Vérification symbolique pour les protocoles de communication

Dorel Marius Bozga

UNIVERSITE JOSEPH FOURIER - GRENOBLE I
SCIENCES ET GEOGRAPHIE

THESE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

présentée et soutenue publiquement par

Dorel Marius BOZGA

Le 17 décembre 1999

Titre :

VERIFICATION SYMBOLIQUE POUR LES PROTOCOLES DE COMMUNICATION

Directeur de thèse :
Jean-Claude Fernandez

COMPOSITION DU JURY :

Président	**Jacques Voiron**
Directeur	**Jean-Claude Fernandez**
Rapporteurs	**Gerard Holzmann**
	Axel van Lamsweerde
Examinateurs	**Roland Groz**
	Claude Jard
	Oded Maler

Remerciements

Je tiens à remercier d'abord l'ensemble des membres du jury :

Jacques Voiron, *Professeur à l'Université Joseph Fourier, pour m'avoir fait l'honneur de présider le jury de cette thèse ;*

Gerard Holzmann, *Senior Researcher à Bell-Labs, pour avoir eu la patience de lire soigneusement le manuscrit et pour les critiques constructives qu'il m'a fournit en retour ;*

Axel van Lamsweerde, *Professeur à l'Université Catholique de Louvain, pour avoir accepté de juger ce travail et pour l'intérêt qu'il lui a accordé ;*

Roland Groz, *Ingénieur en Chef des Télécommunications à France-Télécom, pour avoir accepter de participer à ce jury ;*

Claude Jard, *Directeur de Recherche à l'Irisa, pour avoir examiner soigneusement ce travail et pour ses précieux commentaires et suggestions ;*

Jean-Claude Fernandez, *Professeur à l'Université Joseph Fourier, qui a dirigé cette thèse et qui par ses nombreux conseils a permis son aboutissement. Je le remercie en particulier pour sa disponibilité, le soutien qu'il m'a témoigné en des nombreuses occasions, sa grande compétence et son exigence pour les parties formelles de ce travail ;*

Oded Maler, *Chargé de Recherche à Verimag, pour avoir suivi et contribué à l'élaboration d'une partie des résultats contenus dans ce document et pour avoir accepté d'en être examinateur ;*

Je tiens également à remercier l'ensemble des membres du laboratoire Verimag qui m'ont offert un cadre de travail privilégié. Plus particulièrement, je remercie :

Joseph Sifakis, *directeur du laboratoire, pour m'avoir accueilli dans son équipe et pour avoir toujours su donner des impulsions constructives à ce travail ;*

Susanne Graf *et* Laurent Mounier, *auprès desquels j'ai toujours trouvé des conseils pertinents et qui ont su créer une atmosphère accueillante dont je garderai un excellent souvenir ;*

Stravros Tripakis *et* Lucian Ghirvu, *c'était un plaisir de travailler avec eux ;*

Enfin, je tiens à remercier les membres de ma famille et mes amis qui ont du supporter mon indisponibilité durant ces trois années.

Table des matières

Introduction

Parmi les systèmes critiquesΓune catégorie est constituée des protocoles de télécommunication. Véritables systèmes nerveux des réseaux et des systèmes distribués de tout sorteΓ les protocoles connaissent aujourd'hui un développement spectaculaire. Leur complexité ne cesse de grandirΓdue d'une part aux fonctionnalités et aux applications de plus en plus nombreusesΓet d'autre partΓaux contraintes de fiabilité et de sûreté de plus en plus sévères. Ces raisons font que les protocoles constituent aujourd'hui un vrai défi autant pour la conception que pour la mise en service et l'exploitation.

Formalismes de description

L'utilisation des méthodes et des outils formels d'aide à la conception a été reconnue comme la seule approche en mesure de *garantir* le bon fonctionnement des protocoles. C'est pourquoiΓ pour les décrire de manière conciseΓcomplète et non-ambiguëΓont été conçus trois *langages de spécification formelle* : ESTELLE [ISO89ΓBD88]ΓLOTOS [ISO88ΓBB88] et SDL [IT94dΓ ST87]. Ces formalismesΓdésignés par le nom générique de langages FDT [1]Γont été définis et sont normalisés par des organismes internationaux de standardisation dans le domaine de télécommunicationsΓl'ISO [2] et l'ITU [3]. Assistés par des méthodologies de développement et des outils allant de simples éditeurs et analyseurs syntaxiques jusqu'à des vérificateurs et des générateurs de codeΓces langages constituent actuellement la base formelle de ce qu'on appelle l'ingénierie de protocoles.

Depuis leur définitionΓles langages FDT connaissent une évolution continueΓsoigneusement contrôlée par les comités de normalisationΓafin de faire face aux exigences manifestées par le développement des protocoles. Parmi ces exigencesΓl'introduction des *aspects non-fonctionnels* dans les spécifications tend à occuper actuellement une place centrale. Plus particulièrementΓdes caractéristiques quantitatives mesurant *la qualité des services* (QoS)Γ comme par exemple le temps de réponse ou les performances sont de plus en plus présentes dans les descriptions informelles des protocoles. Ces contraintes ne peuvent plus être ignorées d'autant qu'elles constituent des exigences très fortes sur le bon fonctionnement des protocoles de transfert à haut débit ou multimedia. Elles doivent être prises en compte dès

1. Formal Description Techniques
2. International Standards Organisation
3. International Telecommunication Union

les premières phases de conception, et nécessitent aussi d'être validées autant au niveau de la spécification qu'au niveau de la réalisation concrète du protocole.

Dans ces conditions, la définition des extensions des langages FDT, avec une sémantique adéquate du *temps*, est devenue un problème crucial, actuellement à l'ordre du jour des comités de normalisation et d'autres groupes de travail, tant académiques que industriels. Ce problème est néanmoins difficile à résoudre car, une telle sémantique doit prendre en compte non-seulement des considérations techniques sur *le temps*, mais aussi des considérations plus pragmatiques liées à son utilisation. Sur ce dernier point, on pense notamment à l'intégration d'une telle sémantique au sein des méthodes et des outils déjà existants autour de langages FDT.

Nous nous intéressons plus particulièrement au langage SDL [4] [IT94d], aujourd'hui à la base d'une *technologie industrielle* pour la spécification et la validation formelle des protocoles. Largement employé par les développeurs industriels, son succès incontestable est du à son modèle formel simple et intuitif à base d'automates communicants. Sans doute, la promotion constante faite par l'ITU et l'existence des outils commerciaux d'une bonne qualité ont contribué aussi à ce succès.

Un point de départ pour trouver la bonne sémantique du temps pour SDL sont les résultats issus de l'étude sur la sémantique des *systèmes temporisés* [Dil89, AD94] ainsi que le développement des méthodes de vérification pour ces systèmes [ACD93, HNSY94]. Axe de recherche important au sein du laboratoire VERIMAG, l'analyse de systèmes temporisés a donné lieu à des modèles très expressifs pour la description des aspects temporels [NRSV89, BST97] et à des méthodes et des outils efficaces d'analyse liés à ces modèles [Yov97].

Validation

L'évolution des protocoles et des langages FDT exige implicitement l'évolution et le développement des méthodes et des outils adéquats pour leur validation. Afin de pouvoir garantir le bon fonctionnement d'un protocole, deux méthodes complémentaires existent et sont utilisées depuis longtemps :

- La *vérification formelle* consiste à comparer la description d'un protocole (*la spécification*) avec la description des services attendus (*les propriétés*). Cette comparaison s'effectue selon une relation de satisfaction définie formellement.

  Il existe deux grandes approches pour la vérification. L'approche *déductive* introduite par [Hoa69], consiste à prouver les propriétés en utilisant des axiomes et des règles d'inférence associées à la spécification par une sémantique axiomatique. L'avantage de cette approche est sa généralité. En revanche, elle est au mieux semi-automatisable en utilisant des démonstrateurs automatiques ou des systèmes de réécriture.

  L'approche *basée sur des modèles* ou *model-checking* [QS82, CES83], que nous considérons plus particulièrement dans ce travail, consiste à montrer que l'ensemble des

4. Specification and Description Language

comportements décrits par une spécification est un modèleΓau sens logique du termeΓ pour les propriétés. Pour la vérification des protocolesΓle modèle habituel associé est un *système de transitions étiquetées*Γc'est-à-dire un automate dont les transitions entre états sont étiquetées par les actions du protocole. Le processus de vérification se décompose alors en deux phases : *compilation* d'un protocole vers un système de transitions étiquetéesΓpuis *vérification* sur ce système. L'intérêt pratique de cette approche est qu'elle est complètement automatisableΓsi le modèle du protocole est *fini*. Elle est par contre limitée par la taille des modèles générés.

– Le *test de conformité* [ISO92] consiste à démontrer la conformité d'une réalisation concrète du protocole (*l'implémentation*) à sa description formelle (*la spécification*) de référence. Là aussiΓla relation de conformité peut être définie formellement [Bri88].

  Dans la pratiqueΓla conformité est testée à l'aide d'une *suite de tests*Γconstruits à partir de la spécification et des propriétés que l'on veut tester. Pour les mêmes raisonsΓnous considérons l'approche basée sur les modèles. Le test est décomposé en deux phasesΓ complètement automatisables : *génération* d'une suite de tests à partir du modèle de la spécification et des propriétés à testerΓpuis *exécution* de tests sur l'implémentation à l'aide d'un testeur. Nous nous intéressons uniquement à la phase de génération de suite de testΓcomme le montre la figure I.1.

  HistoriquementΓle test a été la première et principale méthode de validation. Il reste une activité toujours très importanteΓétant données les difficultés majeures de conception et vérification formelle.

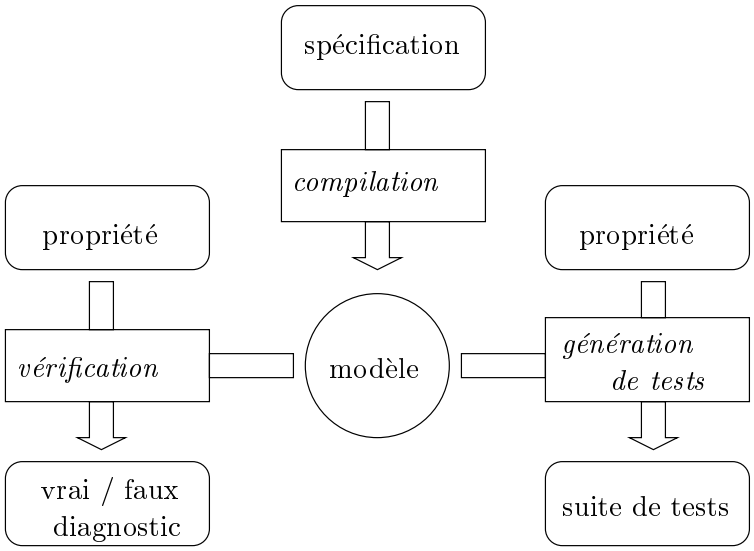

FIG. I.1 – *Validation basée sur les modèles.*

L'application de ces techniques au sein du laboratoire VERIMAG a donné lieu à des outils performants autant pour la vérification que pour le test. En particulierΓnous mentionnons

l'outil ALDEBARAN [Fer88ΓBFKM97] dédié à la vérification des propriétésΓainsi que l'outil TGV [FJJV97]Γréalisé en collaboration avec l'équipe Pampa de l'IrisaΓdédié à la génération de suites de test.

Explosion des états

Le problème lié aux méthodes de validation basée sur les modèles finis est la taille de ces modèles. Dès que les protocoles étudiés sont de complexité réalisteΓla taille du modèle correspondant devient rapidement prohibitive ; ce problème est connu sous le nom de *problème de l'explosion des états*. Les causes principales sont les suivantes :

- *la complexité des données* manipuléesΓpuisqu'une représentation énumérative associe normalement un état à chaque *valuation* des variables du protocole ;
- *l'asynchronisme des composantes*Γen sachant que l'exécution parallèle est représentée par *l'entrelacement* d'actions élémentaires.

L'explosion des états est un problème crucial pour les outils de validation travaillant sur un modèle complètement énuméré. Ces outils se trouvent de fait limités à la validation de modèles ayant environ quelques millions d'étatsΓce qui est largement insuffisant pour des cas réalistes.

Des nombreux travaux ont été effectués pour trouver des solutions à ce problème en ne représentant qu'une partie du modèle en mémoire [JJ89ΓFJJM92]Γen reduisant le nombre des chemins à explorer [GW91ΓVal92ΓPel94ΓGod96]Γen décomposant le système à vérifier [Pnu85ΓWin90ΓGS90b]Γen utilisant techniques d'abstraction [GL93ΓLon93ΓCGL94]Γou encore en utilisant des représentations symboliques compactes du modèle [CBM89ΓBCM+90ΓBry92ΓMcM93].

Dans ce contexteΓl'utilisation de techniques applicables pendant la phase de compilation et capables des fournir des informations sur le comportement des protocolesΓsans construire explicitement les modèlesΓest une approche intéressante. En particulierΓles *analyses de flot de données*Γou analyses statiquesΓsont un instrument très puissant pour calculer des informations sur la manière dont un programme utilisent ses données. Utilisées avec un succès remarquable dans l'optimisation de code (voir par exemple [ASU86ΓMuc97]) ces techniques s'avèrent indispensables pour la validation de protocoles qui comportent des données non-triviales. Face à l'explosion des étatsΓles techniques d'analyse statique constituent une source potentielle des solutionsΓqui méritent d'être explorées et expérimentées.

Objectifs de la thèse

Le travail que nous présentons a comme objectif l'étude et la mise en œuvre d'une représentation intermédiaire permettant la prise en compte des aspects temporels dans les protocoles tout en conservant les techniques de validation existantes. Nous nous intéressons plus particulièrement aux thèmes suivants :

Une représentation intermédiaire

Nous nous intéressons à la définition d'une représentation intermédiaire pour la spécification de protocoles quiΓd'une part doit être suffisamment expressive pour intégrer les aspects essentiels de ces systèmesΓet d'autre part être le mieux adaptée pour la validation formelle de propriétés.

Du point de vue expressivitéΓà part des aspects fonctionnels existants dans les langages FDTΓ on s'intéresse à décrire de manière non-ambiguë différents types de contraintes temporelles. Concernant la validationΓon cherche une représentation située en amont de l'explosion des états où l'application des techniques d'analyse statique soit encore possible et réaliste. De plusΓon cherche une sémantique opérationnelle bien définieΓnous permettant une compilation efficace vers des modèles.

Des nombreux formalismes existentΓcomme par exempleΓles algèbres de processus [Mil80Γ Hoa84]Γavec ou sans valeursΓles réseaux de PetriΓles automates communicantsΓetc. Le formalisme que nous avons choisi pour cette étude est une extension des *automates temporisés* avec urgences [BST97]Γen ajoutant la communication synchrone par rendez-vous et asynchrone par files d'attentes. Il satisfait toutes les exigences imposées et de plusΓc'est un modèle similaire à ceux utilisés par des outils de validation existants.

Analyse statique pour la validation

Nous nous sommes fixé comme deuxième objectif l'étude et l'évaluation de quelques unes des techniques d'analyse statiqueΓvues comme des solutions potentielles à l'explosion des états. Les exigences sont l'efficacité sur des exemples de taille réaliste et l'utilité des résultats obtenus pour la validation.

Parmi les techniques d'analyses de flot de données utilisées dans l'optimisation de code [ASU86Γ Muc97] quelques unes nous ont semblé intéressantes. L'analyse des *variables actives* [ASU86] constitue un moyen très fin et efficace pour calculer l'utilité des données dans un programme. L'application des résultats obtenus peut donner lieu à des réductions spectaculaires sur la taille de modèles générés. De la même manièreΓla *propagation de constantes* [ASU86ΓWZ91] ou des *invariants* [BBM97ΓBL99] nous permet d'augmenter les connaissances sur le fonctionnement d'un protocoleΓet aussi de réduire encore la représentation de son modèle.

FinalementΓon s'intéresse aussi au *calcul d'abstractions*Γune technique indispensable pour faire face à la complexité de protocoles. En particulierΓdes techniques statiques de *slicing*Γ dont une synthèse peut être trouvée dans [Tip94]Γsemblent bien adaptées à nos exigences. Employées principalement pour la mise au point de programmes (debugging)Γelles permettent l'extraction automatique de tranches d'un programme représentatives pour la validation d'une propriété fixée. Leur coût est extrêmement faible.

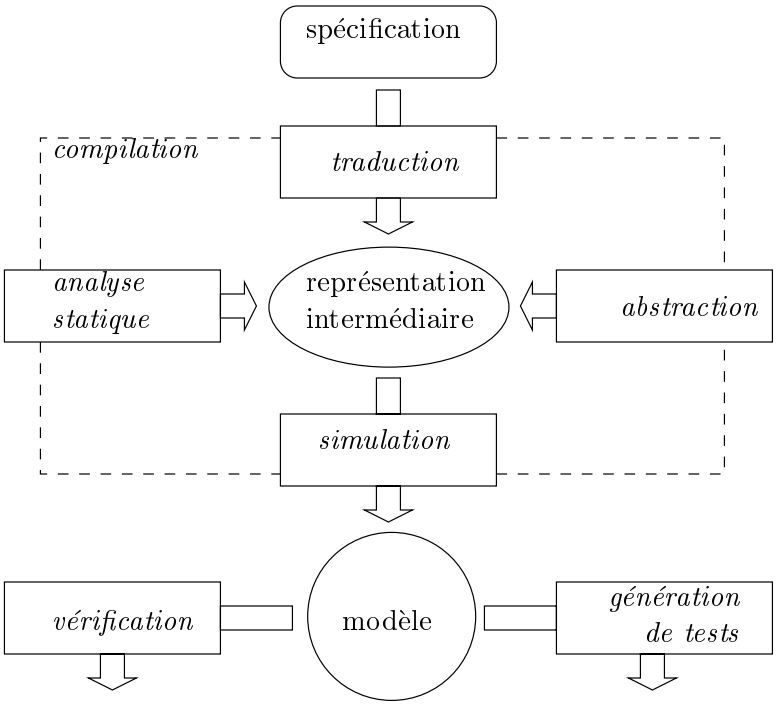

FIG. I.2 – *Nos objectifs.*

Un environnement de validation

Notre dernier objectif est la conception d'un environnement de validation autour de la représentation intermédiaire.

En généralΠl'ensemble des outils d'aide à la conception de protocoles est relativement divisée. D'une partΓles outils commerciaux [ABΓVer] qui respectent les normes FDTΓsont employés par une large catégorie d'utilisateurs mais en revancheΓils offrent des techniques assez limitées pour la validation. D'autre partΓles outils académiques [FGK+96ΓHol91ΓYov97ΓFJJV97ΓBLO98] offrent des meilleures performances pour la validation. Par contre ils sont souvent développés autour de formalismes dont les liens avec les normes ne sont pas toujours claires et n'ont pas beaucoup d'utilisateurs.

Notre environnement doit assurer un lien entre ces deux catégories d'outils. Il doit permettre de spécifier des protocoles complexes (écrits notamment en SDL) sur lesquels nous pouvons appliquer des techniques et algorithmes de validation performants. En particulierΓafin de faciliter la connexion des outils existants ou encore d'en développer d'autresΠl'environnement devrait fournir à l'utilisateur plusieurs ouverturesΓà différents niveaux de représentation de protocoles. D'abordΓau niveau représentation intermédiaire nous allons connecter des outils d'analyse statique et définir des connexions de niveau langage avec d'autres formalismes. EnsuiteΓau niveau du modèle sémantique i.eΓles systèmes de transitions étiquetéesΓnous allons connecter des outils de vérification et de génération de suites de test.

Plan du document

La première partie décrit les formalismes de description que nous avons examiné pour la spécification de protocoles. Nous nous intéressons principalement à la sémantique dynamiqueΓ qui permet d'associer à la description d'un protocole l'ensemble de ses comportements. La sémantique statiqueΓconcernant en général la description des types de données est ignorée. En faitΓles types de données sont une composante orthogonaleΓqui peut être considérée comme un paramètre pour la description du comportement.

Le chapitre 1 présente les principaux aspects syntaxiques et sémantiques du langage SDLΓ basés sur la Recommandation Z.100 de l'ITU [IT94d]. Nous discutons quelques uns des problèmes liés à la sémantique dynamique actuelle du langage SDLΓainsi que quelques unes des sémantiques alternatives proposées au cours du temps.

Le chapitre 2 présente la représentation intermédiaire IFΓsa syntaxe et sa sémantique opérationnelle en termes de systèmes de transitions étiquetées. IntuitivementΓcette représentation est construite à base d'automates temporisés communiquant soit par files d'attenteΓsoit par rendez-vous et représente le meilleur compromis que nous avons trouvé entre l'expressivité de description le support de validation automatique.

Le chapitre 3 présente la traduction d'un sous ensemble du langage SDL vers la représentation intermédiaire IF. Cette traduction expliciteΓà l'aide de primitives de IFΓune grande partie des constructions syntaxiques de SDLΓtout en préservant la sémantique de la norme Z.100. Des choix originaux sont proposés au niveau de la sémantique du tempsΓafin d'obtenir une représentation plus adaptée à la validation.

La deuxième partie décrit les techniques d'analyse et les outils que nous avons développés autour de SDL et IF.

Le chapitre 4 présente l'adaptation de quelques techniques d'analyse statique aux programmes IF. Nous nous sommes intéressé de plus près aux techniques générales issues du domaine de l'optimisation de codeΓcomme par exemple l'analyse de variables actives et la propagation de constantesΓmais aussi aux techniques plus orientées vers la validation comme la génération d'invariants structurels et le calcul d'abstractions.

Le chapitre 5 présente des techniques pour la simulation exhaustive de programmes IF. Plusieurs modes de simulation complémentaires sont définis : énumérativeΓmixte et symbolique. Ils diffèrent par leur manière de représenter les états ainsi que par la manière de calculer les transitions entre ces états. Pour chacun d'euxΓquelques principes généraux d'optimisation basés sur des résultats d'analyse statique sont aussi présentés.

Le chapitre 6 présente l'environnement de validation existant autour de IF. Cet environnement est basé sur l'intégration d'un nombre d'outils industriels et académiquesΓet couvre la plupart des techniques de validation employées couramment pour la validation de programmesΓqui sont la simulation interactive ou aléatoireΓla vérification formelle de propriétésΓet la génération de séquences de test.

