partie vérification est de pouvoir disposer d'un environnement qui facilite l'interconnexion entre des modules de gestion de données et de parcours de systèmes de transitions étiquetées, de manière à pouvoir développer rapidement de nouveaux algorithmes de vérification.

5 Bilan et Perspectives

Bilan

Dans ce document j'ai essayé de présenter les grandes lignes qui ont guidé mon travail de recherche. Le cadre de ce travail est le développement et l'utilisation des méthodes formelles pour spécifier et valider les systèmes critiques. La démarche adoptée consiste à étudier et à mettre en œuvre des algorithmes performants pour la validation puis de mettre en évidence les limites de ces algorithmes au travers d'études de cas. Bien entendu le développement d'algorithmes efficaces n'est pas indépendant de la représentation du modèle.

Des progrès significatifs (nous avons gagné deux ordres de grandeur) ont été enregistrés quant à la taille des modèles traités. Le contexte d'utilisation des algorithmes est déterminé par l'intuition que l'on a de leur performance. L'extension de Paige & Tarjan est surtout utilisée pour minimiser (modulo la bisimulation forte) des systèmes de transitions étiquetées représenté de manière explicite. Il permet de réduire de 10 à 100 fois des systèmes de transitions étiquetées produits par des compilateurs tout en préservant les propriétés comportementales ou logiques. L'algorithme de génération de modèle minimal est utilisé à partir de réseaux de Pétri de systèmes de transitions étiquetées communicants (représentation algébrique) et de systèmes de transitions étiquetées étendus communicants. Cet algorithme combine le calcul d'accessibilité des états et le calcul de raffinement de partition permettant ainsi de minimiser un modèle pendant sa construction. Il peut être paramétré par diverses bisimulations et peut se combiner avec des représentations symboliques des données comme par exemple les diagrammes de décision (binaires ou n-aires) pour les domaines de données fini ou encore les polyèdres représentant des n-uplets d'entiers. Dans ce dernier cas vu que le domaine est potentiellement infini l'algorithme peut ne pas terminer. Les algorithmes à la volée sont utilisés pour la vérification et le test à partir de descriptions LOTOS SDL ou représentées par des systèmes de transitions étiquetées communicants. La vérification peut être soit comportementale soit logique. Nous avons montré que ces techniques pouvaient être appliquées à l'optimisation de code et à la génération automatique d'arbres de test. Sur ce dernier point nous avons conçu et réalisé TGV à l'aide de primitives de CADP. L'utilisation de TGV a été bien percue par les industriels et nous envisageons de l'intégrer à GÉODE de VERILOG.

Le domaine d'application que nous avons privilégié au travers des études de cas est le domaine des protocoles de communication. Cependant les méthodes et les algorithmes sont suffisamment généraux pour être transposés à d'autres domaines. Par exemple l'algorithme de génération de modèle minimal est utilisé dans deux outils développés à Vérimag : le compilateur LUSTRE et KRONOS.

La structure de CADP facilite le prototypage rapide d'algorithmes de validation et l'interfaçage avec de nouvelles bibliothèques de représentation symbolique des données. Même si nos outils ont progressé il n'en reste pas moins qu'il est difficile de résoudre certains problèmes dans le domaine des protocoles. En effet les algorithmes sont plus performants sur des systèmes à

structures régulièresΓcomme par exempleΓcomposés de n processus identiques (au renommage près) disposés en anneauxΓou encore construits sur le modèle client-serveur.

Perspectives

Je terminerai ce document en évoquant quelques perspectives de recherche concernant les modèlesΓles algorithmes et les outils pour la validation.

Les modèles Le modèle sur lequel nous travaillons actuellement est basé sur la notion de systèmes de transitions étendus communicants. Dans ce cadreΓil reste à étudier et développer les techniques de représentation efficace des structures de données.

Un aspect que je voudrais mentionner est l'utilisation des techniques d'interprétation abstraite (ou analyse statique) employées dans le domaine de la compilation pourΓpar exempleΓla phase d'optimisation globaleΓou le calcul de propriétésΓpermettant de paralléliser au mieux du code. Je vois essentiellement deux applications dans le domaine de la vérification. PremièrementΓ les techniques d'analyse statiqueΓlorsque l'on considère des systèmes de transitions étiquetées étendusΓ peuvent permettre d'optimiser les fonctions d'exploration. DeuxièmementΓon peut calculer des domaines abstraits. Des résultats existent déjàΓfixant les conditions pour qu'une propriété vraie sur le domaine abstrait le reste sur le domaine concret initial. CependantΓil reste à spécialiser des résultats théoriques dans le domaine des protocolesΓet à s'interroger sur la représentation de domaines infinis. D'autres concepts doivent être étudiés dans le cadre des systèmes de transitions étendus communicantsΓcomme la notion de prioritéΓle temps ou la communication par file.