La troisième partie présente quelques études de cas validées à l'aide de nos outils ainsi que des conclusions et des perspectives d'évolution futures.

Le chapitre 7 présente des résultats concrets obtenus en appliquant la technologie de validation existante autour de IF sur trois protocoles de communication. Ces résultats confirment le bien fondé de notre approche en relevant la puissance d'expression de notre modèle intermédiaire, l'importance des analyses statiques en amont de la validation, et l'intérêt d'avoir plusieurs techniques de validation au sein d'un environnement.

Finalement, l'annexe A présente la syntaxe et la sémantique du μ-calcul modal arborescent utilisé pour la description de propriétés. L'annexe B présente la syntaxe graphique de SDL.

Notations

Nous présentons les notations utilisées dans la suite du document.

Ensembles

Soient $E\Gamma E_1\Gamma$et E_2 des ensembles. Nous utilisons pour les opérateurs ensemblistes les notations suivantes :

- $\emptyset$ dénote *l'ensemble vide* ;
- $E_1 \cup E_2$ dénote *l'union* de E_1 et E_2 ;
- $E_1 \cap E_2$ dénote *l'intersection* de E_1 et E_2 ;
- $E_1 \setminus E_2$ dénote *la différence* de E_1 et E_2 ;
- $E_1 \subseteq E_2$ dénote *l'inclusion* de E_1 et E_2 ;
- $E_1 \times E_2$ dénote *le produit cartésien* de E_1 et E_2 ;
- $|E|$ dénote *le cardinal* de E ;
- $\mathcal{P}(E)$ dénote *l'ensemble des parties* de E ;
- E^* dénote *l'ensemble des séquences* ayant zéro ou plusieurs éléments de E ;

Nous considérons les ensembles particuliers :

- $\mathbb{B}$ est l'ensemble des valeurs booléennes ;
- $\mathbb{Z}$ ($\mathbb{Z}^+$) est l'ensemble des nombres *entiers* (*positifs ou nul*) ;
- $\mathbb{R}$ ($\mathbb{R}^+$) est l'ensemble des nombres *réels* (*positifs ou nul*) ;

Relations binaires

Soit R une relation binaire définie sur un ensemble E. Les propriétés de R auxquelles on s'intéresse sont :

- *réfléxivité* : $\forall x \in E \ (x,x) \in R$
- *symétrie* : $\forall x,y \in E \ (x,y) \in R \ \Rightarrow \ (y,x) \in R$
- *antisymétrie* : $\forall x,y \in E \ (x,y) \in R \ \wedge \ (y,x) \in R \ \Rightarrow \ x = y$
- *transitivité* : $\forall x,y,z \in E \ (x,y) \in R \ \wedge \ (y,z) \in R \ \Rightarrow \ (x,z) \in R$

Nous considérons plus particulièrement les relations binaires suivantes :

- *préordre* : relation réflexive et transitive ;
- *ordre partiel* : relation réflexiveΓantisymétrique et transitive ;
- *ordre total* : ordre partiel et complet $\forall x,y \in E \ (x,y) \in R \ \vee \ (y,x) \in R$
- *équivalence* : relation réflexiveΓsymétrique et transitive ;

Fonctions

Une *fonction* est une application $f : E_1 \rightarrow E_2$ qui associe à chaque élément $x \in E_1$ un élément $y \in E_2$. E_1 s'appelle *le domaine de définition* et E_2 s'appelle le *domaine de valeurs* de f.

Si $f : E_1 \rightarrow E_2$ et $x_i \in E_1 \Gamma y_i \in E_2$ pour $1 \leq i \leq n$ nous notons par $f[y_1/x_1, \ldots \ y_n/x_n]$ la fonction $g : E_1 \rightarrow E_2$ définie comme suit :

$$g(x) = \begin{cases} y_i & \text{si } x = x_i \\ f(x) & \text{sinon} \end{cases}$$

Treillis

Soit un ensemble E muni d'un relation d'ordre partiel $\sqsubseteq$ et soit $E' \subseteq E$.

- *les majorants* de E' : $Maj(E') = \{x \in E \mid \forall x' \in E' \ x' \sqsubseteq x\}$
- *le plus petit des majorants de* E' (s'il existe) :
  $\sqcup E' \in Maj(E')$ tel que $\forall x \in Maj(E') \quad \sqcup E' \sqsubseteq x$
- *les minorants* de E' : $Min(E') = \{x \in E \mid \forall x' \in E' \ x \sqsubseteq x'\}$

- *le plus grand des minorants de* E' (s'il existe) :

  $\sqcap E' \in Min(E')$ tel que $\forall x \in Min(E') \;\; x \sqsubseteq \sqcap E'$

Le couple $(E, \sqsubseteq)$ est un *treillis* si le plus petit majorant $\sqcup E'$ et le plus grand minorant $\sqcap E'$ existent pour tout partie finie $E' \subseteq E$. Le couple $(E, \sqsubseteq)$ est un *treillis complet* si le plus petit majorant $\sqcup E'$ et le plus grand minorant $\sqcap E'$ existent pour tout partieΓfinie et infinieΓ $E' \subseteq E$.

Un treillis complet satisfait *la propriété de chaînes croissantes* si toute chaîne $(x_i)_{i \geq 0} \in E$ strictement croissante est finie. La longueur maximale de chaînes strictement croissantes sera nommée *la profondeur* du treillis.

Soit $(E, \sqsubseteq)$ un treillis complet et $f : E \to E$ une fonction :

- $x \in E$ est un *point fixe* de f : $f(x) = x$
- f est dite *monotone croissante* : $\forall x, y \in E \; x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$
- f est dite $\sqcup$-*continue* :

  $\forall \; (x_i)_{i \geq 0}$ chaîne croissante de $E \; f(\sqcup\{x_i \mid i \geq 0\}) = \sqcup\{f(x_i) \mid i \geq 0\}$
- f est dite $\sqcap$-*continue* :

  $\forall \; (x_i)_{i \geq 0}$ chaîne décroissante de $E \; f(\sqcap\{x_i \mid i \geq 0\}) = \sqcap\{f(x_i) \mid i \geq 0\}$

Soit $(E, \sqsubseteq)$ un treillis complet et $f : E \to E$ une fonctionΓon a les propositions suivantes :

1. si f est $\sqcup$-continue (ou $\sqcap$-continue) alors elle est monotone ;
2. si f est monotone et si le treillis satisfait la propriété de chaînes croissantes alors f est $\sqcup$-continue (ou $\sqcap$-continue) ;
3. (Tarski) *Le plus grand* (νf) et *le plus petit* (μf) *point fixe* d'une fonction f monotone existent et ils sont uniques ; les éléments $\nu f = \sqcup\{x \mid x \sqsubseteq f(x)\}$ et $\mu f = \sqcap\{x \mid f(x) \sqsubseteq x\}$ satisfont :
   - $f(\nu f) = \nu f$ et $\forall y \; f(y) = y \;\Rightarrow\; y \sqsubseteq \nu f$
   - $f(\mu f) = \mu f$ et $\forall y \; f(y) = y \;\Rightarrow\; \mu f \sqsubseteq y$
4. (Kleene) si f est $\sqcup$-continue (respectivement $\sqcap$-continue) alors :
   - $\mu f = \sqcup\{f^{(i)}(\sqcap E) \mid i \geq 0\}$ (respectivement $\nu f = \sqcap\{f^{(i)}(\sqcup E) \mid i \geq 0\}$).

Syntaxe

Pour les définitions syntaxiques de SDL et IF on utilise le méta-langage défini comme suit :

- les symboles terminaux sont notés en caractères **gras** ;

- les symboles non-terminaux sont notés en caractères *italiques* ;
- le méta-symbole "::=" sépare la partie gauche de la partie droite d'une production ;
- le méta-symbole "|" défini le choix entre ses deux parties droites ;
- le méta-symbole "[ ]" dénote zéro ou une occurrence de l'élément qu'il encadre ;
- la concaténation est notée par juxtaposition ;
- ϵ dénote la séquence vide ;

Nous utilisons aussi des notations étendues :

- la notation A_1 .. A_n dénote la concaténation des n éléments A_1 A_2 ... A_n;
- la notation A_1 , .. A_n dénote la concaténation des n éléments A_1 , A_2 , ... , A_n.

Pour la description du comportement de méta-processus SDL on utilise le méta-langage défini comme suit :

- les méta-opérateurs sont notés en caractères *italiques soulignés* ;
- les méta-processus et les méta-signaux sont notés en caractères sans-serif ;
- les comportements élémentaires sont :
  - *output* s *to* p : l'émission synchrone du signal s vers le processus p ;
  - *input* s *from* p : la réception synchrone du signal s en provenance du processus p ;
  - *i* : une action interne ;
- le méta-opérateur "[]" dénote le choix arbitraire entre deux comportements ;
- le méta-opérateur ";" dénote la composition séquentielle de deux comportements ;
- le méta-opérateur *cycle* *endcycle* dénote l'exécution d'un nombre arbitraire de fois du comportement qu'il encadre.

Première partie

Langages

Chapitre 1

SDL

SDL (*Specification and Description Language*) est un langage de spécification formelle pour les protocoles de communicationΓdéveloppé et standardisé par ITU-T (*Telecommunications Standardisation sector*Γ ancien CCITT *Comité Consultatif International Télégraphique et Téléphonique*).

SDL fait l'objet d'une révision de l'ITU-T tous les 4 ans. La première version a été disponible en 1976. Après les deux évolutions successives de 1980 et 1984Γ SDL atteint en 1988 une forme stableΓdécrite par la Recommandation Z.100 de CCITT [IT94d]Γdont une sémantique formelle est fournie comme annexe [IT94aΠT94b]. Les versions ultérieures de 1992 et 1996 ont concerné notamment l'introduction des concepts orienté-objet. La version de l'année 2000Γ actuellement en cours de standardisationΓconstitue un grand pas vers un rapprochement avec le langage de spécification UML (*Unified Modelling Language*).

Depuis son apparitionΓSDL connaît un grand succès dans le monde industriel et notamment dans le monde des télécommunications. Les raisons sont multiples. D'une partΓ SDL a un modèle formel relativement simple : toute spécification SDL décrit un système réactif composé d'un ensemble de processus concurrentsΓcommuniquant les uns avec les autres et avec leur environnement. Les processus sont des machines d'états finis étendues avec des variables. Les spécifications SDL admettent deux syntaxes équivalentes : la syntaxe textuelle SDL/PRΓ et la syntaxe graphique SDL/GR. La plus utilisée est la syntaxe graphique. Basée sur des symboles simplesΓc'est elle qui a fait du SDL un langage largement accepté par les industrielsΓ au détriment d'autres formalismes plus arides comme LOTOS [ISO88] ou ESTELLE [ISO89]. CependantΓpour la concision et la clarté de la présentationΓnous allons utiliser par la suite le format textuel SDL/PR. FinalementΓil existe des méthodologies et des outils commerciaux pour assister le développement basé sur SDL. C'est une raison aussi importante quiΓà part la standardisationΓdonne confiance aux utilisateurs industriels.

Ce chapitre présente les principaux aspects syntaxiques et sémantiques du langage SDL. Dans une première partie nous présentons une syntaxe abstraite pour le langage SDL/PR. Cette syntaxe est construite à partir de la syntaxe abstraite définie par la norme Z.100. Elle est limitée aux parties nécessaires pour la compréhension de la sémantique dynamique de

SDL. Ensuite nous présentons une synthèse sur la sémantique dynamique du SDL fournie par l'annexe F de la norme Z.100 de l'ITU. Il s'agit d'une présentation relativement informelle dont le rôle est de donner une vue d'ensemble sur les quelques 500 pages de l'annexe F de la norme. Nous finissons par une discussion sur les problèmes liés à cette sémantique et nous présentons brièvement quelques autres solutions proposées dans la littérature.

1.1 Syntaxe abstraite

1.1.1 Structure

Système

Un système est constitué d'un ensemble de blocs connectés par des canaux. Les canaux assurent une communication point à point entre deux blocs ou entre un bloc et l'environnement du système. Ils transportent des signaux avec des paramètres qui sont des valeurs appartenant aux domaines de types des données.

SyntaxiquementΓla définition d'un système contient des définitions globales de blocsΓde canauxΓde signaux et de types de données globauxΓutilisés soit pour la communicationΓsoit à l'intérieur des blocs.

system ::=
system *system-id* **;**
system-component$_1$ .. *system-component*$_m$
endsystem **;**

system-component ::=
block | *channel* | *signal* | *type*

Bloc

Un bloc est constitué d'un ensemble de processus connectés par des routes. Contrairement aux canaux qui connectent des blocsΓles routes assurent la communication point à point entre deux processus ou entre un processus et l'environnement du bloc. NormalementΓdans le deuxième casΓces routes sont connectées aux canaux définis à l'extérieur du blocΓpar des points de connexion.

De plusΓun bloc peut contenir une sous-structure : dans ce cas il est encore décomposé en plusieurs sous-blocs connectés par des canaux. Comme dans le systèmeΓles canaux relient soit deux blocsΓsoit un bloc et son environnementΓà travers des points de connexion aux canaux extérieurs au bloc. On peut aussi définir des nouveaux types et signauxΓlocalementΓ à l'intérieur d'un bloc.

block ::=
block *block-id* **;**

block-component$_1$ .. *block-component*$_m$
channel-to-route$_1$ .. *channel-to-route*$_n$
[*substructure*]
endblock ;

block-component ::=
process | *signalroute* | *signal* | *type*

substructure ::=
substructure *substructure-id* **;**
system-component$_1$ .. *system-component*$_m$
channel-to-channel$_1$ .. *channel-to-channel*$_n$
endsubstructure ;

Processus

Les processus sont les entités fonctionnelles élémentaires d'un système SDL. GénéralementΓ un processus est défini soit par un automate d'états finis étenduΓsoit par une composition parallèle de tels automatesΓnommés services.

Pendant l'exécution d'un système SDLΓ plusieurs instances (copies) du même processus peuvent exister et s'exécuter en même temps. Les instances peuvent être créées et arrêtées dynamiquement ; leur nombre est contrôlé par deux paramètres présents dans la définition du processus. Ces deux paramètres sont le nombre initial d'instances créées au démarrage du système et le nombre maximal d'instances vivantes autorisé à tout moment.

Chaque instance a une mémoire locale constituée respectivement d'un ensemble de variables locales typées et d'un ensemble de variables paramètresΓces dernières étant initialisées lors de la création de l'instance. De plusΓune instance peut consulter des variables définies et exportées par d'autres instances. AussiΓchaque instance possède implicitement une file d'attente qui garde les signaux reçus et pas encore traités. FinalementΓau niveau des processus on peut définir et utiliser des temporisationsΓpour modéliser notamment des contraintes temporelles simples de type expiration (*timeout*).

Si un processus est défini par une composition de servicesΓles services partagent pendant leur exécution les variablesΓles temporisations et la file d'attente du processus. De plusΓles services peuvent communiquer par échanges des signaux à travers des routesΓde la même manière que les processus à l'intérieur des blocs. Là aussiΓles routes relient point à point deux services ou un service et l'environnement : c'est le cas où ils sont normalement connectés aux routes extérieures du processus.

FinalementΓpour mieux structurer la description du fonctionnement des processus ou des servicesΓon peut définir et utiliser des procédures.

process ::=
process *process-id* (*init* , *max*);
fpar *parameter*$_1$, .. *parameter*$_m$;

process-component$_1$ .. *process-component*$_n$
process-behavior
endprocess ;

process-component ::=
procedure | *signal* | *type* | *variable* | *view* | *timer*

process-behavior ::=
state-graph | *service-decomposition*

service-decomposition ::=
service$_1$ .. *service*$_m$
signalroute$_1$ .. *signalroute*$_n$
route-to-route$_1$ .. *route-to-route*$_p$

Service

Afin de simplifier sa descriptionΓun processus peut être défini par une composition parallèle de plusieurs services. AinsiΓun service décrit un comportement partiel du processusΓses réactions à un sous-ensemble d'événements possibles. Les services s'exécutent un à la foisΓ par entrelacementΓet non pas de manière concurrente. Ils peuvent communiquer entre eux soit par envoi de signaux à travers des routesΓsoit par des variables partagéesΓdéfinies au niveau du processus.

Un service peut définir ses propres variables locales et temporisationsΓvoir aussi des types de données ou des procédures locales. Son fonctionnement est décrit par un automate d'états finis étendu.

service ::=
service *service-id* **;**
service-component$_1$ .. *service-component*$_m$
state-graph
endservice ;

service-component ::=
procedure | *type* | *variable* | *view* | *timer*

Procédure

Comme dans les langages de programmation traditionnelsΓles procédures sont des mécanismes d'abstraction du fonctionnement des processus ou des services. Elles peuvent avoir des paramètres formelsΓpassés soit par valeur soit par référence. De plusΓon peut définir dans une procédure des variablesΓdes types de données locaux ou d'autres procédures. Leur fonc-

tionnement est défini aussi par des automates d'états finis étendusΓopérant sur les variables locales et sur les paramètres.

procedure::=
    procedure *procedure-id*;
        fpar $mode_1$ $parameter_1$, .. $mode_m$ $parameter_m$;
        $procedure\text{-}component_1$ .. $procedure\text{-}component_n$
        state-graph
    endprocedure;

procedure-component::=
    procedure | *type* | *variable*

1.1.2 Communication

Signal

Les signaux sont les objets transportés d'un processus à l'autreΓà travers les canaux et les routes du système. Ils sont transportés d'une manière *asynchrone*: l'émetteur n'est pas bloqué pendant l'émission par la disponibilité d'un récepteur. L'émission a toujours lieu et l'émetteur continue son exécution sans attendre. La déclaration d'un signal comporte son nom et la liste des types des paramètres qui sont transportés.

signal::=
    signal *signal-id* ($type\text{-}id_1$, .. $type\text{-}id_m$);

Canal

Un canal assure une connexion point à point entre deux blocs ou entre un bloc et son environnement. NormalementΓle délai de transmission des signaux dans un canal est arbitraireΓ sauf s'il est défini explicitement avec un délai nulΓau cas où la transmission est immédiate. Un canal est paramétré par l'ensemble de signaux qui y transitent.

channel::=
    channel *channel-id* [**nodelay**]
        from *channel-endpoint*
        to *channel-endpoint*
        with $signal\text{-}id_1$, .. $signal\text{-}id_m$
    endchannel;

channel-endpoint::=
    block-id | **env**

Route

Les routes connectent soit deux processus (services) soit un processus (service) et son environnement. Contrairement aux canauxΓles routes n'ont pas de délai de transmissionΓainsi un signal entrant une route est délivré immédiatement au destinataire. De la même manière que les canauxΓles routes sont aussi paramétrées par l'ensemble des signaux qui transitent.

signalroute ::=
signalroute *signalroute-id*
from *signalroute-endpoint*
to *signalroute-endpoint*
with *signal-id*$_1$, .. *signal-id*$_m$;

signalroute-endpoint ::=
process-id | *service-id* | **env**

Connexions

Etant donné un bloc (ou processusΓou service)Γun point de connexion relie normalement un canal (ou route) défini à l'intérieur et dirigé vers l'environnement à un canal (ou route) défini à l'extérieur du bloc. Autrement ditΓces points assurent la connexion du réseau de communication interne de chaque entité et son réseau de communication externe.

channel-to-channel ::=
connect *channel-id* **and** *channel-id*;

channel-to-route ::=
connect *channel-id* **and** *signalroute-id*;

route-to-route ::=
connect *signalroute-id* **and** *signalroute-id*;

1.1.3 Contrôle

Graphe

Le fonctionnement des processusΓservices ou procédures est décrit par des automates d'états finis étendus. Des tels automates sont composés naturellement d'un ensemble fini d'étatsΓ dont un état initialΓet un ensemble fini de transitions entre ces états.

state-graph ::=
start; *transition*
state$_1$
..
state$_m$

Etat

Pour un état donnéΓles entrées (*input*) sont les événements auxquels le processus doit réagir et qui déclenchent les transitions correspondantes. Dans la plupart des casΓles entrées concernent les signaux reçus et présents dans la file d'attente du processus (voir ci-après).

Si dans un étatΓle signal de tête de la file ne peut être consomméΓil sera silencieusement détruit et le processus restera dans le même état. Cette règle assure une certaine réactivité de la part d'un processus SDLΓqui ne sera jamais bloqué à cause d'un signal imprévu arrivé dans sa file. Cette règleΓrelativement forteΓpeut être contournée à l'aide de la clause *save* permettant la définition d'un ensemble de signaux à sauvegarder pour l'état. Si un tel signal arrive en têteΓon passe au suivant dans la fileΓsans le détruireΓpour trouver un autre qui déclenche une transition. Ce signal restera en tête et sera à nouveau analysé lorsque on changera l'état.

state::=
state *state-id*;
$input_1$ $transition_1$
..
$input_m$ $transition_m$
save $sigtim\text{-}id_1$, .. $sigtim\text{-}id_n$;
endstate;

Input

GénéralementΓune entrée (**input**) précise le signal attenduΓdes variables pour stocker ses paramètres et éventuellement une condition de franchissement (**provided**). IntuitivementΓ si cette condition est vraie dans l'état courant et si ce signal se trouve en tête de la fileΓalors le processus peut le consommer et ensuite exécuter la transition correspondante.

D'autres versions plus élaborées des entrées sont possibles. D'abordΓon peut définir des *entrées prioritaires* (**priority input**)Γqui imposent la consommation prioritaire de certains signauxΓquelque soit leur position dans la file. On peut définir des *entrées spontanées* (**input none**) ou des *entrées continues* (**provided**)Γqui ne dépendent plus d'un signal de la file et qui permettent ainsi le déclenchement arbitraire des transitions. La différence entre ces deux est que les entrées continues ne sont consommables que si la file est videΓet éventuellement en fonction d'une priorité explicite.

input::=
input *sigtim-id* ($variable\text{-}id_1$, .. $variable\text{-}id_m$) **provided** *expression*;
| **priority-input** *sigtim-id* ($variable\text{-}id_1$, .. $variable\text{-}id_m$);
| **input none provided** *expression*;
| **provided** *expression*; **priority** *value*;

sigtim-id::=
signal-id | *timer-id*

Transition

SyntaxiquementΓune transition est une séquence d'actions élémentaires qui se termine soit par une action de terminaisonΓsoit par une action de branchement. Les actions peuvent être étiquetées. Les étiquettes assurent des points d'entrée dans la séquenceΓet sont normalement utilisées par des actions de transfert du contrôleΓde type *join*.

transition ::=
    [$label_1$] $action_1$
    ..
    [$label_n$] $action_n$
    [$label_e$] *end-action*

end-action ::=
    decision | *terminator*

Action

Les actions élémentaires exécutées par les transitions sont respectivement des émissions de signaux (**output**)Γdes créations de nouvelles instances de processusΓdes armements (**set**) ou désarmements (**reset**) de temporisationsΓdes affectations à des variables locales (**task**)Γ ou des appels de procédures (**call**).