Plus généralementΓle problème de la définition d'un modèle pour les systèmes asynchronesΓen vue de développer des techniques de vérification efficaceΓreste toujours ouvert.

Notre expérience en matière de test est encore récente. Nous avons développé un modèle et une méthode de génération de test dans un cas bien particulier : l'architecture de test est supposée ne contenir qu'un seul dispositif qui interagit avec l'application. Il faut bien entendu étudier la prise en compte d'architectures de test beaucoup plus générales. D'autre problèmes peuvent être abordés comme le lien entre notre méthode de génération de test et le test structurel (i.e. à un niveau syntaxique). On peut aussi étendre l'expressivité de la modélisation des objectifs de testΓen utilisant des formules du μ-calcul (ou d'une autre logique temporelle) à la place de systèmes de transitions étiquetées. D'autres définitions de la cohérence peuvent être étudiéesΓ en liaison avec la notion de contrôlabilité.

Les méthodes et les algorithmes de validation Nous utilisons trois types d'algorithmes : les algorithmes de minimisationΓ basés sur l'extension de Paige & TarjanΓles algorithmes de génération de modèle minimal et les algorithmes de comparaison à la volée. Un certain nombre de travaux restent encore à faire.

Un premier aspect concerne la méthodologie de vérification : les algorithmes constituent des éléments de base intervenant dans une vérification complexeΓnotamment en présence de l'opérateur de composition parallèle. L'enchaînement de ces algorithmes demande une certaine expertise

et est actuellement réalisé manuellement. On peut imaginer des méthodologies de vérification comportant des enchaînements de compositions partielles et de réductions. Ces stratégies de vérification peuvent être déterminées à un niveau structurel en déterminant les sous-comportements destinés à être générés.

Un autre aspect est lié à l'évolution des modèles. Certains algorithmes comme la génération de modèle minimal sont tout à fait adaptés au modèle "systèmes de transitions étiquetées étendus". Il reste à étudier les combinaisons entre les techniques d'interprétation abstraite et les algorithmes de vérification comme par exemple les algorithmes à la volée. Les algorithmes sont paramétrables : l'algorithme de génération de modèle minimal fait appel à la fonction *pre*(qui à un état associe ses prédécesseurs) et aux fonctions ensemblistes (pour partitionner des classes) ; les algorithmes à la volée font appel à la fonction *post*(qui à un état associe ses successeurs) et aux fonctions ensemblistes.

Pour la génération de séquences de test d'autres stratégies peuvent être mises en œuvre pour calculer des préambules des postambules et pour prendre en compte l'asynchronisme qui est générateur d'états et de transitions supplémentaires.

Et bien sûr il reste d'autres algorithmes à inventer !

Les environnements de développement En une dizaine d'années nous avons acquis une certaine compétence dans la vérification de protocoles. L'évolution d'un environnement universitaire de développement comme le nôtre doit à notre avis poursuivre deux objectifs :

- un transfert vers l'industrie des algorithmes bien maîtrisés
- l'expérimentation des techniques les plus avancées sur la représentation des modèles et le développement des algorithmes.

Nous pensons que les algorithmes à la volée ont acquis une maturité qui permet le transfert au sein d'outils industriels. Ceci existe déjà dans l'environnement GEODE qui dispose de techniques de vérification et d'exploration interactives couplées avec le simulateur (appelée *vérification par observateurs*). Nous disposons dans l'équipe d'un accès aux représentations internes de GEODE ce qui nous permet d'expérimenter nos algorithmes et d'accéder aux études de cas SDL.

Enfin ces environnements doivent rester modulaires pour pouvoir accueillir de nouveaux modules sans pertuber le comportement de l'ensemble.

Bibliographie

[AC88] A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transitions systems. *Inform. Process. Lett.*(29):56–66,1988.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill and H. Wong-Toi. Minimization of timed transition systems (extended abstract). In *CONCUR'92, Stony Brook*. LNCS 630, Springer Verlag, août 1992.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and analysis of computer Algorithms*. Addison Wesley, 1974.

[AN82] A. Arnold and M. Nivat. Comportement de processus. *Les mathématiques de l'informatique*, 1982.

[And92] H.R. Andersen. Model checking and boolean graphs. In *ESOP'92, LNCS 582*, 1992.