Il faut noter que l'émission de signaux ne repose pas sur l'indication explicite d'un processus destinataire. Dans tous les casΓle signal est délivré au réseau de communicationΓet cherche dynamiquement son destinataireΓétant donné les contraintes imposées par le profil des canaux et des routes (qui peuvent transporter uniquement certains signaux) et ensuite les informations de routage supplémentaires précisées dans l'émissionΓles clauses *to* et *via*. Si aucun destinataire satisfaisant toutes ces contraintes n'est trouvéΓle signal est silencieusement détruit. Par contreΓsi plusieurs destinataires existentΓle signal est délivré au hasardΓ à exactement l'un d'entre eux.

action ::=
    output *signal-id* ($expression_1$, .. $expression_m$)
        [**to** *expression*] **via** $path\text{-}id_1$, .. $path\text{-}id_n$;
| **create** *process-id* ($expression_1$, .. $expression_m$) ;
| **set** *expression* , *timer-id* ($expression_1$, .. $expression_m$) ;
| **reset** *timer-id* ($expression_1$, .. $expression_m$) ;
| **task** *variable-id* **:=** *expression* ;
| **call** *procedure-id* ($expression_1$, .. $expression_m$) ;

path-id ::=
    channel-id | *signalroute-id*

Décision

Les décisions sont des actions un peu particulières. Elles assurent le branchement du flot de contrôle à l'intérieur des transitions SDL. AinsiΓune transition qui démarre dans un état peut se brancher à l'aide d'une décisionΓcontinuer de façon différente dans chacune des branches et finalement se terminer dans des états SDL différents. On distingue les décisions formelles (**decision** *expression*) oùΓen fonction de la valeur courante d'une expression on choisit dynamiquement la branche correspondanteΓet des décisions informelles (**decision any**) où le choix de la branche à suivre est non-déterministe.

decision ::=
    decision *decision-question* **;**
        ([$expression_1$] **):** $transition_1$
        ..
        ([$expression_m$] **):** $transition_m$
    enddecision ;

decision-question ::=
    expression | **any**

Terminaison

Toute séquence syntaxique d'actions élémentaires dans une transition est terminée soit par une décisionΓsoit par une action de transfert du contrôle. NormalementΓcela peut être soit le passage dans un état SDL (**nextstate**)Γsoit le transfert vers une action étiquetée dans une transition (**join**)Γsoit le retour d'une procédure vers l'entité appelante (**return**)Γou même l'arrêt de l'entité en cours d'exécutionΓprocessus ou service (**stop**).

terminator ::=
    nextstate *state-id* **;**
| **join** *label* **;**
| **return ;**
| **stop ;**

1.1.4 Temporisation

Les temporisations sont utilisées en SDL pour modéliser des contraintes temporelles simplesΓ comme par exempleΓle fait de ne pas attendre indéfiniment l'arrivée d'un certain signalΓou d'introduire un délai entre deux opérationsΓetc.

Plus précisémentΓune temporisation peut être armée à une valeur positive du temps qui indique normalement un instant dans le futurΓpar rapport à l'instant courant. Dès que ce moment arriveΓla temporisation expire et un signal implicite associé à elle est mis dans la

file d'attente du processus propriétaire. AinsiΓle processus sera averti de la progression du temps lors de la consommation du signal d'expiration. Un certain nombre de paramètres valeurs supplémentaires peuvent être associés au moment de l'armementΓet ils seront transmis comme paramètres du signal d'expiration correspondant.

A tout momentΓil est possible de tester l'activité d'une temporisationΓvoir si elle est armée et n'a pas encore expiré. De plusΓon peut la désarmer ou la réarmer à une autre valeur.

timer ::=
timer *timer-id* (*type-id*$_1$, .. *type-id*$_m$);

1.1.5 Données

Variable

La notion de variable en SDL est la même que celle des langages impératifs de type Pascal ou C. Une variable à un nom uniqueΓqui permet son identification dans le programme et un type. Sa durée de vie est égale à celle de l'entité où elle a été déclarée (processusΓserviceΓ procédure). La portée est aussi limitée à cette entitéΓà l'exception des variables définies dans un processus décomposé en plusieurs servicesΓqui sont aussi visibles dans ses services. CependantΓla portée d'une variable peut être globaleΓsi l'on déclare comme *revealed*. Elle est exportée et pourra ensuite être consultée par d'autres entités.

variable ::=
dcl [**revealed**] *variable-id type-id*;

View

Pour qu'une variable *revealed*Γà porté globaleΓsoit consultée par une autre entité SDLΓ celle-ci devrait contenir une déclaration *viewed*Γlui assurant la visibilité syntaxique de cette variable. Ce mécanisme permet de contraindre syntaxiquement l'accès aux variables locales exportéesΓuniquement aux entités qui sont intéressées. De plusΓil faut noter que l'accès est permis uniquement en lectureΓainsi on peut juste consulterΓsans modifierΓla valeur d'une variable *viewed*.

view ::=
viewed *variable-id type-id*;

Paramètres

Les paramètres sont utilisés dans les définitions des processus et dans celles des procédures. Dans le premier casΓils sont considérés comme des variables localesΓavec la possibilité d'être initialisés lors de la création du processus. Dans le deuxième casΓles paramètres ont leur sémantique habituelle des langages de type procédural. Deux modes de passageΓ*in* et *in/out*

sont possiblesΓet correspondent au passage respectivement par valeur et par référence.

parameter ::=
    parameter-id type-id

mode ::=
    in/out | **in**

Expression

Etant donné que les types de données et les opérations sur leurs valeurs peuvent être considérés comme un paramètre dans la définition de la sémantique dynamique de SDLΓnous avons considéré un ensemble relativement réduit des expressions SDLΓmais suffisamment expressif pour décrire les aspects liés au contrôle. Mis à part les expressions classiquesΓconstruites à partir des variables et des opérationsΓnous avons des expressions plus spécifiques au langage SDLΓque nous détaillons par la suite.

L'opérateur **view** permet l'accès aux valeurs des variables importées à l'aide d'une déclaration **viewed**. L'opérateur **active** test l'activité d'une temporisation : si elle est armée et en attente du moment d'expiration. L'expression **now** donne l'instant courant du temps global. Les expressions **self**Γ**sender**Γ**offspring** et **parent** sont des identificateurs d'instances de processus respectivementΓ**self** est l'instance couranteΓ**sender** est l'instance depuis laquelle provient le dernier signal consomméΓ**parent** est l'instance qui a créé l'instance courante et **offspring** est la dernière instance créée par l'instance courante.

expression ::=
    variable-id
| *function-id* ($expression_1$, .. $expression_m$)
| **view** (*variable-id*)
| **active** (*timer-id* ($expression_1$, .. $expression_m$))
| **now** | **self** | **parent** | **offspring** | **sender**

1.2 Sémantique dynamique

La sémantique dynamique officielle de SDL est donnée par l'annexe F3 de la Recommandation Z.100 de l'ITU [IT94b].

Un système SDL est *interprété* par un ensemble de *méta-processus* concurrents communicants. La communication est *synchrone* à la CSP [Hoa78]. La structure globale du modèle d'interprétation est présentée à la figure 1.1. Les méta-processus utilisés sont brièvement décrits par la suite.

- system : gère la création dynamique et le routage de signaux entre différentes instances de processus SDL. Il existe une instance unique du méta-processus system pendant

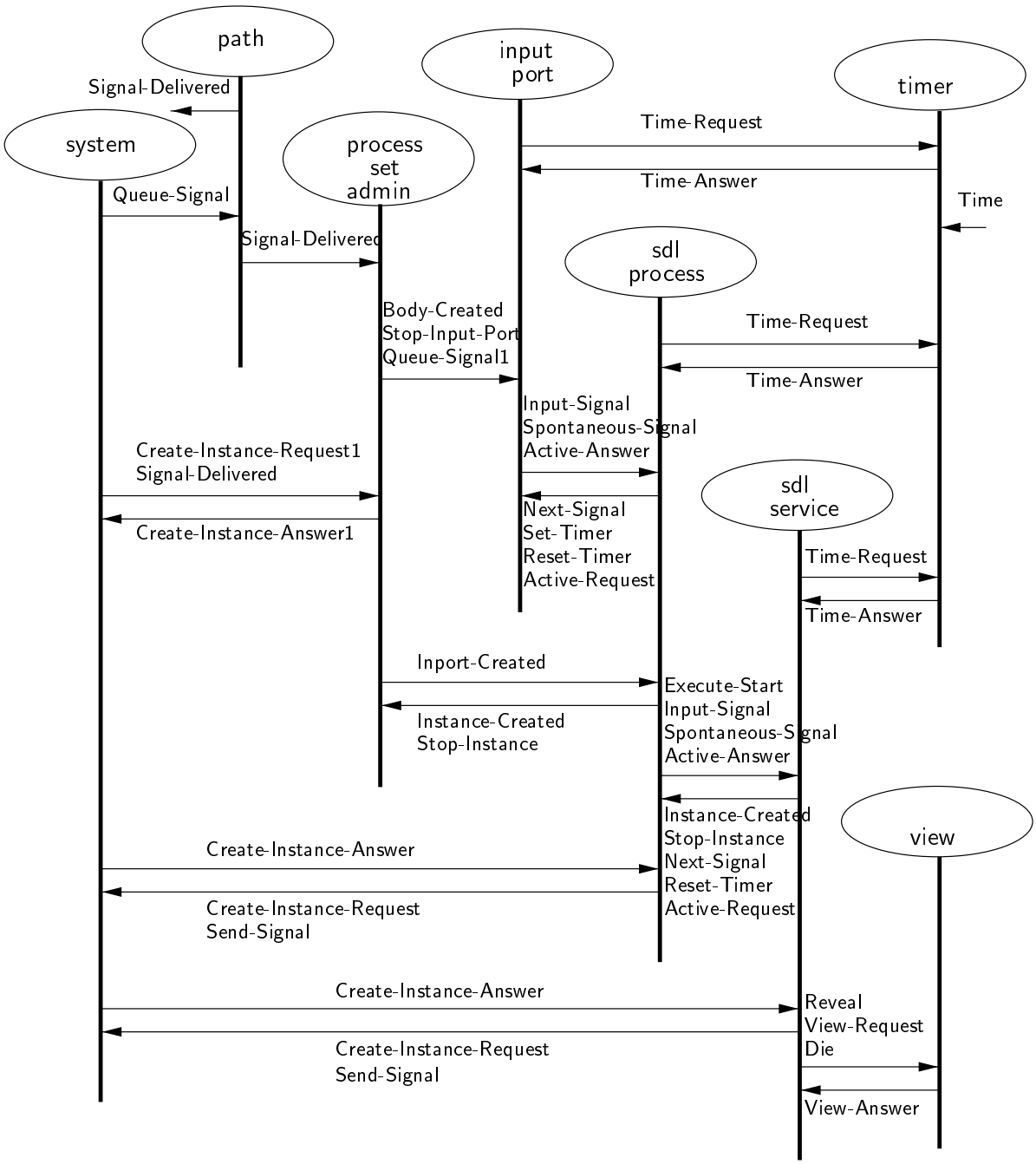

FIG. 1.1 – *La structure d'interprétation de programmes SDL.*

l'interprétation.

- path : gère les délais de transmission arbitraires associés aux canaux. Il existe plusieurs instances du méta-processus pathΓune pour chaque séquence de canaux connectant deux blocs feuilles [1]Γou un bloc feuille et l'environnement.
- view : gère les variables partagées entre processus. Il existe une instance unique du méta-processus view pendant l'interprétation.
- timer : gère l'évolution du temps global. Il existe aussi une instance unique du méta-processus timer pendant l'interprétation.
- process-set-admin : gère la création dynamique et la réception de signaux pour l'ensemble d'instances appartenant au même processus SDL. Il existe plusieurs instances du méta-processus process-set-adminΓune pour chaque processus SDL défini dans le système.
- input-port : gère la réception de signaux et l'expiration de temporisations pour une instance de processus SDL. A tout momentΓil existe plusieurs instances du méta-processus input-portΓune pour chaque instance active de processus SDL.
- sdl-process : interprète le comportement d'une instance de processus SDL. Il existe aussi plusieurs instances du méta-processus sdl-processΓune pour chaque instance active de processus SDL.
- sdl-service : interprète le comportement d'un service dans une instance de processus SDL. Il existe plusieurs instances du méta-processus sdl-serviceΓune pour chaque instance active de service.

Nous allons maintenant détailler chacun de ces méta-processus et notamment leurs interactions pendant l'interprétation d'un système SDL.

1.2.1 system

Le méta-processus system gère la création dynamique d'instances et le routage de signaux entre différents instances de processus SDL. Son fonctionnement abstrait est présenté à la figure 1.2.

Plus précisémentΓle méta-processus system reçoit les demandes de création de nouvelles instances issues par des méta-processus sdl-process ou sdl-service (Create-Instance-Request). Il transmet la demande au méta-processus process-set-admin associé à l'ensemble d'instances du processus crée (Create-Instance-Request1). Après avoir éventuellement crée la nouvelle instanceΓprocess-set-admin répond au system (Create-Instance-Answer1). FinalementΓsystem répond au méta-processus sdl-process ou sdl-service créateur (Create-Instance-Answer).

1. blocs constitués de processus

Le routage de signaux est effectué de la manière suivante. Le méta-processus system reçoit tous les envois de signaux SDL par des méta-processus sdl-process ou sdl-service (Send-Signal). EnsuiteΓen fonction de l'information contenue dans le signalΓil cherche une route connectant le processus émetteur et le processus récepteur. Si une telle route n'existe pasΓle signal est silencieusement perdu (i). Si la route existe et contient un canalΓle signal est délivré au méta-processus path correspondantΓpour introduire le retard (Queue-Signal). SinonΓle signal est délivré immédiatement au process-set-admin associé au processus SDL récepteur (Signal-Delivered).

```
system ≙
    cycle
        input Create-Instance-Request from sdl-processΓsdl-service;
            output Create-Instance-Request1 to process-set-admin;
                input Create-Instance-Answer1 from process-set-admin;
                    output Create-Instance-Answer to sdl-processΓsdl-service;
        []
        input Send-Signal from sdl-processΓsdl-serviceΓenv;
            (
            output Queue-Signal to path;
            []
            output Signal-Delivered to process-set-admin;
            []
            i;
            )
    endcycle
```

FIG. 1.2 – *Le méta-processus* system.

1.2.2 path

Le méta-processus path gère les délais de transmission arbitraires associés aux canaux. Son fonctionnement abstrait est présenté à la figure 1.3.

Ce méta-processus simule le comportement d'une file infinie de signaux. A tout momentΓil est possible soit qu'un nouveau signal arrive dans la file (Queue-Signal)Γsoit que le signal d'entête est délivré au destinataire (Signal-Delivered). Les signaux arrivent dans la file via le méta-processus system et sont délivrés aux instances de méta-processus process-set-admin.

1.2.3 view

Le méta-processus view gère les variables partagées entre plusieurs instances de processus SDL. Son fonctionnement abstrait est présenté à la figure 1.4.

Le méta-processus view garde une copie de toutes les variables partagées existantes dans

path ≙
cycle
input Queue-Signal *from* system;
[]
output Signal-Delivered *to* process-set-admin;
endcycle

FIG. 1.3 – *Le méta-processus* path.

les instances actives du système. Chaque fois qu'un processus ou un service met à jour une variable partagée il envoi sa nouvelle valeur au méta-processus view (Reveal). D'autre partΓ si un processus demande la valeur d'une variable partagée (en utilisant l'expression **view**) il reçoit cette valeur de la part du méta-processus view (View-RequestΓView-Answer). Le méta-processus view est aussi notifié de la destruction d'instancesΓpour détruire ses copies des variables partagées (Die).

view ≙
cycle
input Reveal *from* sdl-processΓsdl-service;
[]
input View-Request *from* sdl-processΓsdl-service;
output View-Answer *to* sdl-processΓsdl-service;
[]
input Die *from* sdl-processΓsdl-service;
endcycle

FIG. 1.4 – *Le méta-processus* view.

1.2.4 timer

Le méta-processus timer gère l'évolution du temps global dans le modèle d'interprétation SDL. Son fonctionnement abstrait est présenté à la figure 1.5.

Le fonctionnement du méta-processus timer est basé sur la supposition que l'environnement lui envoit à des intervalles réguliers un signal particulier (Time). Ce signal signifie une évolution discrète en temps réel et sert pour mettre à jour la valeur courante du temps dans le système. EnsuiteΓchaque fois qu'un processus ou un service utilise l'expression **now** il obtient la valeur correspondante de la part du méta-processus timer (Time-RequestΓTime-Answer). La gestion des temporisations par les instances du méta-processus input-port est basée sur la même interaction pour obtenir la valeur courante du temps

timer ≙
cycle
input Time *from* env;
[]
input Time-Request *from* sdl-processΓsdl-serviceΓinput-port ;
output Time-Answer *to* sdl-processΓsdl-serviceΓinput-port;
endcycle

FIG. 1.5 – *Le méta-processus* timer.

1.2.5 process-set-admin

Le méta-processus process-set-admin gère la création dynamique d'instances et la réception de signaux pour l'ensemble d'instances appartenant au même processus SDL. Son fonctionnement abstrait est présenté à la figure 1.6.

Le process-set-admin reçoit les demandes de création de nouvelles instances de la part du méta-processus system (Create-Instance-Request1). Si le nombre courant d'instances ne dépasse pas la valeur maximale autorisée pour le processusΓil démarre une nouvelle instance du méta-processus input-port et une autre du méta-processus sdl-process (Inport-CreatedΓBody-CreatedΓInstance-Created). EnsuiteΓil répond au system (Create-Instance-Answer1).

D'autre part le process-set-admin gère la réception de signaux pour l'ensemble d'instances (Signal-Delivered). Si une instance peut être identifiée comme étant le destinataire du signalΓ alors le signal est délivré au input-port correspondant (Queue-Signal1). SinonΓle signal est silencieusement perdu (i).

FinalementΓprocess-set-admin est notifié par la destruction de ses instances (Stop-Instance).

1.2.6 input-port

Le méta-processus input-port gère la réception de signaux et l'expiration de temporisations pour une instance de processus SDL. Son fonctionnement abstrait est présenté à la figure 1.7.

Un input-port est créé par le process-set-admin (Body-Created) et arrêté par l'instance du méta-processus sdl-process associé (Stop-Input-Port).

Le premier rôle du méta-processus input-port est la gestion de signaux destinés à l'instance associée. D'une partΓ input-port peut recevoir des signaux de la part du méta-processus process-set-admin (Queue-Signal1). Ces signaux sont stockés dans une file interneΓsupposée infinie. D'autre partΓinput-port répond aux demandes de signaux issues par le sdl-process associé (Next-Signal). Il peut soit fournir le premier signal de la file qui n'est pas sauvéΓsi un tel signal existe (Input-Signal)Γsoit générer un signal spontanéΓs'il existe des transitions spontanées possibles à l'état courant du méta-processus sdl-process (Spontaneous-Signal).

Le deuxième rôle du méta-processus input-port est la gestion des temporisations et leurs expirations (ou *timeouts*). Il gère la table des temporisations actives dans le sdl-process associé.

```
process-set-admin ≙
    cycle
        input Create-Instance-Request1 from system;
            (
            output Body-Created to input-port;
                output Inport-Created to sdl-process;
                    input Instance-Created from sdl-process;
            []
            i;
            )
                output Create-Instance-Answer1 to system;
        []
        input Signal-Delivered from systemΓpath;
            (
            output Queue-Signal1 to input-port;
            []
            i;
            )
        []
        input Stop-Instance from sdl-process;
    endcycle
```

FIG. 1.6 – *Le méta-processus* process-set-admin.

Chaque fois qu'une temporisation est armée par le sdl-process elle est rangée dans la tableΓen enlevant sa précédente version active (Set-Timer). De mêmeΓau moment d'un désarmementΓ la temporisation est enlevée de la table (Reset-Timer). L'évaluation d'une expression **active** est traduite aussi vers des interactions avec le input-port (Active-RequestΓActive-Answer). FinalementΓinput-port scrute le passage du temps pour détecter l'expiration de temporisations présentes dans sa table (Time-RequestΓTime-Answer). Si une temporisation active a expiré elle est enlevée de la table et ajoutée dans la file des signaux.

```
input-port ≙
    input Body-Created from process-set-admin;
        cycle
            input Queue-Signal1 from process-set-admin;
            []
            input Next-Signal from sdl-process;
            []
            output Input-Signal to sdl-process;
            []
            output Spontaneous-Signal to sdl-process;
            []
            input Set-Timer from sdl-process;
            []
            input Reset-Timer from sdl-process;
            []
            input Active-Request from sdl-process;
                output Active-Answer from sdl-process;
            []
            output Time-Request to timer;
                input Time-Answer from timer;
        endcycle
            input Stop-Input-Port from sdl-process;
```

FIG. 1.7 – *Le méta-processus* input-port

1.2.7 sdl-process

Le méta-processus sdl-process simule l'exécution d'un processus SDL. Deux situations sont possibles : le processus SDL a été spécifié soit par son graphe d'étatsΓsoit par une composition de services. Dans la première situationΓ le fonctionnement du méta-processus sdl-process consiste à interpréter le graphe d'états. Dans la deuxième situationΓprésentée à la figure 1.8Γ le fonctionnement consiste à coordonner l'exécution de ses services. Nous allons détailler ce deuxième casΓle premier étant plus ou moins similaire au fonctionnement d'un sdl-service (à voir plus loin).

GénéralementΓun sdl-process est créé par le process-set-admin (Inport-CreatedΓInstance-Created).

Immédiatement après sa créationΓsdl-process démarre une instance du méta-processus sdl-service pour chaque service le composant (Instance-Created).

Dans un premier temps il demande à chaque service d'exécuter sa transition start Execute-Start. L'exécution de ces transitions est *atomique* i.e. une fois lancée le sdl-process n'en lance pas une autre tant que celle ci n'est pas terminée. Pendant l'exécution d'une transition dans un serviceΓsdl-process joue le rôle d'interface pour les messages concernant le input-port. AinsiΓtout armement et désarmement d'une temporisation (Set-TimerΓReset-Timer) ou test d'activité (Active-RequestΓActive-Answer) passe par le sdl-process vers le input-port. La terminaison d'une transition est signalée au processus soit par une demande d'un nouvel signal à consommer (Next-Signal) ou par la terminaison effective du service (Stop-Instance).

Une fois toutes les instances démarréesΓsdl-process demande au input-port le prochain signal à consommer (Next-Signal). Tous les signaux sauvés par au moins un des services sont sauvés au niveau du processus. Une transition spontanée peut avoir lieu s'il en existe au moins une dans un des services. Le input-port répond soit par le prochain signal (Input-Signal)Γou par un signal spontané (Spontaneous-Signal). En fonction du réponseΓsdl-process choisis le service qui devrait le traiter et lance sa transition correspondante (Input-SignalΓSpontaneous-Signal ...). L'exécution est toujours *atomique*Γles autres services restant en attente.

EnfinΓle sdl-process finit son exécution au moment où tous les services ont cessé d'exister (Stop-Instance). La fin est signalée au méta-processus process-set-admin (Stop-Instance) et au meta-processus view (Die).

1.2.8 sdl-service

Le méta-processus sdl-service simule l'exécution d'un service SDL. Son fonctionnement abstrait est présenté à la figure 1.9.

Un sdl-service est créé et démarré par un sdl-process parent (Instance-CreatedΓExecute-Start). EnsuiteΓpendant tout son exécution il interagit avec le processus parent pour obtenir des signaux à consommer de l'input-port correspondant (Next-SignalΓInput-SignalΓSpontaneous-Signal). Il exécute les transitions déclenchées par la consommation de ces signaux. Par exempleΓil peut envoyer des signaux (Send-Signal)Γinitier la création de nouvelles instances (Create-Instance-RequestΓCreate-Instance-Answer)Γou armer et désarmer des temporisations (Set-TimerΓReset-Timer). Il évalue des expressions sur les variables partagées (View-RequestΓ View-Answer)Γsur le temps (Time-RequestΓTime-Answer)Γou sur les temporisations actives (Active-RequestΓActive-Answer)Γetc. EnfinΓson exécution peut être terminée suite à une action **stop** ; la terminaison est signalée aux méta-processus parents sdl-process (Stop-Instance) et view (Die).

```
sdl-process ≙
        input Inport-Created from process-set-admin;
            cycle input Instance-Created from sdl-service; endcycle
                output Instance-Created to process-set-admin;
                    cycle
                        (
                        output Execute-Start to sdl-service;
                        []
                        output Next-Signal to input-port;
                            (
                            input Input-Signal from input-port;
                                output Input-Signal to sdl-service;
                            []
                            input Spontaneous-Signal from input-port;
                                output Spontaneous-Signal to sdl-service;
                            )
                        )
                            (
                            input Stop-Instance from sdl-service;
                            []
                            input Next-Signal from sdl-service;
                            []
                            input Set-Timer from sdl-service;
                                output Set-Timer to input-port;
                            []
                            input Reset-Timer from sdl-service;
                                output Reset-Timer to input-port;
                            []
                            input Active-Request from sdl-service;
                                output Active-Request to input-port;
                                    input Active-Answer from input-port;
                                        output Active-Answer to sdl-service;
                            )
                    endcycle
                        output Stop-Instance to process-set-admin;
                            output Die to view;
```

FIG. 1.8 – *Le méta-processus* sdl-process

```
sdl-service ≙
    output Instance-Created to sdl-process;
        input Execute-Start from sdl-process;
            cycle
                output Next-Signal to sdl-process;
                    (
                    input Input-Signal from sdl-process
                    []
                    input Spontaneous-Signal from sdl-process
                    )
                []
                output Set-Timer to sdl-process;
                []
                output Reset-Timer to sdl-process;
                []
                output Send-Signal to system;
                []
                output Create-Instance-Request to system;
                    input Create-Instance-Answer from system;
                []
                output Active-Request to sdl-process;
                    input Active-Answer from sdl-process;
                []
                output Reveal to view;
                []
                output View-Request to view;
                    input View-Answer from view;
                []
                output Time-Request to timer;
                    input Time-Answer from timer;
            endcycle
                output Stop-Instance to sdl-process;
                    output Die to view;
```

FIG. 1.9 – *Le méta-processus* sdl-service

1.3 Discussion

Un certain nombre de critiques ont été formulées à propos de la définition formelle Z.100Γ dont les plus importantes sont les suivantes :

- la définition formelle Z.100 est basée sur un modèle de *communication synchrone*. L'exécution d'actions dans une instance est fortement synchronisée avec des actions de méta-processus globaux uniques ce qui introduit des contraintes bizarres sur le fonctionnement globale d'un système. Par exempleΓl'envoi d'un signal par une instance n'est pas possible tant que le système répond a une demande de création pour une autre instance. Ça ne correspond à aucune intuition sur le principe d'exécution asynchrone de SDL.
- le *temps* est traité d'une manière *rudimentaire*. Il existe une horloge globale qui avance indépendamment du fonctionnement du système et les méta-processus peuvent uniquement le consulter. Avec ce modèle aucune contrainte temporelle n'est respectée. Par exempleΓla sémantique actuelle ne garantit pas qu'une temporisation T_1 expire avant une temporisation T_2Γdans le cas naturel oùΓon arme d'abord T_1 à un temps t_1 et ensuite T_2 à un temps t_2 tel que $t_1 < t_2$. Un scénario possible est le suivant. Les deux temporisations sont rangées dans la table des temporisations actives. EnsuiteΓle temps avance au delà des temps d'expirations sans que le méta-processus input-port le consulte. FinalementΓle méta-processus input-port trouve que les deux ont expirés et le met dans la file dans un ordre quelconqueΓpar exemple T_2 avant T_1.
- finalementΓla sémantique Z.100 est difficilement utilisable à cause de sa taille : environ 500 pages de descriptions de fonctions META-IV.

D'autres sémantiques ont été proposées dans la littérature. Les plus intéressantes sont détaillées dans la suite.

1.3.1 Réseaux de flot de données asynchrones

Dans [Bro91] a été proposée une sémantique à base de réseaux de flot de données asynchrones. Les notions centrales sont celle de flot (*stream*)Γdes séquences sur un alphabetΓet celle de fonction sur les flots (*stream processing function*).

IciΓon considère les canaux et les routes d'un système comme étant des flots définis sur l'ensemble de signaux. Chaque processus est vu comme une fonction de transformation sur les flots. IntuitivementΓà l'exécution d'un processus correspond une fonction définie sur l'ensemble des contenus des canaux et des routes.

Cette sémantique présente l'avantage d'être compositionnelle : à partir des fonctions associées à chaque processusΓon construit des fonctions associées aux blocs et finalement au système. De plusΓelle donne une vision abstraite du fonctionnement d'un système SDL.

CependantΓcertains aspects de SDL ne sont pas du tout modélisés. Il s'agit notamment de la création dynamique des instances et des aspects temporelsΓdont le traitement n'est pas facilement envisageable dans un tel formalisme.

1.3.2 Systèmes de transitions étiquetées

Dans [God91] est présentée une sémantique *opérationnelle* pour Basic SDLΓun sous-ensemble significatif de SDL. Elle consiste à définir de manière incrémentale des systèmes de transitions étiquetées associés respectivement aux instancesΓblocsΓcanaux et systèmes :

- à chaque instance d'un processus SDL est associé un système de transitions étiquetées qui définit toutes ses exécutions possibles. Les états de ces systèmes sont des tuples composés de l'état de contrôleΓdes valeurs des variablesΓdes signaux en attente dans la fileΓet des temporisations actives. Les transitions correspondent aux actions élémentaires du processus SDL. AinsiΓnous retrouvons des transitions *visibles* ou *internes*. Les transitions visibles correspondent à la réception de signaux dans la fileΓà l'émission de signaux vers d'autres instances ou à la création dynamique de nouvelles instances. Les transitions internes correspondent aux affectationsΓaux évaluation de gardesΓà l'expiration/armement/désarmement de temporisationsΓà l'appel de procéduresΓetc.
- les systèmes de transitions étiquetées associés aux instances sont composés pour obtenir les systèmes de transitions étiquetées associés aux blocs. La composition est asynchrone pour les actions internes (*entrelacement*) et synchronise les actions visibles duales dans instances différentes (e.gΓla réception d'un signal dans une instance avec l'émission de ce signal dans une autre). Des transitions supplémentaires visibles sont définies pour modéliser l'interaction avec l'environnement e.gΓla réception/émission de signaux vers de canaux.
- un système de transitions étiquetées est associé à chaque canal défini dans le système SDL. Ce système modélise simplement une file infinie de signauxΓcapable à tout moment soit de recevoir un nouvel élémentΓsoit de délivrer le premier élément stocké.
- le système de transitions étiquetées associé au système SDL est finalement obtenu en composant les systèmes de transitions associées aux blocs et aux canauxΓavec la prise en compte de l'environnement. La composition est asynchroneΓexception faite des actions duales dans les blocs et les canaux qui sont synchronisées (e.gΓl'émission d'un signal vers un canal et la réception du signal par ce canal).

Là aussiΓle traitement du temps est *inutilement compliqué*. Il est supposé l'existence d'une horloge globale unique *time*. Cette horloge avance d'un pas discretΓde manière indépendante du fonctionnement du système. De plusΓchaque instance possède une variable locale *now*Γ pour mémoriser la valeur du temps global. La sémantique prévoit des transitions internes pour mettre à jour cette variableΓindépendamment dans chaque instance. Pour satisfaire les contraintes temporelles associées aux expirations de temporisationsΓune solution basée sur une notion de priorité est adoptée. AinsiΓles transitions d'expiration et les transitions

de mise à jour des variables *now* sont prioritaires par rapport aux autres. La solution est cependant partielle en ce sens queΓcomme dans la Z.100Γle temps peut toujours progresser au delà de la limite d'expiration d'une temporisationΓsans que l'expiration a eu lieu.

1.3.3 Algèbre de processus

Une sémantique basée sur une algèbre de processus a été initialement proposée dans [BM95] et ensuite ameliorée dans [BMU98]. Elle traite un sous ensemble sans blocs et sans canaux de SDL nommé φ^-SDL.

L'algèbre de processus utilisée est relativement complexe. C'est une extension d'une algèbre de processus temporisée classique avec des opérateurs plus ou moins spécifiques à SDL : signaux propositionnels et conditionsΓrécursionΓopérateur de comptage de processusΓopérateur d'étatΓetc. Une sémantique complète de cette algèbre a été définie et permet de dériver de systèmes de transitions à partir de ses termes.

La sémantique de φ^-SDL est définie en deux phases :

- initialementΓune sémantique faisant abstraction des aspects de comportement dynamique et définie pour les processus. Elle consiste à leur associer un terme dans l'algèbre qui les décrit syntaxiquement. Cette phase est purement syntaxiqueΓdes règles basées sur la syntaxe de φ^-SDL permettant de construire les termes correspondants.
- ensuiteΓla sémantique d'un système est définie par la composition parallèle de termes qui le compose et en ajoutant notamment l'opérateur d'étatΓpour interpréter les actions dans le processus. ConcrètementΓon défini l'ensemble support pour les états et les fonctions de transformation associées aux actions élémentaires e.gΓentréeΓsortieΓ affectation.

La sémantique du temps considérée dans cette approche est plus intéressante. En faitΠl'évolution du temps est prise en compte à l'intérieur de chaque processus : le temps peut progresser uniquement si le processus est dans un étatΓen attente d'un signal à consommer. La progression du temps est synchronisée à travers tous les processus présents dans le système.

1.3.4 Réseaux de Petri

Dans [FG98] est proposée une sémantique à base de réseaux de Petri. Les auteurs présentent une méthode compositionnelle pour synthétiser un réseau étiqueté à partir d'une spécification SDL. Un grand nombre de détails sont traités dont la création dynamique des instancesΓ l'appel de procéduresΓla communicationΓetc. CependantΓles aspects temporels sont complètement ignorés.

En conclusion généraleΓnous avons retenu d'une part l'absenceΓdans la plupart des casΓ d'une sémantique adéquate pour le temps dans SDL etΓd'autre part l'absence de résultats

concrets montrant l'utilité pratique de ces sémantiques pour la validation de spécifications SDL. Ces deux aspects ont une importance majeure sur l'utilisation effective de SDL dans des applications réelles de télécommunications (ou d'autres applications) où les contraintes de fonctionnement temps-réel deviennent de plus en plus importantes et doivent être prises en compte pendant la validation.

Dans le chapitre suivant nous présentons un modèle pour des systèmes asynchrones. Avec une sémantique opérationnelle complète et bien définie ce modèle sera utilisé par la suite pour donner une sémantique non-ambiguë et pour développer des outils de validation pour un sous-ensemble représentatif du langage SDL.

Chapitre 2

IF

IF (*Intermediate Format*) est un modèle intermédiaire à base d'automates temporisés communicantsΓconçu pour la description et la validation formelle de systèmes asynchrones. Ce modèle a été choisi pour satisfaire un certain nombre d'exigencesΓdont les plus importantes sont l'expressivité par rapport aux formalismes de description de haut niveauΓune sémantique opérationnelle bien définie et un maximum de support pour la validation formelle.

Nous utilisons IF comme niveau intermédiaire de représentation pour des systèmes asynchrones. Il se situe entre des formalismes de description standardisésΓcomme SDL ou LOTOSΓ et des modèles mathématiques utilisés dans la vérificationΓcomme les automates temporisés ou les systèmes de transitions étiquetées. IF est un modèle suffisamment expressif pour rester en amont du problème d'explosion d'états et pour simuler simplement des concepts existants dans les formalismes de spécification. En faitΓles programmes IF gardent de manière explicite un certain nombre d'informations comme par exemple le parallélismeΓle tempsΓou encore les données manipulées.

IF a une sémantique opérationnelle complètement définie en termes de systèmes de transitions étiquetées. ConcrètementΓla sémantique consiste dans un ensemble des règles pour construireΓà partir d'un programme IFΓl'ensemble de tous ses comportementsΓsous la forme d'un système de transitions étiquetées. C'est un aspect très important pour la validation automatique basée sur des modèlesΓétant donné que toute technique existante de validation s'applique habituellement à ce niveau d'abstraction.

Ce chapitre présente les principaux aspects syntaxiques et sémantiques du modèle intermédiaire IF. Nous commençons par présenter brièvement une syntaxe abstraiteΓconçue pour avoir une représentation textuelle du modèle. Ensuite nous présentons la sémantique opérationnelle. Nous finissons par une brève discussion sur les limites actuelles de IFΓainsi que sur certaines perspectives d'évolution.

2.1 Syntaxe abstraite

2.1.1 Structure

Système

Un système IF est constitué d'un ensemble d'automates temporisésΓqui communiquent soit de manière asynchrone par envoi de signaux à travers des files d'attenteΓsoit de manière synchrone par rendez-vous aux portes de synchronisation. SyntaxiquementΓun programme IF contient une liste de définitions : des types de donnéesΓdes signaux de communication asynchroneΓdes portes des synchronisationΓdes files d'attenteΓdes variables partagéesΓune expression de synchronisation et des processus.

system ::=
system *system-id* **;**
type $type_1$ .. $type_m$
signal $signal_1$ .. $signal_n$
gate $gate_1$ .. $gate_p$
buffer $buffer_1$ .. $buffer_q$
var $variable_1$ .. $variable_r$
sync *sync-expression* **end ;**
$process_1$ .. $process_s$

Processus

Un processus IF est un automate temporisé étendu. Il est identifié par un nom unique qui est aussi son *pid*. Un processus a une file d'attente associée en entrée. Un processus est constitué respectivement d'un ensemble de variables locales typéesΓincluant des horlogesΓ d'un ensemble d'états de contrôle et d'un ensemble de transitions entre ces états.

process ::=
process *process-id* **;**
buffer *buffer-id* **;**
var $variable_1$ .. $variable_m$
state $state_1$ .. $state_n$
transition $transition_1$ .. $transition_p$

2.1.2 Communication

Signal

Les signaux sont les objets transportés de manière asynchrone à travers les files d'attente. Comme en SDLΓla déclaration d'un signal comporte respectivement son nom et la liste de

types des paramètres transportés.

signal::=
    signal-id (*type-id*$_1$, .. *type-id*$_m$);

Buffer

Les files d'attente (ou *buffers*) assurent la réalisation de la communication asynchrone dans un système IF. Elles sont identifiées par un nom unique et transportent des signaux envoyés entre processus ou entre processus et l'environnement du système. Il n'y a pas de restrictions d'utilisation de files : chaque processus peut envoyer ou recevoir des signaux à travers toutes les files du système. Nous considérons plusieurs catégories de files d'attente : des files bornées ou non-bornéesΓordonnées ou non-ordonnées et avec ou sans pertes.

buffer::=
    buffer-id: *buffer-class* **of** *signal-id*$_1$, .. *signal-id*$_m$;

Porte

Les portes servent à réaliser la communication synchrone dans un système IF. AinsiΓdeux ou plusieurs processus peuvent se synchroniser et échanger des valeurs par l'intermédiaire d'une porte. La synchronisation est celle existante dans le langage LOTOS [ISO88]. Pour simplifier l'analyse du modèleΓles portes sont typées : chaque porte précise la liste des types des valeurs qui peuvent y être échangées.

gate::=
    gate-id (*type-id*$_1$, .. *type-id*$_m$);

Synchronisations

L'expression de synchronisation décrit comment les processus communiquent de manière synchrone les uns avec les autres par l'intermédiaire des portes de synchronisation. Une telle expression est construite à partir des identificateurs de processus en utilisant l'opérateur binaire de composition parallèle avec synchronisation sur un ensemble de portes.

sync-expression::=
    process-id
| *sync-expression* | [*gate-id*$_1$, .. *gate-id*$_m$] | *sync-expression*

2.1.3 Contrôle

Etat

Les états sont un élément important dans la description du contrôle de processus IF. Les états correspondent aux modes de fonctionnement des processus : à tout moment un processus est dans un état et suivant les événement internes ou externes qui se produisent il peut passer dans un autre. Les états possèdent un certain nombre de caractéristiques. D'abordΓ un état peut être *initial*Γc'est-à-dire le processus peut démarrer son exécution à partir de cet état. EnsuiteΓun état peut être *stable* ou *instable*. Cette propriété concerne l'atomicité des séquences d'exécution. IntuitivementΓà l'exécution du système IF globalΓun processus ne pourra pas être interrompu par des autres lorsqu'il est dans un état instable. Aux états peuvent être associés des *filtres* sur les files d'attente. Ces filtres permettent soit de retarder la consommation des certains signaux pour des moments ultérieursΓsoit la destruction de signaux considérés comme inutiles. FinalementΓà chaque état peut être associée une condition de progression du temps. IntuitivementΓle processus peut rester dans un état tant que cette condition est vraie.

state ::=
    state-id init stable
        discard *signal-filter*$_1$ .. *signal-filter*$_m$
        save *signal-filter*$_1$ .. *signal-filter*$_n$
        tpc *expression* ;
    end ;

signal-filter ::=
    signal-id **in** *buffer-id* **if** *expression* ;

Transition

Les processus évoluent d'un état à l'autre en exécutant des transitions. Les transitions sont plus ou moins urgentes en fonction d'une échéance associée explicitement. Elles sont de deux catégories: synchrones ou asynchrones. Dans les deux casΓles transitions comportent une garde qui conditionne leur exécution et des affectations aux variables. De plusΓune transition asynchrone peut être conditionnée par la réception d'un signal d'une file d'attente et comporte des émissions asynchrones de signaux vers d'autres files. Une transition synchrone comporte un rendez-vous synchrone avec échanges de valeurs sur une porte spécifiée.

transition ::=
    from *state-id urgency*
        transition-body
    to *state-id* ;

transition-body ::=
    if *expression*

[*input*] *output*$_1$, .. *output*$_m$
assign$_1$, .. *assign*$_n$
| **if** *expression*
synchronous-input
assign$_1$, .. *assign*$_m$

Action

Les actions élémentaires associées aux transitions sont respectivement des réceptions ou émissions asynchronesΓdes synchronisations et des affectations. Les réceptions asynchrones indiquent le signal attenduΓles paramètres en réception et la file d'attente concernée. Une garde supplémentaire portant éventuellement sur les valeurs reçues peut contraindre la réalisation de la réception. RéciproquementΓles émissions asynchrones précisent le signal envoyéΓ la liste de ses paramètres et la file d'attenteΓdonnée soit explicitement par son nomΓsoit implicitement par une expression de type pid. Dans ce deuxième casΓle signal sera mis dans la file associée par défaut au processus avec le pid correspondant. Les synchronisations comportent le nom de la porte utilisée et une liste d'offresΓrespectivement émissions de valeurs d'expressions ou réceptions de valeurs dans des variables. Là aussiΓune garde supplémentaire portant sur les valeurs reçues lors de la synchronisation peut encore contraindre sa réalisation. FinalementΓles affectations consistent à attribuer la valeur d'une expression à une variableΓsoit de remettre la variable à une valeur initiale fixée.

input::=
input *signal-id* (*variable-id*$_1$, .. *variable-id*$_m$) **from** *buffer-id* **if** *expression*

output::=
output *signal-id* (*expression*$_1$, .. *expression*$_m$) **to** { *buffer-id* | *expression* }

synchronous-input::=
input *gate-id offer*$_1$ .. *offer*$_m$ **if** *expression*

offer::=
? *variable-id* | **!** *expression*

assign::=
set *variable-id* **:=** *expression*
| **reset** *variable-id*

2.1.4 Données

Variable

Les variables sont identifiées par un nom unique et sont typées. Elles peuvent être soit globalesΓdéfinies au niveau du système IF et ainsi visibles dans tous les processusΓsoit

locales Γ définies et visibles uniquement à l'intérieur d'un processus IF.

variable ::=
    variable-id **:** *type-id* **;**

Expression

Généralement Γ les expressions peuvent être mises sous la forme simplifiée donnée ci-après. Elles sont définies par des termes. Un terme est soit l'application d'une fonction à des termes Γ soit une variable Γ soit une constante. Nous n'insistons pas sur ces aspects classiques dans les langages Γ dans la mesure où l'aspect donnée est complètement orthogonale à l'aspect contrôle.

expression ::=
    variable-id
| *function-id*($expression_1$, .. $expression_m$)

2.2 Sémantique dynamique

Nous allons définir la sémantique dynamique des programmes IF à l'aide d'une hiérarchie de modèles basés sur des automates. Le modèle de base est le système de transitions étiquetées. Ensuite nous considérons le modèle automate temporisé avec échéances Γ choisit pour modéliser les aspects temporels. Nous continuons avec le modèle automate temporisé communiquant par files d'attentes. Finalement Γ nous présentons le modèle automate temporisé communicant étendu avec des données Γ et qui décrit un processus IF.