[Arn89] A. Arnold. Mec: a system for constructing and analysing transition systems. In *International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, pages 117–132. Springer Verlag, 1989.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants*. Masson, 1992.

[AV95] H.R. Andersen and B. Vergauwen. Efficient checking of behavioural relations and modal assertions using fixed-point inversion. In *CAV'95, LNCS 939*, 1995.

[BCM+89] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. Technical report, Carnegie Mellon University, 1989.

[BdS92] A. Bouali and R. de Simone. Symbolic bisimulation minimisation. In *Fourth Workshop on Computer-Aided Verification, Montreal*, june 1992.

[BFG+91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *18th ICALP*. Springer Verlag, july 1991.

[BFH90] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In *Workshop on Computer-aided Verification, Rutgers*, Discrete Mathematics and Theoritical Computer Science. American Mathematical Society, Association for Computing Machinery, juin 1990.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, C. Ratel, and P. Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3), June 1992.

[Bou85] G. Boudol. Calculs de processus et vérification. Technical Report RR424, INRIA, Sophia-Antipolis, 1985.

[Boz96] M. Bozga. Virification formelle de systhmes distribuis diagrammes de dicision multivaluis. Rapport de DEA, Grenoble, June 1996.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. An automata-theoritic approach to branching-time model checking. In *Workshop on Computer-Aided Verification 94, LNCS 818*, 1994.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*. Springer Verlag, 1989.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2th Int. Symp. on Programming*, pages 106–130. Dunod, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, janvier 1977.

[CES83] E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. In *10th. Annual Symp. on Principles of Programming Languages*, 1983.

[CFG95] P. Caspi, J.-C. Fernandez, and A. Girault. An algorithm for reducing binary branchings. In *FST & TCS Bangalore*, Lecture Notes in Computer Science. Springer Verlag, December 1995.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[Cle90] R. Cleaveland. Tableau based model checking in the propositional μ-calculus. *Acta Informatica*, 1990.

[CS91] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation free modal μ-calculus. In *Workshop on Computer-Aided Verification, LNCS 575*, July 1991.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yanakakis. Memory efficient algorithms for the verification of temporal properties. In *Workshop on Computer-Aided Verification*, june 1990.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer -Aided Verification (Aalborg, Denmark)*, number 575 in LNCS, july 1–4 1991.

[EL86] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional Mu-calculus. In *Symposium on Logic in Computer Science*, 1986.

[Fer88] J.-C. Fernandez. *Aldébaran, Un système de vérification par réduction de processus communicants*. PhD thesisΓUniversité de GrenobleΓ1988.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*Γ13(2-3)ΓMay 1990.

[Fer93] J.-C. Fernandez. Abstract interpretation and verification of reactive systems. In *Static Analysis*. LNCS 724 Springer VerlagΓseptembre 1993.

[FJJM92] J.-C. FernandezΓC. JardΓT. JéronΓand L. Mounier. On the fly verification of finite transition systems. *Formal Methods in System Design, Kluwer Academic Publishers*Γ 1:251–273Γ1992.

[FJJV96a] J.-C. FernandezΓC. JardΓT. JéronΓand C. Viho. An experiment in automatic generation of test suites for protocoles with verification technology. *Science of Computer Programming*Γ1996. To appear.

[FJJV96b] J.Cl-. FernandezΓ C. JardΓ T. JéronΓ and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*. LNCS 1102 Springer VerlagΓ 1996.

[FKM93] J.-C. FernandezΓA. KerbratΓand L. Mounier. Symbolic equivalence checking. In *CAV'93, University of Crete*. LNCS 697 Springer VerlagΓjuly 1993.

[FM90] J.-C. Fernandez and L. Mounier. Verifying bisimulations on the fly. In *Proceedings of the Third International Conference on Formal Description Techniques FORTE'90 (Madrid, Spain)*Γpages 91–105. North-HollandΓNovember 1990.

[FM91] J.-C. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalences and preorders. In *Workshop on Computer-aided Verification, Aalborg University, Denmark (CAV'91)*. LNCS 575ΓSpringer VerlagΓjuly 1–4 1991.

[FM95] J.-C. Fernandez and L. Mounier. A local checking algorithm for boolean equation systems. Technical Report Spectre-95-07ΓVerimagΓGrenoble-FranceΓ1995.

[Gir94] A. Girault. Sur la répartition de programmes synchrones. thèseΓINPGΓGrenobleΓ janvier 1994.