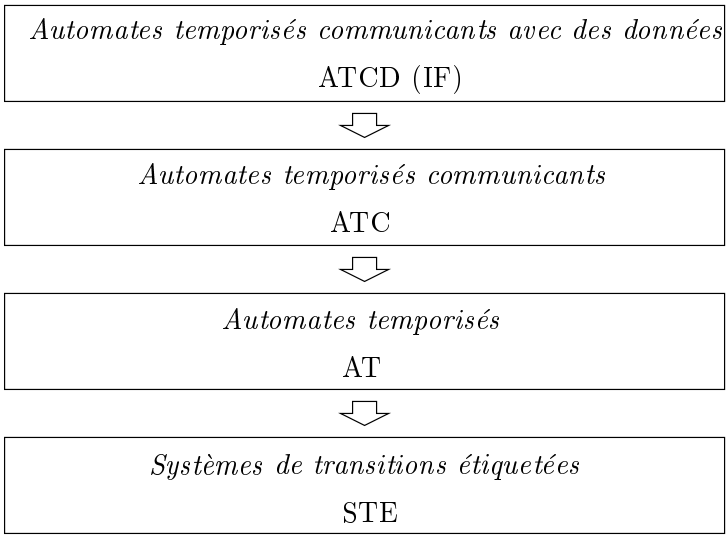

FIG. 2.1 – *La hiérarchie de modèles sémantiques de IF.*

Pour chacun de ces modèles nous allons définir la syntaxeΓla sémantique dynamique en terme du modèle de niveau précédent et un opérateur de composition parallèle avec synchronisation.

2.2.1 Systèmes de transitions étiquetées

Définition 2.1 (STE : syntaxe) *Un système de transitions étiquetées est un tuple* (A, Q, T) *où :*

- A *est un ensemble d'actions ;*
- Q *est un ensemble d'états ;*
- $T \subseteq Q \times A \times Q$ *est un ensemble de transitions étiquetées. Une transition* $t = (q, a, q') \in T$ *sera notée par* $q \xrightarrow{a} q'$ *;* $q = source(t)$, $q' = target(t) \in Q$ *sont respectivement les états source et destination et* $a = action(t) \in A$ *est l'action associée à* t.

Pour comparer les systèmes de transitions étiquetéesΓnous avons adopté la sémantique arborescente induite par la relation de bisimulation forte [Mil80ΓPar81].

Définition 2.2 (STE : sémantique) *Soit* $P_k = (A_k, Q_k, T_k)_{k=1,2}$ *deux STE.* P_1 *et* P_2 *sont dits fortement équivalents si et seulement si il existe une relation* $\approx \subseteq Q_1 \times Q_2$ *vérifiant les propriétés ci-dessus :*

1. $\forall q_1 \in Q_1 \quad \exists q_2 \in Q_2 \quad t.q. \quad q_1 \approx q_2$
2. $\forall q_2 \in Q_2 \quad \exists q_1 \in Q_1 \quad t.q. \quad q_1 \approx q_2$
3. $q_1 \approx q_2 \quad \Rightarrow \quad \forall a \in A_1 \ \forall q_1' \in Q_1 \ q_1 \xrightarrow{a} q_1' \quad \exists q_2' \in Q_2 \ q_2 \xrightarrow{a} q_2' \quad t.q. \quad q_1' \approx q_2'$ *et*

   $\forall a \in A_2 \ \forall q_2' \in Q_2 \ q_2 \xrightarrow{a} q_2' \quad \exists q_1' \in Q_1 \ q_1 \xrightarrow{a} q_1' \quad t.q. \quad q_1' \approx q_2'$

Définition 2.3 (STE : composition parallèle) *Soit* $P_k = (A_k, Q_k, T_k)_{k=1,2}$ *deux STE. La composition parallèle de* P_1 *et* P_2 *avec la synchronisation d'actions* A_s *est le STE noté par* $P_1|[A_s]|P_2 = (A_1 \cup A_2, Q_1 \times Q_2, T)$ *où* T *est défini par les règles suivantes :*

$$\frac{q_1 \xrightarrow{a} q_1' \quad q_2 \xrightarrow{a} q_2' \quad a \in A_s}{(q_1, q_2) \xrightarrow{a} (q_1', q_2')}$$

$$\frac{q_1 \xrightarrow{a_1} q_1' \quad a_1 \notin A_s}{(q_1, q_2) \xrightarrow{a_1} (q_1', q_2)} \qquad \frac{q_2 \xrightarrow{a_2} q_2' \quad a_2 \notin A_s}{(q_1, q_2) \xrightarrow{a_2} (q_1, q_2')}$$

2.2.2 Automates temporisés

Le deuxième modèle dans la hiérarchie IF sont les automates temporisés (AT) avec échéances définis dans [BST97Γ Bor98]. Ce modèle est une généralisation des automates temporisés définis dans [ACD93] par rapport auxquels il présente deux avantages importants :

- La modélisation claire de l'urgenceΓun concept très important dans la description de systèmes avec des contraintes temporelles. Les automates temporisés classiques modélisent l'urgence d'une manière impliciteΓà l'aide des *conditions de progression du temps* appelées invariants. Ce sont des prédicats définis sur les horlogesΓassociés aux états. Dans ce modèleΓon dit que le temps ne peut plus progresser si l'invariant devient ou est faux. Ainsi la seule possibilité d'avancer sera de prendre une transition discrète quiΓ dans un telle situation est devenue urgente. Dans le cas des automates temporisés avec échéancesΓl'urgence est modélisée d'une manière explicite à l'aide d'échéances associées aux transitions. AinsiΓune transition peut être soit urgente (*eager*)Γretardable (*delayable*) ou paresseuse (*lazy*). Les transitions urgentes sont prioritaires par rapport à la progression du temps. Ainsi le temps ne peut pas progresser si une telle transition est franchissable. Par contreΓdans les mêmes circonstancesΓles transitions retardables n'empêche pas le temps de progresser sauf siΓpar une telle progression elles ne sont plus franchissables (les transitions retardables deviennent urgentes au dernier moment). FinalementΓles transitions paresseuses n'imposent aucune contrainte sur la progression du tempsΓelle peuvent aussi bien franchies qu'ignorées à tout moment.
- Les automates temporisés avec échéances sont un modèle compositionnelΓce qui est très important pour la spécification des systèmes réalistes. Les automates temporisés classiques n'ont pas cette propriété. AinsiΓil est relativement facile de concevoir des exemples oùΓpar la simple mise en parallèle des automates temporisésΓon introduit des blocages temporels au niveau de leur composition. Le modèle avec échéances évite par construction ce genre de problèmesΓet il permet de plus la définition de plusieurs opérateurs de composition parallèleΓsans contraintes supplémentaires [Bor98].

Exemple 2.1 (blocage temporel)

Les automates (a) *et* (b) *de la figure 2.2 sont bien conçus : aucun ne comporte des états de blocage temporel. Par contre, l'automate* (c) *obtenu par composition, atteint un état de blocage temporel au moment* $c_1 = c_2 = 2$*, s'il n'a pas quitté l'état initial avant.*

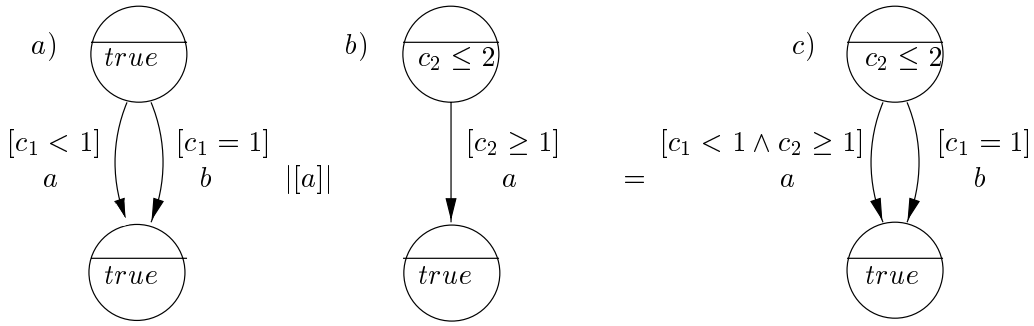

FIG. 2.2 – *Exemple de blocage temporel.*

Définition 2.4 (AT : syntaxe) *Un AT est un tuple (A, C, Q, T) où :*

- *A est un ensemble d'actions ;*
- *C est un ensemble d'horloges. Nous considérons l'ensemble $BExp[C]$ des expressions booléennes construites en utilisant les opérateurs booléens habituels ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $-$) à partir des atomes de la forme $c_i - c_j \# k$ ou $c_i \# k$ où $c_i, c_j \in C$ sont des horloges, $k \in \mathbb{Z}$ est une constante et $\#$ est un opérateur relationnel quelconque ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$). Nous considérons aussi l'ensemble $Rst[C] = \big\{\{c_1 := 0, \ldots\ c_n := 0\} \mid c_i \in C\big\}$ de remises à zéro d'un sous-ensemble d'horloges.*
- *Q est un ensemble d'états ;*
- *$T \subseteq Q \times BExp[C] \times A \times Rst[C] \times Urg \times Q$ est un ensemble de transitions. $Urg = \{eager, delayable, lazy\}$ est l'ensemble des échéances. Une transition $t = (q, h, a, r, u, q') \in T$ sera notée par $q \xrightarrow[u]{h\ a\ r} q'$; $h = guard(t) \in BExp[C]$ est la garde, $r = reset(t) \in Rst[C]$ est la remise à zéro et $u = urgency(t) \in Urg$ est l'échéance de t.*

Quelques notions supplémentaires sont nécessaires pour la définition de la sémantique des automates temporisés. Soit C un ensemble d'horloges. Nous appelons contexte sur les horloges C toute fonction $\gamma : C \to \mathbb{R}$ qui associe une valeur réelle à chaque horloge ; cette valeur sera notée $\gamma(c)$ pour une horloge c.

Soit γ un contexte. Si $t \in \mathbb{R}^+$ est une valeur réelle positive ou nulle nous définissons le contexte $\gamma \oplus t$ obtenu à partir de γ par l'incrémentation des valeurs avec t. Formellement $(\gamma \oplus t)(c) = \gamma(c) + t,\ \forall c \in C$. Si $r \in Rst[C]$ est une remise à zéro d'horloges nous notons par $\gamma[r]$ le contexte obtenu par l'application de remises à zéro dans le contexte γ. Formellement si $r = \{c_1 := 0, \ldots\ c_n := 0\}$ alors $\gamma[r] = \gamma[0/c_1, \ldots 0/c_n]$.

Finalement les contextes peuvent être étendus de manière naturelle aux expressions $e \in BExp[C]$. Nous notons par $\gamma(e)$ la valeur (booléenne) de e dans le contexte γ. Nous allons noter par $[[e]]$ l'ensemble de contextes où e est satisfaite $[[e]] = \{\ \gamma \mid \gamma(e)\ \}$. Une expression $e \in BExp[C]$ est dite *ouverte* si et seulement si $\forall \gamma \in [[e]]\ \exists t > 0$ t.q. $\forall t' \in [0, t)\ \gamma \oplus t' \in [[e]]$. Sinon l'expression est dite *fermée*. Dans ce cas nous allons noter par $e{\downarrow}$ la *frontière* de e i.e

l'expression satisfaite uniquement par les contextes maximaux par rapport à la progression du tempsΓ$[[e\downarrow]] = \{\ \gamma \in [[e]] \mid \forall t > 0\ \exists t' \in [0,t)\ \neg(\gamma \oplus t')(e)\ \}$.

La sémantique des automates temporisés avec échéances est donnée en terme de systèmes de transitions étiquetées. Les états du système de transitions associé à un automate temporisé sont des couples d'un état de contrôle et d'un contexte sur les horloges. Les transitions correspondent soit aux transitions de l'automate soit à la progression du temps.

Définition 2.5 (AT : sémantique) *Soit $P = (A, C, Q, T)$ un AT. La sémantique de P est donnée par le STE $(A^\bullet, Q^\bullet, T^\bullet)$ où :*

- $A^\bullet = A \cup \{time(t) \mid t \in \mathbb{R}^+\}$
- $Q^\bullet = \{(q,\gamma) \mid q \in Q,\ \gamma : C \to \mathbb{R}\}$;
- $T^\bullet$ *est défini par les règles suivantes :*

$$\frac{q \xrightarrow[u]{h\ a\ r} q' \quad \gamma(h)}{(q,\gamma) \xrightarrow{a} (q', \gamma[r])}$$

$$\frac{\forall t' \in [0,t) \quad \left\{\ q \xrightarrow[eager]{h\ a\ r} q' \mid (\gamma \oplus t')(h)\ \right\} = \emptyset \quad \left\{\ q \xrightarrow[delayable]{h\ a\ r} q' \mid (\gamma \oplus t')(h\downarrow)\ \right\} = \emptyset}{(q,\gamma) \xrightarrow{time(t)} (q, \gamma \oplus t)}$$

La première règle définit comment sont exécutées les transitions discrètes de l'automate. AinsiΓchaque transition peut être exécutée dans un contexte donné si sa garde est satisfaite et a comme effet la remise à zéro des horloges correspondantes. La deuxième règle définit comment les transitions temporelles sont exécutéesΓautrement ditΓcomment le temps peut progresser. AinsiΓdans un état et un contexte donnéΓle temps peut avancer d'une valeur t si et seulement si pendant l'intervalle $[0,t)$ il n'y avait pas aucune transition urgente franchissable et aucune transition retardable au but de son échéance.

Nous présentons un opérateur de composition parallèle avec synchronisation pour des automates temporisés. Cet opérateurΓconnu sur le nom de synchronisation ANDΓa été défini dans [Bor98] et fait partie avec les synchronisations MIN et MAXΓd'une famille d'opérateurs définis pour la composition parallèle des automates temporisés avec échéances.

Définition 2.6 (AT : composition parallèle) *Soit $P_k = (A_k, C_k, Q_k, T_k)_{k=1,2}$ deux AT. La composition parallèle de P_1 et P_2 avec la synchronisation d'actions A_s est l'AT $P_1|[A_s]|P_2 = (A_1 \cup A_2, C_1 \cup C_2, Q_1 \times Q_2, T)$ où T est défini par les règles suivantes :*

$$\frac{q_1 \xrightarrow[u_1]{h_1\ a\ r_1} q_1' \quad q_2 \xrightarrow[u_2]{h_2\ a\ r_2} q_2' \quad a \in A_s}{(q_1, q_2) \xrightarrow[u_1 \circ u_2]{h_1 \wedge h_2\ a\ r_1 \sqcup r_2} (q_1', q_2')}$$

$$\frac{q_1 \xrightarrow[u_1]{h_1\ a_1\ r_1} q_1' \quad a_1 \notin A_s}{(q_1, q_2) \xrightarrow[u_1]{h_1\ a_1\ r_1} (q_1', q_2)}$$

$$\frac{q_2 \xrightarrow[u_2]{h_2\ a_2\ r_2} q_2' \quad a_2 \notin A_s}{(q_1, q_2) \xrightarrow[u_2]{h_2\ a_2\ r_2} (q_1, q_2')}$$

La composition est réalisée respectivement par synchronisation de transitions ayant les actions mentionnées dans la liste de synchronisation et par entrelacement des autres. Les transitions obtenues par synchronisation ont comme garde le et-logique des gardes $(h_1 \wedge h_2)$Γ remettent à zéro les horloges remises dans l'une et l'autre des composantes $(r_1 \sqcup r_2)$Γet sont étiquetées par la même action pour favoriser un rendez-vous multiple. Les échéances sont composées dans la manière précisée par la table suivante :

$u_1 \circ u_2$	*eager*	*delayable*	*lazy*
eager	*eager*	*eager*	*eager*
delayable	*eager*	*delayable*	–
lazy	*eager*	–	*lazy*

A l'exception de synchronisation entre une transition retardable et une paresseuseΓl'échéance est toujours bien définie par composition. Dans le cas mentionnéΓl'échéance composée n'est plus de ces trois types ; si une telle situation se présentent dans la composition il est nécessaire (si possible) de remplacer la transition retardable par deux transitionsΓune urgente et une paresseuse qui simulent le même comportementΓet de recomposer ensuite.

Pour la clarté de la présentationΓnous avons considéré un modèle des automates temporisés relativement simple. Un certain nombre d'extensions sont possiblesΓsans augmenter la complexité ou l'expressivité du modèle :

- il est possible de considérer un modèle avec des opérations de remise à des valeurs autres que zéro ;
- il est possible de considérer des horloges décroissantes ; en fait une telle horloge T peut être toujours simulée à l'aide de son copie inverse $-T$;
- il est toujours possible d'ajouter des conditions de progression du temps aux étatsΓen gardant le risque d'introduire ensuite des blocages temporels par composition parallèle ;

2.2.3 Automates temporisés communicants

Le troisième modèle dans la hiérarchie IF est le modèle des automates temporisés communicants (ATC) avec à la fois communication synchrone par rendez-vous et communication asynchrone à travers des files d'attente. Ce modèle étend naturellement les modèles précédents qui sont basés uniquement sur une communication par rendez-vous.

Quelques éléments syntaxiques nouveaux sont ajoutés pour la définition des ces automates. NotammentΓnous distinguons deux grandes primitives de communication: les files d'attenteΓ pour réaliser la communication asynchrone et les portes de synchronisationΓpour réaliser la communication par rendez-vous. Les actions de l'automate seront soit asynchrones (un ensemble de réceptions et d'émissions vers les files d'attente) soit synchrones (rendez-vous sur l'une de portes).

Définition 2.7 (ATC : syntaxe) *Un ATC est un tuple* (S, B, G, C, Q, T) *où :*

- S *est un ensemble de signaux ;*
- B *est un ensemble de files d'attente qui transportent des signaux de* S*. Nous définissons l'ensemble d'entrées* $In[S,B] = \left\{\{b_1^{in}?s_1, \ldots\ b_n^{in}?s_n\} \mid b_i^{in} \in B,\ s_i \in S\right\}$*. Nous définissons l'ensemble de sorties* $Out[S,B] = \left\{\{b_1^{out}!w_1, \ldots\ b_m^{out}!w_m\} \mid b_j^{out} \in B,\ w_j \in S^*\right\}$ *et nous notons par* $Async[S,B] = In[S,B] \times Out[S,B]$ *l'ensemble d'actions asynchrones ;*
- G *est un ensemble de portes de synchronisation. Nous notons par* $Sync[G] = \{g! \mid g \in G\}$ *l'ensemble d'actions synchrones aux portes* G *;*
- C *est un ensemble d'horloges ; nous notons par* $BExp[C]$ *l'ensemble des expressions booléennes définies à base de* C *et par* $Rst[C]$ *l'ensemble des remises à zéro définies sur* C *de la même manière que pour les automates temporisés ;*
- Q *est un ensemble d'états ;*
- $T \subseteq Q \times BExp[C] \times A[S,B,G] \times Rst[C] \times Urg \times Q$ *est un ensemble de transitions où* $A[S,B,G] = Async[S,B] \cup Sync[G]$ *est l'ensemble d'actions asynchrones ou synchrones, qui peuvent être associées aux transitions.*

  Nous utilisons les mêmes notations $source(t)$*,* $guard(t)$*, etc pour accéder aux informations d'une transition* t*. Nous notons par* $gate(t)$ *la porte de synchronisation si* t *contient une action synchrone. Sinon, nous notons par* $input(t)$ *l'entrée et par* $output(t)$ *la sortie associées à* t*.*

Quelques définitions supplémentaires sont nécessaires pour définir la sémantique des automates temporisés communicants. Soient S un ensemble de signaux et B un ensemble de files d'attente. Nous définissons les contextes sur les files B comme des fonctions $\delta : B \to S^*$ qui associent une séquence de signaux à chaque file ; pour une file b cette séquence sera notée par $\delta(b)$.

La sémantique des automates temporisés communicants est donnée en terme d'automates temporisés. A chaque automate temporisé communicant on peut lui associer un automate

temporisé simple dont les états sont des couples état de contrôle et contexte sur les files d'attente. Les transitions sont dérivées de transitions de l'automate initial.

Définition 2.8 (ATC : sémantique) *Soit $P = (S, B, G, C, Q, T)$ un ATC. La sémantique de P est donnée par l'AT $(A^\bullet, C, Q^\bullet, T^\bullet)$ où :*

- $A^\bullet = Sync[G] \cup Async[S, B]$
- $Q^\bullet = \{(q, \delta) \mid q \in Q,\ \delta : Q \to S^*\}$
- $T^\bullet$ *est défini par les règles suivantes :*

$$\frac{q \xrightarrow[u]{h\ g!\ r} q'}{(q,\delta) \xrightarrow[u]{h\ g!\ r} (q,\delta)}$$

$$\frac{\begin{array}{c} q \xrightarrow[u]{h\ \ b_1^{in}?s_1, \ldots\ b_n^{in}?s_n\ \ b_1^{out}!w_1, \ldots b_m^{out}!w_m\ \ r} q' \\ \forall i = 1, n\ \ \delta(b_i^{in}) = s_i.w_i'\ \ s_i \in S\ \ w_i' \in S^* \\ \delta'' = \delta[w_1'/b_1^{in}, \ldots\ w_n'/b_n^{in}] \\ \delta' = \delta''[\delta''(b_1^{out}).w_1/b_1^{out}, \ldots\ \delta''(b_m^{out}).w_m/b_m^{out}] \end{array}}{(q,\delta) \xrightarrow[u]{h\ \ b_1^{in}?s_1, \ldots\ b_n^{in}?s_n\ \ b_1^{out}!w_1, \ldots b_m^{out}!w_m\ \ r} (q',\delta')}$$

Dans la suite nous allons présenter un certain nombre d'extensions intéressantes au modèle ATC. Ces extensions visent notamment deux aspectsΓd'une part de considérer d'autres types de files d'attentes et d'autre part d'enrichir le langage d'actions utilisées dans ce modèle.

Les files d'attente sont caractérisées par un nombre de paramètres comme par exemple la manière de stockage de signaux (ordonné ou nonΓavec ou sans duplicationΓ...)Γla capacité (bornée ou non-bornée)Γla fiabilité (avec ou sans pertes de signaux)Γetc. Ces paramètres ont été instanciés par défaut dans le modèle ATC présenté auparavantΓmais ils peuvent être modifiés pour obtenir d'autres modes de communication.

Files d'attente non-ordonnées

Dans les ATCΓles files d'attente utilisent une discipline FIFO (*First-In First-Out*) pour gérer les signaux. En faitΓce choix représente le mieux ce qu'on attend d'une ligne de communi-

cation: la réception sans perte et dans l'ordre d'émission. Cependant cette propriété n'est pas vérifiée par tous les protocoles : le protocole UDP de la couche transport ne garantit pas l'ordre d'arrivée des messages. D'où l'idée de relâcher la contrainte sur la préservation de l'ordre sur les signaux. Cela revient à remplacer les queues de signaux par des multi-ensembles (ou sacs ou *bags*) ; formellement on considère les contextes sur les files comme des applications $\delta : B \to S \to \mathbb{Z}^+$ qui associent à chaque file le nombre d'occurrences pour chacun de signaux. Avec cette interprétation les prémisses portant sur les contenus des files dans la deuxième règle de sémantique deviennent :

$$\forall i = 1, n \quad \delta(b_i^{in})(s_i) \geq 1$$

$$\delta''(b)(s) = \begin{cases} \delta(b)(s) - 1 & \text{si } \exists i \ b = b_i^{in} \text{ et } s = s_i \\ \delta(b)(s) & \text{sinon} \end{cases}$$

$$\delta'(b)(s) = \begin{cases} \delta''(b)(s) + nb(s, w) & \text{si } \exists j \ b = b_j^{out} \text{ et } w = w_j \\ \delta''(b)(s) & \text{sinon} \end{cases}$$

Une autre possibilité que nous avons envisagée est l'utilisation de files d'attente sous forme d'ensemble de signaux. Dans ce cas un contexte sur les files est une application $\delta : B \to \mathcal{P}(S)$ qui associe à chaque file les signaux qu'il contient. Les prémisses portant sur les contenus des files deviennent :

$$\forall i = 1, n \quad s_i \in \delta(b_i^{in})$$

$$\delta''(b) = \delta(b) \setminus \{s \mid \exists i \ b = b_i^{in}, \ s = s_i\}$$

$$\delta'(b) = \delta''(b) \cup \{s \mid \exists j \ b = b_j^{out}, \ s \in w_j\}$$

Files d'attente bornés ou non-bornées

Les spécifications des protocoles de communication imposent souvent des restrictions sur la taille maximale des files d'attente utilisées. Dans beaucoup des cas les limites sont relativement grandes pour simuler effectivement une capacité pratiquement infinie de stockage ; dans telles situations l'utilisation d'un modèle de files non-bornées pour la spécification et la vérification est le mieux adapté. Par contre il existe des cas où les limites sont strictes et doivent être prises en compte. Le problème qui se pose concerne les *débordements* : comment traiter les transitions dont la sortie fait déborder la taille maximale permise d'une file? Plusieurs solutions se présentent par exemple :

- elles peuvent être considérées comme non-franchissables ;
- elles peuvent être franchies avec la perte de signaux supplémentaires.

Files d'attente avec ou sans pertes

Une autre propriété à prendre en compte pour les files d'attente est leur fiabilité. L'hypothèse que les files peuvent perdre des signaux est complètement naturelle pour modéliser certaines lignes de communication physiques. De plus cette hypothèse peut réduire la complexité du problème de vérification. Plus formellement la prise en compte des pertes des messages peut se modéliser par la règle supplémentaire suivante ou $\xrightarrow{loose}$ dénote la relation de perte sur les contextes :

$$\frac{\delta \xrightarrow{loose} \delta'' \quad (q,\delta'') \xrightarrow[u]{a} (q',\delta''') \quad \delta''' \xrightarrow{loose} \delta'}{(q,\delta) \xrightarrow[u]{a} (q',\delta')}$$

La relation $\xrightarrow{loose}$ est définie en fonction de la catégorie de files d'attente ; par exemple dans le cas où les contenus sont des séquences de signaux et si $\preceq$ dénote l'inclusion sur les séquences on a :

$$\delta \xrightarrow{loose} \delta' \quad - \quad \forall b \in B \ \delta'(b) \preceq \delta(b)$$

Filtrages sur les files

Dans le modèle ATC le franchissement d'une transition asynchrone repose sur le fait que certains signaux existent *exactement* en tête des files. Si ce n'est pas le cas la transition n'est pas franchissable.

Un degré plus grand de flexibilité peut être obtenu à l'aide des filtres sur les contenus des files associés aux transitions asynchrones. Le rôle de ces filtres sera de masquer détruire ou renommer certaines parties des contenus. Ainsi le franchissement de chaque transition sera décidé dans un contexte *filtré* et non-plus dans le contexte global. Concrètement nous avons expérimenté des filtres visant respectivement à sauver ou à détruire le plus longue préfixe constitué d'un ensemble de signaux donné. La sémantique précise d'utilisation est présentée par la suite.

Formellement un filtre est une application $\theta : B \to S \to \{save, discard, none\}$. Il associe à chaque file une manière de filtrer chacun de signaux. Ainsi un signal sera soit sauvé ($save$) soit détruit ($discard$) soit non filtré ($none$). La prise en compte d'un filtre θ lorsqu'on exécute les transitions comportant des actions asynchrones est précisée par les règles supplémentaires suivantes :

$$\frac{\theta(b)(s) = save \quad \delta(b) = s.w \qquad (q, \delta[w/b]) \xrightarrow[u]{a} (q', \delta') \quad \delta'(b) = w'}{(q, \delta) \xrightarrow[u]{a} (q', \delta'[s.w'/b])}$$

$$\frac{\theta(b)(s) = discard \quad \delta(b) = s.w \qquad (q, \delta[w/b]) \xrightarrow[u]{a} (q', \delta')}{(q, \delta) \xrightarrow[u]{a} (q', \delta')}$$

Opérations sur les files

L'utilisation d'autres opérations d'accès ou de modification de files d'attenteΓen plus d'entrées et de sortiesΓsemble d'être un besoin complètement naturel. Par exempleΓon peut argumenter sur la nécessité des opérations comme le test d'une file vide ou pleineΓle nombre de signaux stockésΓl'accès aux i-ème signal (si la file est ordonnée)Γle vidage ou la copie d'une file dans une autreΓetc. En toute généralitéΓles files peuvent être considérées comme des instances de certains types abstraits sur lesquels ont été définis toutes ces opérationsΓ avec une sémantique bien-précise.

L'utilisation de ces primitives à l'intérieur des transitionsΓdans les gardes ou les corpsΓou encore pour contraindre l'application de certains filtres augmente la puissance d'expression du modèle ATC. MaisΓd'une autre partΓces opérations imposent des contraintes supplémentaires pour la validation. AinsiΓà part les contraintes d'efficacité sur la représentation des contextesΓ on doit prendre en compte le fait que ces opérations puissent aussi être effectuéesΓavec des coûts raisonnables pour la simulation.

Dans la suiteΓnous présentons un opérateur de composition parallèle avec synchronisation pour des automates temporisés communicants. Il est similaire à l'opérateur défini sur les automates temporisés : la synchronisation s'applique uniquement aux transitions comportant des actions synchrones aux portes spécifiées dans un ensemble de synchronisation et toutes les autres transitions sont exécutées par entrelacement.

Définition 2.9 (ATC : composition parallèle) *Soit* $P_k = (S_k, B_k, G_k, C_k, Q_k, T_k)_{k=1,2}$ *deux ATC. La composition parallèle de* P_1 *et* P_2 *avec synchronisation de portes* G_s *est l'ATC* $P_1|[G_s]|P_2 = (S_1 \cup S_2, B_1 \cup B_2, G_1 \cup G_2, C_1 \cup C_2, Q_1 \times Q_2, T)$ *où* T *est défini par les règles suivantes :*

$$\frac{q_1 \xrightarrow[u_1]{h_1\ a\ r_1} q_1' \quad q_2 \xrightarrow[u_2]{h_2\ a\ r_2} q_2' \quad a \in Sync[G_s]}{(q_1,q_2) \xrightarrow[u_1 \circ u_2]{h_1 \wedge h_2 \quad a \quad r_1 \sqcup r_2} (q_1',q_2')}$$

$$\frac{q_1 \xrightarrow[u_1]{h_1\ a_1\ r_1} q_1' \quad a_1 \notin Sync[G_s]}{(q_1,q_2) \xrightarrow[u_1]{h_1\ a_1\ r_1} (q_1',q_2)} \qquad \frac{q_2 \xrightarrow[u_2]{h_2\ a_2\ r_2} q_2' \quad a_2 \notin Sync[G_s]}{(q_1,q_2) \xrightarrow[u_2]{h_2\ a_2\ r_2} (q_1,q_2')}$$

2.2.4 Automates temporisés communicants avec des données

Le dernier modèle de la hiérarchie IF sont les automates temporisés communicants étendus avec des données (ATCD). Ces automates sont obtenus à partir des automates temporisés communicants en ajoutant un ensemble de variables typéesΓdont les valeurs peuvent être respectivement testéesΓmodifiéesΓéchangées à travers de synchronisations ou envoyées comme paramètres de signaux de communication asynchrone.

Définition 2.10 (ATCD : syntaxe) *Un ATCD est un tuple (D, X, S, B, G, C, Q, T) où :*

- *D est un ensemble de types de données. Nous supposons que l'on dispose d'un ensemble d'opérations définies sur chacun, et nous notons par $domain(d)$ l'ensemble de valeurs du type $d \in D$;*
- *X est un ensemble de variables discrètes typées. Nous notons par $type(x) \in D$ le type de la variable x. Nous notons par $Exp[X]$ ($BExp[X]$) l'ensemble des expressions (booléennes) construites à partir de variables de X et d'opérations existantes sur les types ; nous supposons que chaque expression a un type unique qui peut être calculé statiquement et qui sera noté par $type(e)$, pour toute $e \in Exp[X]$. Finalement, nous notons par $Rst[X] = \big\{\{x_1 := e_1, \ldots\ x_n := e_n\} \mid x_i \in X,\ e_i \in Exp[X]\big\}$ l'ensemble des affectations aux variables de X, des fonctions partielles définies sur l'ensemble de variables avec valeurs dans l'ensemble d'expressions ;*
- *S est un ensemble de signaux typés. Nous notons par $type(s) \in D^*$ le profile du signal $s \in S$, ainsi toute réception ou émission de s devrait être paramétrée par une liste de valeurs de ce type ;*
- *B est un ensemble de files d'attente. Nous considérons l'ensemble d'entrées $In[X, S, B]$ défini comme $\big\{\{b_1^{in}?s_1(\overrightarrow{x_1}), \ldots\ b_n^{in}?s_n(\overrightarrow{x_n})\} \mid b_i^{in} \in B,\ s_i \in S,\ \overrightarrow{x_i} \in X^*\big\}$; dans la*

construction $b_i^{in}?(\overrightarrow{x_i})$, $\overrightarrow{x_i}$ *désigne une liste des variables pour récupérer les paramètres transportés par le signal* s_i. *Nous supposons que les liste* $\overrightarrow{x_i}$ *sont deux à deux disjointes et contiennent des variables deux à deux distinctes. L'ensemble de sorties* $Out[X,S,B]$ *est défini comme* $\left\{\{b_1^{out}!W_1,\dots\ b_m^{out}!W_m\} \mid b_j^{out} \in B,\ W_j = s_{j1}(\overrightarrow{e_{j1}})\dots s_{jm_j}(\overrightarrow{e_{jm_j}}) \in (S \times Exp[X]^*)^*\right\}$; *là aussi, l'émission d'un signal* s_{jk} *est augmentée par une liste des expressions* $\overrightarrow{e_{jk}}$, *ses paramètres. Finalement, l'ensemble d'actions asynchrones est* $Async[X,S,B] = In[X,S,B] \times Out[X,S,B]$;

- G *est un ensemble de portes de synchronisation. Nous notons* $type(g) \in D^*$ *le profile de la porte* $g \in G$. *Toute synchronisation sur* g *doit être paramétrée par un échange de valeurs de ce type. Nous notons par* $Sync[G,X] = \{g\ o_1 \dots\ o_n \mid g \in G,\ o_i = !e_i \in Exp[X]$ *ou* $o_i = ?x_i \in X\}$ *l'ensemble d'actions synchrones aux portes* G. *Dans ce cas aussi, on suppose que les variables* x_i *présentes dans la liste des offres sont deux à deux distinctes ;*

- C *est un ensemble d'horloges ; nous notons par* $BExp[C]$ *l'ensemble des expressions booléennes et par* $Rst[C]$ *l'ensemble des remises à zéro, comme précédemment;*

- Q *est un ensemble d'états ;*

- $T \subseteq Q \times BExp[C] \times BExp[X] \times A[X,S,B,G] \times Rst[C] \times Rst[X] \times Urg \times Q$ *est un ensemble de transitions,* $A[X,S,B,G] = Async[X,S,B] \cup Sync[X,G]$. *Chaque transition contient, en plus de la garde et de la remise à zéro d'horloges, une garde et une affectation de variables discrètes, une action de communication asynchrone ou synchrone et l'échéance.*

Nous définissons comme contexte sur les variables de X toute application $\rho : X \to \bigcup_{d \in D} domain(d)$ qui associe à toute variable x la valeur $\rho(x)$ dans le domaine de x. Les contextes sur les variables peuvent être étendus d'une manière naturelle à l'ensemble des expressions $Exp[X]$; nous notons par $\rho(e)$ la valeur de l'expression $e \in Exp[X]$ dans le contexte ρ. Soit $r \in Rst[X]$ une affectation partielle de variables de X. Nous notons par $\rho[r]$ le nouvel contexte obtenu à partir du contexte ρ en appliquant l'affectation r. FormellementΓsi $r = \{x_1 := e_1, \dots\ x_n := e_n\}$ alors $\rho[r] = \rho[\rho(e_1)/x_1, \dots\ \rho(e_n)/x_n]$.

La sémantique des automates temporisés communicants étendus est définie en termes d'automates temporisés communicants simples. AinsiΓpour chaque ATCD on construit un ATC où les signaux et les portes de communications sont aplatisΓles états sont des couples – état de contrôle et contexte sur les variables. On a la définition suivante.

Définition 2.11 (ATCD : sémantique) *Soit* $P = (D,S,B,G,C,X,Q,T)$ *un ATCD. La sémantique de* P *est donnée par l'ATC* $(S^\bullet, B, G^\bullet, C, Q^\bullet, T^\bullet)$ *où :*

- $S^\bullet = \{s(v_1,\dots,v_n) \mid s \in S,\ type(s) = (d_1,\dots,d_n),\ \forall i = 1,n\ v_i \in domain(d_i)\}$
- $G^\bullet = \{g\ !v_1 \dots !v_n \mid s \in S,\ type(g) = (d_1,\dots,d_n),\ \forall i = 1,n\ v_i \in domain(d_i)\}$
- $Q^\bullet = \{(q,\rho) \mid q \in Q,\ \rho : X \to \bigcup_{d \in D} domain(d)\}$

– $T^\bullet$ *est défini par les règles suivantes :*

$$\frac{q \xrightarrow[u]{h^c \ h^x \quad g \ o_1 \ldots o_n \quad r^c \ r^x} q'}{\rho(h^x) \quad v_i = \rho(e_i) \ \forall o_i = !e_i \quad \rho' = \rho[\{x_j := v_j \mid \forall o_j = ?x_j\}]}$$

$$\frac{}{(q,\rho) \xrightarrow[u]{h^c \quad g \ !v_1 \ldots !v_n \quad r^c} (q', \rho'[r^x])}$$

$$\frac{q \xrightarrow[u]{h^c \ h^x \quad b_1^{in}?s_1(\overrightarrow{x_1}) \ldots \ b_n^{in}?s_n(\overrightarrow{x_n}) \quad b_1^{out}!W_1 \ldots \ b_m^{out}!W_m \quad r^c \ r^x} q'}{\rho(h^x) \quad \rho' = \rho[\overrightarrow{v_1}/\overrightarrow{x_1}, \ldots \ \overrightarrow{v_n}/\overrightarrow{x_n}]}$$

$$\frac{}{(q,\rho) \xrightarrow[u]{h^c \quad b_1^{in}?s_1(\overrightarrow{v_1}) \ldots \ b_n^{in}?s_n(\overrightarrow{v_n}) \quad b_1^{out}!\rho'(W_1) \ldots \ b_m^{out}!\rho'(W_m) \quad r^c} (q, \rho'[r^x])}$$

La première règle concerne les transitions qui comportent des actions synchrones. Dans un contexte donné ρΓelles sont franchies si et seulement si la garde portant sur les variables discrètes h^x est satisfaite. Elles ont comme effet d'abord l'affectation des variables $?x_j$ présentes dans la liste des offres à des valeurs quelconques v_jΓpuis l'application de l'affectation discrète r^x dans le nouvel contexte ρ'. La deuxième règle concerne les transitions asynchrones. Elles aussi sont franchies uniquement si leur garde discrète est satisfaiteΓpuis elles affectent les variables présentes dans les entrées $\overrightarrow{x_i}$ à des valeurs quelconques $\overrightarrow{v_i}$Γpuis interprètent les séquences de sorties W_j et exécutent l'affectation discrète r^x.

FinalementΓnous présentons un opérateur de composition parallèle avec synchronisation pour les automates temporisés communicants avec des données. Comme dans le cas des ATCΓcet opérateur synchronise respectivement les transitions qui comportent des actions synchrones aux portes de la liste de synchronisation et exécute indépendammentΓpar entrelacementΓ les autres. A cause des variables discrètesΓla synchronisation est une opération relativement complexe et nécessite des restrictions supplémentaires à l'égard de la forme des composantes des transitions concernées.

D'abord nous introduisons quelques notations auxiliaires. Soit o_1 et o_2 deux offres comportant des expressions ou des variables d'un même type. Nous notons par $h(o_1, o_2)$ la condition nécessaire pour pouvoir synchroniser ces offresΓpar $o(o_1, o_2)$ la nouvelle offre et par $r(o_1, o_2)$ l'affectation qui résulte suite de leur synchronisationΓformellement :

		$?x_2$	$!e_2$
$h(o_1, o_2)$		$true$	$true$
$o(o_1, o_2)$	$?x_1$	$?x_2$	$!e_2$
$r(o_1, o_2)$		$\{x_1 := x_2\}$	$\{x_1 := e_2\}$
		$true$	$e_1 = e_2$
	$!e_1$	$!e_1$	$!e_1$
		$\{x_2 := e_1\}$	$\{\}$

Ces opérations sont étendues sur les listes d'offres ayant la même longueur $(n)\Gamma$des types équivalents et de plusΓsi les ensembles des variables affectées globalement par chaque liste sont disjointes :

$$h(\overrightarrow{o_1}, \overrightarrow{o_2}) = \bigwedge_{i=1}^{n} h(o_{1i}, o_{2i}) \qquad o(\overrightarrow{o_1}, \overrightarrow{o_2}) = o(o_{1i}, o_{2i})_{i=1,n} \qquad r(\overrightarrow{o_1}, \overrightarrow{o_2}) = \bigsqcup_{i=1}^{n} r(o_{1i}, o_{2i})$$

Soit r_1, r_2 des affectations des variables discrètes. Nous allons notée par $r_1 \sqcup r_2$ l'affectation collatérale de r_1 et r_2 : elle est définie uniquement si les ensembles des variables affectées par chacune sont disjointes et correspond à l'union au sens classique de r_1 et r_2. Nous allons noter par $r_1; r_2$ l'affectation séquentielle de r_1 puis r_2 ; cette opération est toujours définie et le résultat peut être remis sous la forme d'une affectation parallèle r.

Définition 2.12 (ATCD : composition parallèle) *Soit $P_k = (D_k, X_k, S_k, B_k, G_k, C_k, Q_k, T_k)_{k=1,2}$ deux ATCD. La composition parallèle de P_1 et P_2 avec synchronisation de portes G_s est l'ATCD $P_1|[G_s]|P_2 = (D_1 \cup D_2, S_1 \cup S_2, B_1 \cup B_2, G_1 \cup G_2, C_1 \cup C_2, X_1 \cup X_2, Q_1 \times Q_2, T)$ où T est défini par les règles suivantes :*

$$\frac{q_1 \xrightarrow[u_1]{h_1^c \; h_1^x \;\; g \;\; \overrightarrow{o_1} \;\; r_1^c \; r_1^x} q_1' \quad q_2 \xrightarrow[u_2]{h_2^c \; h_2^x \;\; g \;\; \overrightarrow{o_2} \;\; r_2^c \; r_2^x} q_2' \quad g \in G_s}{(q_1, q_2) \xrightarrow[u_1 \circ u_2]{h_1^c \wedge h_2^c \;\; h_1^x \wedge h_2^x \wedge h(\overrightarrow{o_1}, \overrightarrow{o_2}) \;\; g \;\; o(\overrightarrow{o_1}, \overrightarrow{o_2}) \;\; r_1^c \sqcup r_2^c \;\; r(\overrightarrow{o_1}, \overrightarrow{o_2}); (r_1^x \sqcup r_2^x)} (q_1', q_2')}$$

$$\frac{q_1 \xrightarrow[u_1]{h_1^c \; h_1^x \;\; a_1 \;\; r_1^c \; r_1^x} q_1' \quad a_1 \notin Sync[G_s, X_1]}{(q_1, q_2) \xrightarrow[u_1]{h_1^c \; h_1^x \;\; a_1 \;\; r_1^c \; r_1^x} (q_1', q_2)}$$

$$\frac{q_2 \xrightarrow[u_2]{h_2^c\ h_2^x\ \ a_2\ \ r_2^c\ r_2^x} q_2' \quad a_2 \notin Sync[G_s, X_2]}{(q_1, q_2) \xrightarrow[u_2]{h_2^c\ h_2^x\ \ a_2\ \ r_2^c\ r_2^x} (q_1, q_2')}$$

La première règle définit la synchronisation de transitions qui comportent des actions synchrones. La transition composée comporte une garde obtenue par le et-logique des gardes respectivement temporisées $h_1^c \wedge h_2^c$ et discrètes $h_1^x \wedge h_2^x$. La garde discrète est renforcée par la condition de synchronisation des offres $h(\overrightarrow{o_1}, \overrightarrow{o_2})$. L'action est synchrone à la même porteΓ avec la liste des offres obtenue par la synchronisation des deux listes initiales $o(\overrightarrow{o_1}, \overrightarrow{o_2})$. Finalement$\Gamma$la nouvelle affectation discrète est obtenue par composition séquentielle de l'affectation résultée par synchronisation $r(\overrightarrow{o_1}, \overrightarrow{o_2})$ et l'affectation collatérale $r_1^x \sqcup r_2^x\Gamma$ si elle existe. Les transitions asynchrones sont exécutées par entrelacement.