[GS90] S. Graf and B. Steffen. Compositional minimisation of finite state processes. In *Workshop on Computer-Aided Verification*ΓRutgersΓJune 1990. LNCS 531ΓSpringer Verlag.

[GV90] Jan Friso Groote and Frits Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *17th ICALP, LNCS 443*Γ pages 626Γ638Γ WarwickΓ1990.

[GW89] R.J. Van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). CS-R 8911ΓCentrum voor Wiskunde en InformaticaΓAmsterdamΓ1989.

[Hal79] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. PhD thesis, Université de Grenoble, 1979.

[Hol85] G. Holzmann. Tracing protocols. *ATT technical journal*, 64(10), 1985.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, août 1991.

[ISO92] OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992.

[JJ91] C. Jard and T. Jéron. Bounded-memory algorithms for verification "on the fly". In *CAV'91*. LNCS 575, Springer Verlag, july 1–4 1991.

[Ker94] A. Kerbrat. *Méthodes symboliques pour la vérification de Processus Communicants : étude et mise en oeuvre*. PhD thesis, Université Joseph Fourier, November 1994.

[Koz83] D. Kozen. Results on the propositional μ-calculus. In *Theoretical Computer Science*. North-Holland, 1983.

[Kri96] J.P. Krimm. Une approche compositionnelle pour la virification de programmes LOTOS. Rapport de DEA, Grenoble, June 1996.

[KS90] P. Kanellakis and S. Smolka. CCS expressions, finite state processes and three problems of equivalence. *Information and Computation*, 86(1), May 1990.

[Lar92] K. Larsen. Efficient local correctness checking. In *Computer-Aided Verification, LNCS 630*, July 1992.

[LY92] D. Lee and M. Yanakakis. Online minimization of transition systems. In *24th ACM Symp. on the Theory of Computing, STOC'92, Vancouver, B.C.*, 1992.

[Mil80] R. Milner. A calculus of communication systems. In *LNCS 92*. Springer Verlag, 1980.

[Mil89a] R. Milner. *Communication and Concurrency*. C.A.R. Hoare, Editor. Prentice Hall International Series in Computer Science, 1989.

[Mil89b] R. Milner. A complete axiomatization for observational congruence of finite-state behaviours. *Information and Computation*, 81:227–247, 1989.

[Mou92] L. Mounier. *Méthodes de Vérification de Spécifications Comportementales : étude et mise en oeuvre*. PhD thesis, Université Joseph Fourier, January 1992.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: the temporal framework. In *The Correctness Problem in Computer Science*Γ London Γ1982. International Lecture Series in Computer Science Γ Academic Press.

[Niv79] M. Nivat. Sur la synchronisation des processus. *Revue Technique Thomson-CSF*Γ (11):899Ι919Γ1979.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theorical Computer Science*. Springer VerlagΓ1981. LNCS 104.

[Pha94] M. Phalippou. *Relations d'implantations et Hypothèses de Test sur des automates à entrées et sorties*. Thèse de doctoratΓ Université de BordeauxΓ FranceΓ1994.

[Pnu85] A. Pnueli. The temporal logic of concurrent programs. In *LNCS 194*. 1985. 12th ICALP.

[PT87] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput., No. 6*Γ16Γ1987.

[QS83] J.P. Queille and J. Sifakis. Fairness and related properties in transition systems. *Acta Informatica*Γ19Γ1983.

[RdS90] V. Roy and R. de Simone. Auto and Autograph. In R. KurshanΓ editorΓ *International Workshop on Computer Aided Verification, Rutgers*Γ juin 1990.

[RHR91] C. RatelΓ N. HalbwachsΓ and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New OrleansΓ December 1991.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *TCS*Γ18Γ1982.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computation*Γ 2(1)Γ june 1972.

[Tre95] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In A. Cavalli and S. BudkowskiΓ editorsΓ *8th Int. Workshop on Protocols Test Systems, Evry, France*Γ pages 461–476Γ September 1995.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *Proc. 8th Conference Computer-Aided Verification, CAV'96*Γ RutgersΓ NJΓ July 1996. LNCS 1102 Springer Verlag.

[vG90] R.J. van Glabbeek. The linear time - branching time spectrum. Technical Report CS-R9029Γ Centre for Mathematics and Computer ScienceΓ 1990.

[VL92] B. Vergauwen and J. Levi. A linear algorihm for solving fixed-point equations. In *CAAP, LNCS 581*Γ1992.

[VWL94] B. VergauwenΓ J. WaumanΓ and J. Levi. Efficient fixpoint computation. In *Static Analysis Symposium, SAS'94, LNCS 864*Γ1994.