Un certain nombre d'extensions sont encore nécessaires au modèle ATCD afin de le rendre équivalent aux processus IF. MaisΓelles touchent généralement la forme et non pas le fond de ce modèle.

Par exempleΓon peut considérer des gardes supplémentaires portant sur les valeurs reçues par une entrée asynchrone ou par une synchronisation. FormellementΓune telle garde peut être prise en compte naturellement dans les règles de définition de la sémantique. L'exécution d'une transition sera encore contrainte par la satisfaction de cette garde supplémentaire dans le contexte intermédiaire ρ'.

En outreΓon peut distinguer des états stables ou instables. Cette propriété peut se modéliser à l'aide d'un prédicat *stable* portant sur les états d'un ATCD. La stabilité concerne uniquement l'opérateur de composition parallèle défini sur les ATCD. D'une partΓla stabilité se propage naturellement aux états de l'ATCD produitΓ $stable((q_1, q_2)) = stable(q_1) \wedge stable(q_2)$. D'autre part$\Gamma$l'exécution d'une transition non-synchronisable sera contrainte par la stabilité de l'état du départΓ par rapport à la stabilité de l'état du partenaire. Les règles d'exécution des transitions non-synchronisables changent alors vers :

$$\frac{q_1 \xrightarrow[u_1]{h_1^c\ h_1^x\ \ a_1\ \ r_1^c\ r_1^x} q_1' \quad a_1 \notin Sync[G_s, X_1] \qquad \neg stable(q_1)\ \vee\ stable(q_2)}{(q_1, q_2) \xrightarrow[u_1]{h_1^c\ h_1^x\ \ a_1\ \ r_1^c\ r_1^x} (q_1', q_2)}$$

$$\frac{q_2 \xrightarrow[u_2]{h_2^c\ h_2^x\ \ a_2\ \ r_2^c\ r_2^x} q_2' \quad a_2 \notin Sync[G_s, X_2] \qquad \neg stable(q_2)\ \vee\ stable(q_1)}{(q_1, q_2) \xrightarrow[u_2]{h_2^c\ h_2^x\ \ a_2\ \ r_2^c\ r_2^x} (q_1, q_2')}$$

2.3 Discussion

Une première remarque concerne l'ordre dans lequel nous avons introduit les concepts primitives dans la définition de la sémantique. En faitΓnous avons une seule contrainte concernant l'introduction du temps : le franchissement des transitions temporelles dans le modèle repose sur le franchissement *effectif* des transitions non-temporelles ainsi que sur leurs échéances. DoncΓla progression du temps ne peut pas être définie tant que le franchissement de transitions non-temporelles n'est pas complètement décidé. C'est le raison pour lequel le temps n'est pas une primitive tout à fait orthogonale au modèle et nécessitent un traitement à part. Par contreΓentre les files et les variables le choix a été plus ou moins arbitraire. Nous avons pu aussi bien commencer par l'introduction des variablesΓet ensuite par l'introduction des files. CependantΓnous avons préféré l'inverse car cela nous a permis de mieux illustrer les problèmes liés aux files et à la communication asynchroneΓen faisant abstraction de données.

Une deuxième remarque concerne les limites du modèle considéré. En faitΓce modèle est complètement statique. Il ne permet aucune forme de création dynamique de processusΓ filesΓ variables ou autres. Cette restriction est parfois très limitativeΓ d'ailleurs elle nous empêche de modéliser la création dynamique de processus ou l'appel de procédures récursives de SDL. CependantΓcette restriction a une explication pragmatique : nous avons défini un modèle adéquate pour faciliter l'application des techniques d'analyses statiquesΓvues comme l'ingrédient essentiel pour aboutir dans la validation automatique des systèmes de grande taille.

Le chapitre suivant présente une première application de IF i.eΓla traduction de spécifications SDL vers des spécifications IF. Cette application a un double rôle. D'une partΓelle peut être vue comme une évaluation de l'expressivité du modèle IFΓface à un vrai langage de spécification. D'autre partΓla traduction peut être vue comme étant la proposition d'une nouvelle sémantique pour un sous-ensemble de SDLΓà base du modèle intermédiaire IF.

Chapitre 3

Traduction

Ce chapitre présente la traduction de SDL vers IF.

SDL est un *langage de spécification* très expressifΓcontenant beaucoup de constructions syntaxiquesΓplus ou moins compliquées. L'explication est simple : il a été conçu pour *spécifier* de manière lisible et compacte une grande variété de protocolesΓet plus généralementΓde systèmes asynchrones.

IF est un *modèle intermédiaire* pour décrire le fonctionnement des systèmes asynchrones. Les aspects langage sont moins évoluésΓles constructions syntaxiques sont simples et explicites et décrivent notamment des entités manipulés concrètement lorsque un système s'exécute.

La traduction n'est donc pas une simple mise en forme en syntaxe IF de systèmes SDL car même si les deux décrivent plus ou moins la même choseΓils le font à des niveaux d'abstraction différents. Le rôle de cette traduction est d'expliciter une partie de la sémantique dynamique des spécifications SDL. Elle peut être ainsi vue comme une tentative de définir une nouvelle sémantique opérationnelle d'une sous-classe de spécifications SDLΓbasée sur IF et sa sémantique.

CependantΓen raison des limites d'expressivité de IFΓla traduction n'est définie que pour une sous-classe de spécifications SDL. Des restrictions concernent notamment les aspects *dynamiques* du langage. La plus limitative est l'absence de création dynamique d'instances i.eΓ on ne traduit que des systèmes SDL complètement statiques. De plusΓcertaines sorties sans destinataire explicite ou encore l'accès à certaines variables partagées ne sont pas traduits. FinalementΓnous ne traitons pas les canaux avec délai. L'avantage de ces choix est que la complexité du programme IF obtenu par traduction automatique est comparable à la complexité du programme SDL initial.

La traduction des types de données n'est pas considérée dans ce travail. C'est un problème orthogonalΓqui peut être traité indépendamment de la traduction du comportement. Les types de données de SDL sont des types abstraits algébriques ACT-ONE [EM85]. La traduction vers d'autres formalismesΓet plus particulièrement la génération automatique de codeΓ a été longtemps étudiée. Des techniques efficaces ont été développées et sont actuellement implémentées par des outils de traduction performants.

La traduction est réalisée en deux phasesΓexpansion et génération. D'abordΠla phase d'expansion vise à simplifier les spécifications SDL en remplaçant certaines constructions syntaxiques complexes par d'autres plus simples. EnsuiteΓla phase de génération consiste à synthétiser un système IF à partir d'une spécification SDL ainsi simplifiée.

3.1 Expansion

La phase d'expansion vise à simplifier les spécifications SDL en remplaçant certaines constructions syntaxiques complexes par d'autres plus simples équivalentes. Elle comporte les sous-phases d'expansion des blocsΓinstanciation des processusΓcomposition de services et intégration de procédures. Ces sont des opérations similaires à celles utilisées pendant la compilation de langages impératifs à structure de blocs et nécessite la mise en œuvre de techniques très classiques de compilation comme par exemple la liaison d'identificateurs et la gestion de noms uniques.

3.1.1 Expansion des blocs

En SDLΓles blocs servent à structurer la définition d'un système. La notion de bloc est complètement statique i.eΓcette notion est perdue lors de l'exécution.

L'expansion de blocs aplatit la structure de blocs d'un système SDL. Elle prend en entrée un système SDL quelconque et fournit en sortie un autre équivalentΓconstitué de processus connectés par des routes[1].

Le système résultat comporte toutes les définitions de typesΓde signaux et de processus existantes dans les blocsΓet tous les sous-blocsΓdu système SDL de départ. Il comporte des routes entre deux processus ou entre un processus et l'environnementΓpour chaque *chemin de communication* reliant les deux processusΓou le processus et l'environnement du système SDL initial. La construction des chemins de communication est détaillée ci-après. La seule difficulté liée à l'expansion de blocs est la gestion de noms uniquesΓétant donné que différents objets ont pu être définis avec des mêmes nomsΓdans la portée de blocs différents.

La définition des chemins de communications nécessitent quelques précisions supplémentaires. D'abord par *chemin de communication* nous entendons une séquence de routes et de canaux connectés les uns avec les autres et reliant deux processus SDL ou un processus et l'environnement du système. FormellementΓon définit le graphe orienté de communication de manière suivante :

- les *nœuds* correspondent aux processusΓroutes et canaux définis dans le système initial plus un nœud supplémentaire dénotant l'environnement;

1. La syntaxe SDL empêche la définition de systèmes constitués de processus. On devrait plutôt construire un seul bloc contenant les processus et les routes. Cependant, pour la clarté de la présentation nous avons gardé le terme système.

– les *arcs* sont construits de manière suivante. Chaque nœud de type route sera connecté aux processus mentionnés dans sa définition. Les routes et les canaux seront connectées en fonction des points de connexion définis dans les blocs et les sous-structures. Finalement les canaux définis au niveau de système vers ou de l'environnement seront connectés au nœud spécial dénotant l'environnement.

Un chemin de communication est un chemin dans le graphe de communication qui relie deux nœuds processus ou un nœud processus et le nœud environnement et qui transporte un ensemble non-vide de signaux. L'ensemble de signaux transportés sur un chemin c'est l'intersection des ensembles transportés par chaque route et chaque canal composant.

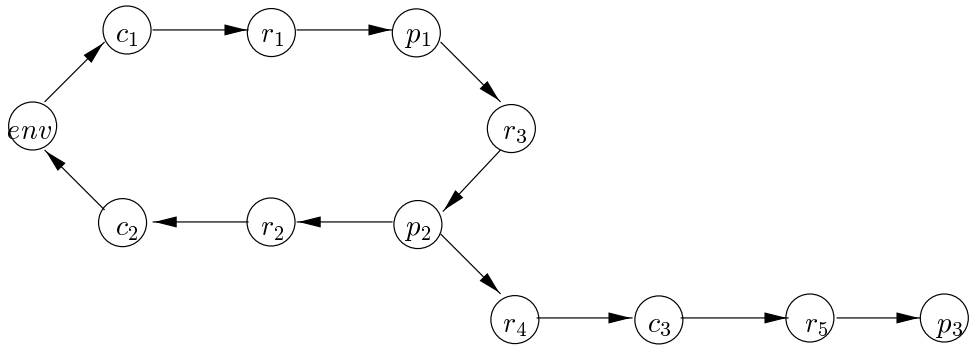

FIG. 3.1 – *Graphe de communication pour l'exemple de la figure 3.2.*

Remarque 3.1

Le système résultat par expansion est équivalent au système initial si et seulement si les canaux n'ont pas de délai de transmission. En fait cette information est ignorée lorsque les chemins de communication sont remplacés directement par des routes. ■

Exemple 3.1 *Un exemple d'expansion de blocs est présenté dans la figure 3.2.*

3.1.2 Instanciation des processus

La deuxième opération préliminaire est le remplacement de processus SDL avec instances multiples par plusieurs processus SDL identiques avec instance unique. Ça nous permet de rapprocher la notion syntaxique de processus et la notion dynamique d'instance qui sert comme base pour définir les processus IF.

L'instanciation de processus entraîne naturellement l'instanciation de routes. Les processus correspondantes aux instances doivent avoir les mêmes routes de communication que leur processus parent. Là-aussi le seul problème délicat est la gestion de noms uniques pour les processus et les routes ainsi créées.

Remarque 3.2

Cette transformation est réaliste si et seulement si le système SDL est complètement *statique* : s'il ne contient pas de la création dynamique d'instances. Dans ces conditions pour chaque

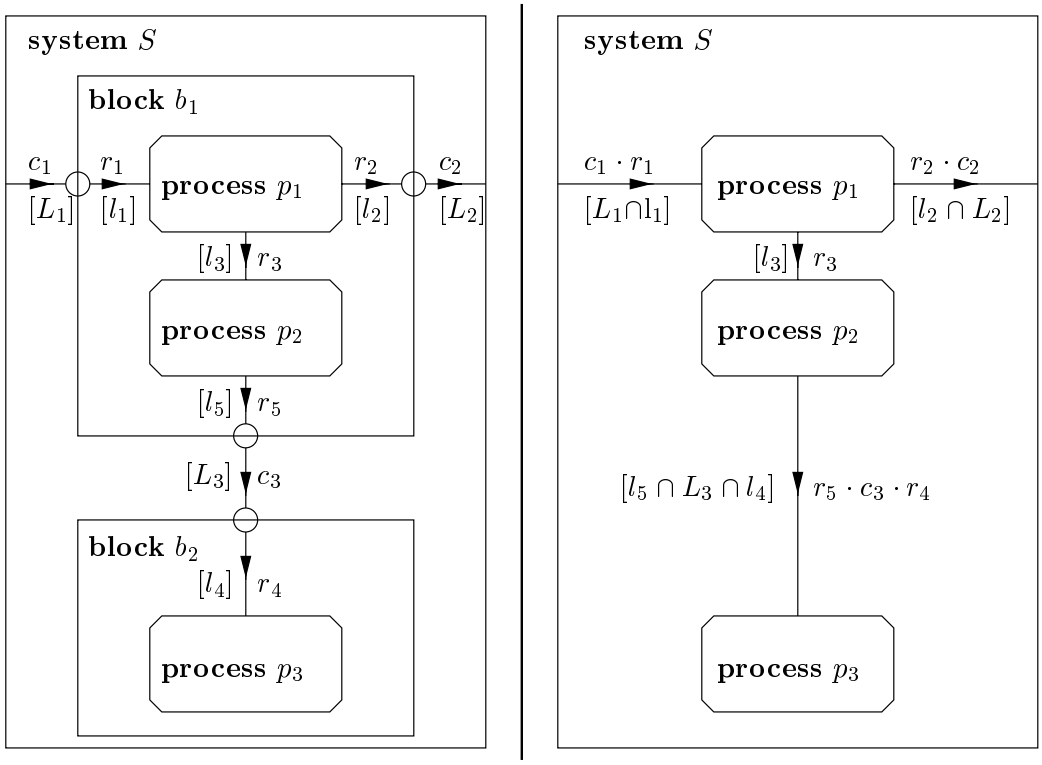

FIG. 3.2 – *Exemple d'expansion des blocs.*

processus SDL il existe un ensemble bien déterminé d'instancesΓcelles qui sont créées lors de l'initialisation du système. Elles seront les seules à exister pendant toute évolution ultérieure du système. ■

Exemple 3.2 *Un exemple d'instanciation de processus est présenté dans la figure 3.3.*

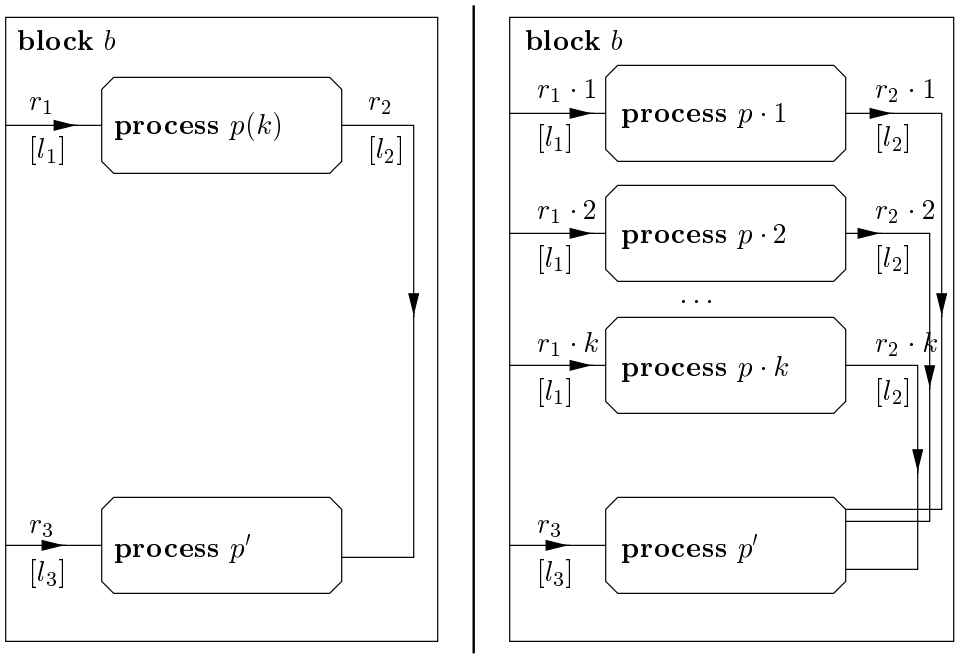

FIG. 3.3 – *Exemple d'instanciation des processus.*

3.1.3 Composition des services

La composition de services réalise la transformation de processus SDL décomposés en plusieurs services vers des processus SDL simplesΓdéfinis directement par un graphe d'états. IntuitivementΓles définitions locales aux services sont remontées au niveau des processus. Le graphe de contrôle est obtenu par la composition asynchrone des graphes de contrôle des services. Des routes supplémentaires sont éventuellement ajoutées pour réaliser la communication inter-services.

Le principe général de la composition asynchrone des automates a été rappelé dans le chapitre 2. Nous présentons ci-après les points plus particuliers liés à la composition des graphes de contrôle SDL.

Nous rappelons qu'un graphe de contrôle SDL consiste d'une transition initiale et d'un ensemble d'états SDL. Si deux tels graphes correspondent à deux services d'un processusΓ nous définissons leur composition asynchrone de manière suivante :

– la *transition initiale* consiste dans un choix non-deterministe sur l'exécution des deux

entrelacements possibles des transitions initiales. Ce traitement est nécessaire car la sémantique Z.100 prévoit bien deux étapes distinctes dans le fonctionnement d'un processus décomposé en services. Il s'agit d'une première étape où tout service exécute sa transition initialeΓet une deuxième étape où les services exécutent des autres transitions.

– les *états* du produit asynchrone sont construits pour chaque couple d'états de graphes initiauxΓen considérant le terminator **stop** comme un état particulier. IntuitivementΓ les entrées et les transitions correspondantes à un état du produit seront obtenues par l'union des entrées et des transitions des composantes. De la même manièreΓl'ensemble de signaux à sauver dans l'état du produit sera l'union des ensembles des états à sauver des composantes. A l'état **stop** on considère que les ensembles d'entrées est de signaux à sauver sont vides.

  La sémantique statique de SDL garantit que les ensembles de signaux traités (reçus ou sauvés) par chacun de services d'un processus sont deux à deux disjoints. AinsiΓil n'y a pas des conflits entre services sur les signaux en entrée. Cette propriété est gardée par la composition parallèle. Dans un état du produitΓun signal est sauvé si au moins un des services le sauveΓou consommé si au moins un des services le consomme. Ce comportement est exactement celui précisé par la norme Z.100 au sujet de l'exécution entrelacée des services.

La composition asynchrone telle que nous venons de la définir est une opération associative et commutative. AinsiΓla composition d'un ensemble de plus de deux services ne pose pas des problèmes supplémentaires i.eΓles graphes peuvent être composés deux par deuxΓdans n'importe quel ordreΓet le résultat final sera le même.

Un point sensible pour la composition des services est la communication asynchrone inter-services. En fait il est possible qu'un service envoit des signaux à un autreΓen utilisant des routes définies à l'intérieur du processus. En composant les servicesΓces routes deviennent des routes sous forme de bouclesΓde et vers le même processusΓet assurent ainsi le transfert des signaux internes au processus.

Remarque 3.3

Avec cette approcheΓil existe le risque d'explosion des états du à la composition asynchrone. CependantΓil semble que c'est la solution qui satisfait au mieux la sémantique des services. Par exempleΓon aurait pu traduire chaque service SDL par un processus IF. En conséquenceΓ il fallait considérer que ces processus IF partagent une même file d'attente de signaux en entrée. De cette manièreΓune contrainte s'impose sur l'ordre d'exécution de transitions dans ces processus. En faitΓil faut toujours que le *bon* service s'exécute et consomme le signal situé en tête de la fileΓet si aucunΓle signal est détruit. Cette contrainte est globale sur l'ensemble de services et ne peut pas se modéliser à l'aide des contraintes ou des filtres locauxΓdans les processus IF correspondants. ■

Exemple 3.3 *Un exemple de composition de services est montré dans la figure 3.4.*

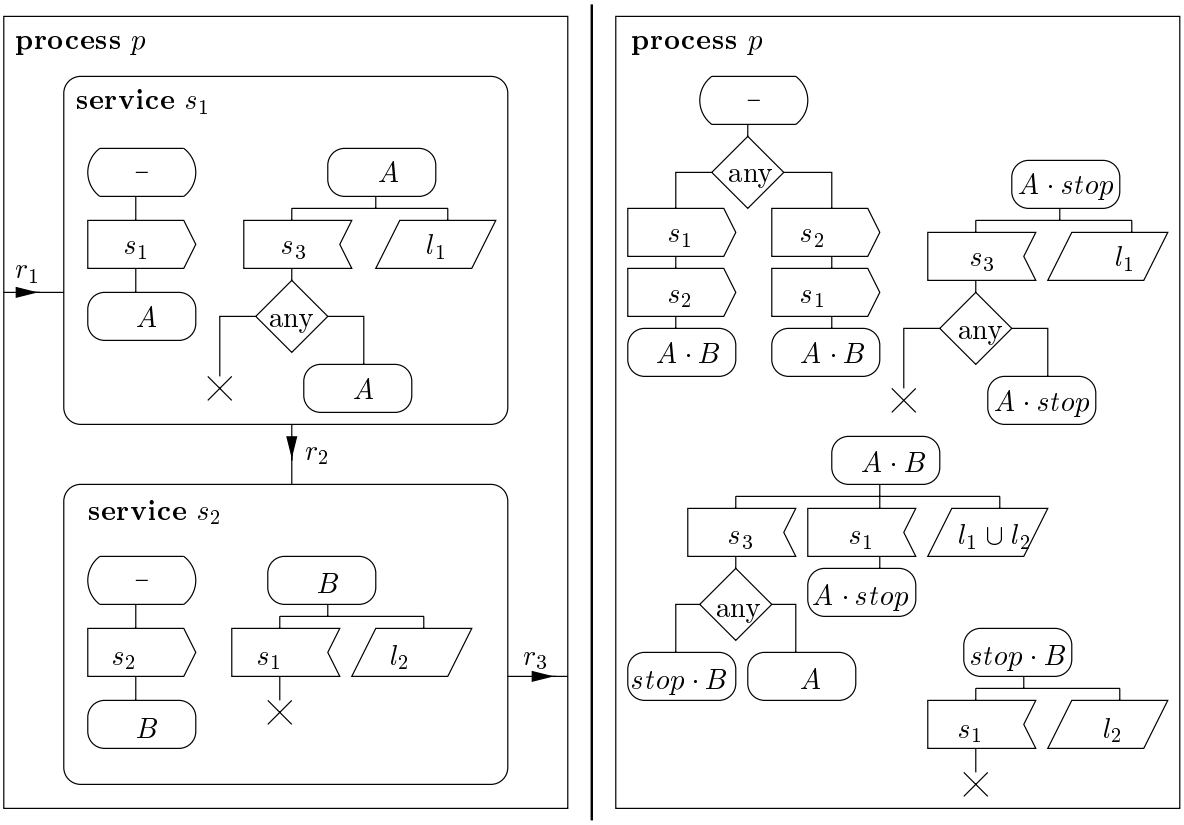

FIG. 3.4 – *Exemple de composition des services.*

3.1.4 Intégration des procédures

L'intégration de procédures prend en entrée un processus SDL et fournit en sortie un autre équivalentΓqui n'utilise plus des procédures. Il s'agit d'une technique très classique de compilation. Le principe est de remonter les définitions locales de la procédure dans chacune des entités appelantesΓpuis de remplacer les appels de la procédure par leur corps instancié avec les paramètres effectifs.

Si les procédures ne sont pas récursivesΓleur intégration ne pose aucun problème. InformellementΓon procède d'une manière ascendante sur la structure du système SDL. On commence avec les procédures qui n'appellent pas des autresΓet ensuite suiteΓjusqu'à leur élimination complète.

Les procédures récursives peuvent aussi être éliminées en sachant que généralementΓelles peuvent être remplacées par des versions non-récursives équivalentes. MaisΓcette transformation n'est pas toujours triviale et dépasse le cadre de la traduction de SDL vers IF.

Exemple 3.4 *Un exemple d'intégration des procédures est présenté dans la figure 3.5.*

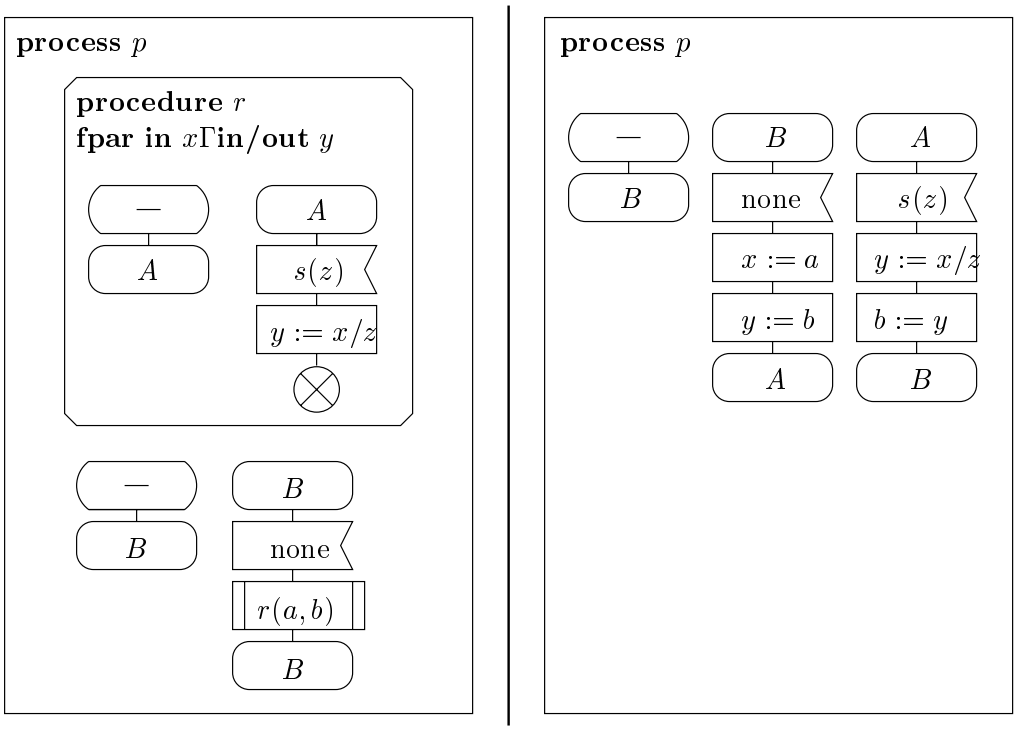

FIG. 3.5 – *Exemple d'intégration des procédures.*

3.2 Génération

La phase de génération consiste à construire un système IF équivalent à partir d'un système SDL expansé. Dans cette formeΓun système SDL est constitué d'un ensemble de processus connectés par des routes. Chaque processus a une seule instanceΓson comportement est défini explicitement par un graphe de contrôleΓet ce graphe ne contient pas des appels aux procédures.

3.2.1 Structure

Pour chaque processus SDL p_i nous construisons un processus IF homonyme p_i et une file d'attente de signaux b_i associée en entrée. Une file d'attente supplémentaire *env* modélise la sortie vers l'environnement du système.

Les processus IF construits sont complètement asynchrones. Ils communiquent à travers les files d'attente et éventuellement par certaines variables partagées. Le graphe de contrôle et les variables de chaque processus IF sont obtenus à partir de leurs correspondants dans le processus SDL. La manière exacte de génération est détaillée dans les sections suivantes.

Chaque file IF b_i transporte l'ensemble de signaux qui peuvent être reçus par le processus associé p_i i.eΓl'ensemble des signaux transportés par les routes entrantes dans p_i. La file *env* transporte l'ensemble des signaux qui peuvent être envoyés à l'extérieur du système i.eΓ l'ensemble des signaux transportés par les routes allant vers l'environnement.

system s;

  process p_1; ...
  endprocess;
  ...
  process p_n; ...
  endprocess;

  signalroute r_1 ...
  ...
  signalroute r_m ...

endsystem

system s;

buffer
  b_1 : **queue** ...
  ...
  b_n : **queue** ...
  env : **queue** ...

sync p_1 |[]| ... |[]| p_n **end**;

process p_1;
  buffer b_1;
  ...
process p_n;
  buffer b_n;
  ...

3.2.2 Communication

Signaux

Les signaux SDL sont traduits par des signaux IFΓétendus avec un paramètre supplémentaire de type *pid*. Ce paramètre transporte toujours l'identité du processus émetteur et sera utile pour traduire l'expression **sender** du côté des processus récepteurs (voir plus loin).

signal $s(t_1\Gamma\ldots\ t_n)$	$s(pid\Gamma t_1\Gamma\ldots\ t_n)$

Routes

Les routes ne se traduisent pas directement en IF. Elles sont utilisées lors de la traduction pour calculer un certaine nombre d'informations connexes :

- *les signaux reçus par processus*

  Les routes servent à calculer l'ensemble de signaux valides en réception pour chaque processus SDL. Il s'agit de signaux dont le processus pourra le recevoirΓétant donnée la topologie du réseau de communication. Pour chaque processus SDLΓcet ensemble comporte tous les signaux transportés par les routes entrantes dans le processus.

- *calcul des destinataires*

  Les routes servent aussi pour identifier les destinataires dans la traduction des sorties avec destination implicite (voir la section 3.2.3). AinsiΓun signal émis par un processus est toujours contraint à suivre seulement des routes qui peuvent le transporter. De plusΓ il doit suivre les routes mentionnées explicitement dans la clause *via* de son émission.

3.2.3 Contrôle

Chaque état d'un graphe de contrôle SDL est traduit en un état IF stable. Chaque transition SDL est traduite vers une ou plusieurs transitions IFΓétant donné que les premières comportent des *séquences* d'actions élémentaires (**output**Γ **task**Γ **set**Γ etc)Γ finies par des branchements (**decision**) ou des transferts de contrôle (**join**) Le découpage est nécessaire pour modéliser l'exécution séquentielle de SDLΓcontrairement à IF où l'exécution des actions dans une transition est parallèle. Le principe général du découpage est présenté dans la figure 3.6.

Etats

Les états d'un processus SDL sont traduits par des états *stables* dans les processus IF correspondants. Chaque état IF ainsi créé a les caractéristiques suivantes :

- l'ensemble de signaux à *sauver* est construit à partir de l'ensemble équivalent en SDL plus des conditions implicite de sauvegarde introduites par les entrées avec condi-

tion de franchissement (**provided**) et les signaux prioritaires (**priority input**). AinsiΓ d'une part tout signal non-consommable à cause d'une condition de franchissement non-satisfaite est explicitement sauvé à l'état IF. De plusΓpour assurer la consommation prioritaireΓla présence d'un signal prioritaire dans la file d'attente implique la sauvegarde de tout autre signal non-prioritaire.

- l'ensemble de signaux à *détruire* est aussi construit explicitement. Il comporte tous les signaux recevables et non-traités (reçus ou sauvés) par le processus SDL à l'état correspondant.
- *la condition de progression du temps* est vraie.

Le principe de la traduction d'un état q dans le processus p est donnée par la règle suivante. On remarque notamment la sauvegarde explicite des signaux reçus s_i si la condition de franchissement e n'est pas satisfaite ou un signal prioritaire s_p se trouve dans la file d'attente b ; par s_d nous avons noté un signal arbitraire non reçu ou sauvegardé à q.

state q;
  ...
  input s_i(...) **provided** e; ...
  ...
  priority input s_p(...); ...
  ...
  save s_sΓ...
  ...
endstate;

state q;
  save
    ...
    s_i **in** b **if** not e;
    s_i **in** b **if** contains(bΓs_p)
    ...
    s_s **in** b **if** true;
    ...
  discard
    ...
    s_d **in** b **if** true;
    ...
  tpc true;
end

Deux états supplémentaires sont ajoutés de manière systematique dans chaque processus IF pour modéliser respectivement l'état de départ *start* et l'état d'arrêt *stop* du processus. Ces états sont stablesΓils ont les ensembles de signaux à sauver et à détruire videsΓet la condition de progression du temps vraie.

FinalementΓles états supplémentaires ajoutés pour couper les transitions SDL sont *instables*Γ les ensembles de signaux à sauvegarder ou détruire sont videsΓet la condition de progression du temps est fausse.

Remarque 3.4

Le choix de l'instabilité des états intermédiaires assure *l'atomicité* de l'exécution de séquences de transitions IFΓcorrespondantes à une transition SDL. Ce choix n'est pas conforme à la norme Z.100 où l'atomicité est assurée plutôt pour l'exécution d'une actions élémentaire individuelle. MaisΓbien que ce comportement peut être obtenu en IFΓen considérant stables

tous les états intermédiairesΓil est beaucoup trop fin pour les techniques de validation que nous voulons appliquer. ■

Transitions

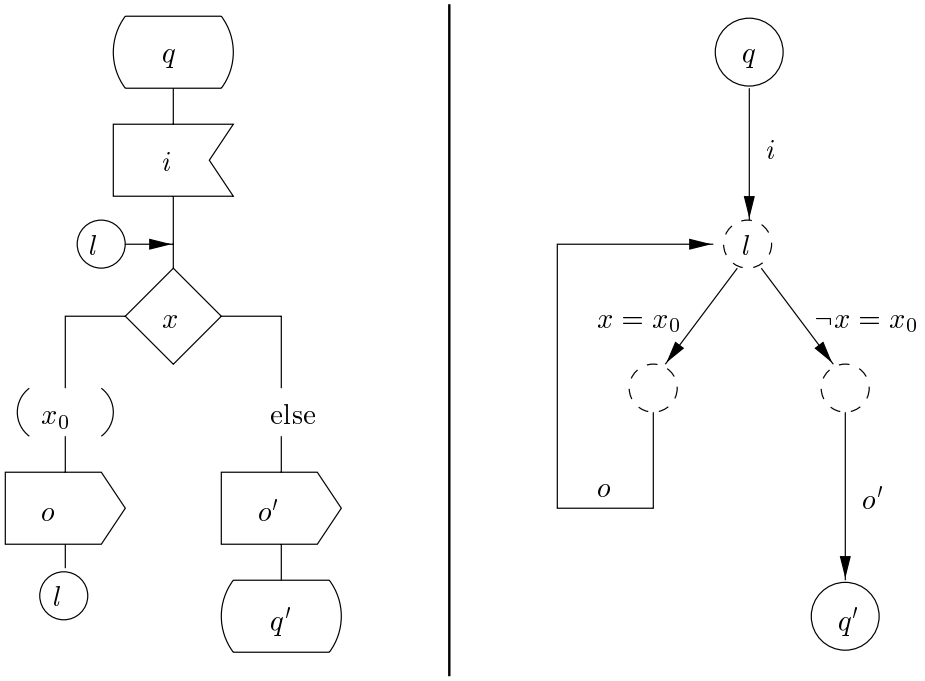

FIG. 3.6 – *Exemple de génération de transitions IF.*

Une transition IF correspond à une action élémentaire SDL (voir figure 3.6). Toutes les transitions IF sont construites implicitement à l'échéance *eager*. Ça donne une priorité minimale à la progression du tempsΓet assure une réactivité maximale de la part du système. Ce choix est à la fois agréé par les utilisateurs de SDL et pragmatique du point de vue validation.

Par suiteΓnous allons décrire la forme des transitions IFΓen fonction du type de l'action SDL correspondante.

entrées

Les entrées des signaux SDLΓnormales ou prioritairesΓsont traduites par des entrées des signaux IF. Dans tous les casΓnous considérons le traitement explicite de la variable *sender*Γ systématiquement ajoutée comme paramètre de chaque entrée. Son rôle est l'évaluation de l'expression SDL **sender**. La condition de franchissement est ajoutée comme garde dans la transition IF correspondante. En considérant que l'entrée a lieu dans le processus p avec la file d'attente associée bΓla traduction est faite par les règles suivantes :

input $s(x_1\Gamma\ldots\ x_n)$ **provided** e	**if** e **input** $s(sender\Gamma x_1\Gamma\ldots\ x_n)$ **from** b
priority input $s(x_1\Gamma\ldots\ x_n)$	**input** $s(sender\Gamma x_1\Gamma\ldots\ x_n)$ **from** b

Les entrées des signaux issus d'expirations des temporisations sont traduites par des gardes portant sur les valeurs des horloges correspondants. Le désarmement de l'horloge concernée est nécessaire tout suite après la garde afin d'éviter plusieurs *consommations* d'une même expiration. Une présentation plus détaillée des choix faits à propos de la traduction de temporisations se trouve dans la section 3.2.4.

input $c(x_1\Gamma\ldots\ x_n)$ **provided** e	**if** $e\ \wedge\ c=0\quad sender := p\Gamma$**reset** c

FinalementΓles entrées spontanées et les entrées continues ne donnent pas des entrées IF : elles se traduisent uniquement par des gardes en IF et respectivement la mise à jour du *sender*. Les priorités sur les entrées continues sont traduites par des gardes supplémentaires. Si $h \in \mathbb{Z}$ est une priorité nous notons par e_h la disjonction des gardes sur les entrées continues avec une priorité strictement plus grande que h. NaturellementΓe_h sera vraie s'il existe au moins une entrée de priorité plus grande que h qui est franchissable. La traduction est faite de manière suivante :

input none provided e	**if** $e\quad sender := p$
provided e **priority** h	**if** $e\ \wedge\ \neg e_h\quad sender := p$

sorties

L'envoi de signaux en SDL est considéré dynamique : un signal est délivré au réseau et il trouve son destinataire en fonction d'informations de routage associées (clauses *to* et *via*) et de profils des routes et des canaux. En IFΓl'envoi de signaux est statique : l'émetteur doit spécifier soit la file d'attente soit le pid du processus destinataire (au cas où il serait déposé dans la file associée).

Si à partir d'informations de routage et de profiles des routes on peut statiquement déduire un destinataire uniqueΓla sortie SDL sera traduite dans une sortie IF équivalente. SinonΓ la traduction nécessite l'introduction de plusieurs transitions pour expliciter statiquement le choix arbitraire du destinataireΓparmi tous ceux qui sont possibles.

output $s(e_1\Gamma\ldots\ e_n)$ [**to** e] **via** ...	**output** $s(p\Gamma e_1\Gamma\ldots\ e_n\ )$ **to** b'

affectations

Les affectations SDL sont traduites directement par des affectations IF.

task $x := e$	$x := e$

armements / désarmements

L'armement et le désarmement des temporisations sont traduites par des affectations des horloges correspondants. Nous mentionnons que le passage de paramètres à travers de temporisations n'est pas traduit i.eΓles expressions correspondantes sont ignorées.

set $eΓc(e_1Γ\ldots \quad e_n)$	**set** $c := e$
reset $c(e_1Γ\ldots \quad e_n)$	**reset** c

décisions

Les décisions formelles sont traduites par des gardes des transitions IF. Les décisions informelles servent juste pour introduire du non-déterminisme. Elles se traduisent par le branchement arbitraire du contrôle dans les processus IF.

terminaisons

Les terminaisons sont traduites implicitement dans le flot de contrôle du processus IF.

3.2.4 Temporisations

Les temporisations SDL sont traduites par des horloges décroissants de IF.

timer $c(t_1Γ\ldots \quad t_n)$	$c : timer$

La gestion des temporisations est relativement simplifiée. D'abordΓle transfert des paramètres à travers de temporisations n'est plus supporté. EnsuiteΓles entrées des signaux d'expiration sont remplacées par des gardes testant l'égalité des horloges à zéro (la valeur d'expiration). Les actions d'armement et désarmement sont traduites par des affectations d'horloges et le test d'activité revient lui aussi à une comparaison à zéro.

Même en absence de paramètres à porterΓcette traduction *n'est pas conforme* à la sémantique Z.100 parce que :

- la traduction remplace syntaxiquement et donc confond la consommation du signal issue de l'expiration avec le test d'expiration de la temporisationΓqui sont des événements dissociés dans la sémantique Z.100 ;
- la traduction ne fait plus le passage d'un signal d'expiration à travers la file d'attente et du coûtΓcertaines opérations spécifiques aux gestion des signaux comme la sauvegarde ou la consommation prioritaire ne sont plus traduites au cas de signaux issus de temporisations.

CependantΓnous pouvons obtenir exactement le même comportement en IF en faisant une traduction différente. On peut construire un processus supplémentaire responsable de la gestion d'une temporisation qui simule exactement le comportement spécifiée par la Z.100.

3.2.5 Données

Variables

Les variables SDL sont traduites par des variables IF. Elles sont globales au système ou locales aux processus IF en fonction de la portée SDL :

- *globale*Γsi la variable est à portée globale (**revealed**) ; dans ce cas elle est susceptible d'être partagée par plusieurs processus IF associés aux différents processus SDL ;
- *locale*Γsi la variable est à portée locale ; dans ce casΓelle sera locale au processus IF correspondant.

Les paramètres formels des processus sont vus comme des variables locales au processus SDL et traduits en conséquence.

FinalementΓpour chaque processus SDL nous introduisons trois variables *sender*Γ*parent* et $offspring$ de type *pid* afin de pouvoir traduire les expressions SDL prédéfinies **sender**Γ **parent** et **offspring**.

Expressions

Les expressions SDL se traduisent en appliquant les principes suivantsΓau cas par cas:

- les expressions **view** font des références aux variables **revealed** des autres processus. NormalementΓelles sont directement remplacées par les noms des variables en cause. CependantΓdans le cas où plusieurs processus exportent des variables avec le même nomΓla traduction n'est pas possible carΓstatiquementΓnous n'avons pas le moyen de distinguer la bonne référence ;
- l'expression **active** sur un timer se traduit vers une condition portant sur la valeur de l'horloge correspondante i.eΓon traduit **active**(x) par $x \geq 0$;
- l'expression **now** est traduite par la valeur zéro ; ça nous permet de traduire uniquement des systèmes qui ne font jamais référence au temps absoluΓmais uniquement au temps relative à la valeur courante de **now** ;
- l'expression **self** est remplacée par le nom du processus correspondantΓqui sert comme identificateur du processus dans le système IF ;
- les expressions **sender**Γ**parent** et **offspring** sont remplacées chacune par la variable avec le nom correspondant.

3.3 Discussion

Nous avons présenté la traduction d'un sous-ensemble du langage SDL vers le modèle intermédiaire IF. L'exigence principale a été de préserverΓpour le sous-ensemble considéréΓla sémantique de référence de SDL fournie par la Recommandation Z.100. Bien que aucune preuve formelle n'est disponible sur ce pointΓtous nos choix ont été faits pour obtenir le même ensemble de comportements dynamiques dans les deux cas.

Un point important de la traduction est la définitionΓde manière exacte et expliciteΓde la sémantique du temps pour SDL. Sur ce pointΓla norme Z.100 ainsi que d'autres sémantiques ultérieures suscitent beaucoup de questions. Nous avons opté pour une traduction assurant la *réactivité maximale* de la part du système i.eΓen considérant toutes les transitions urgentes. D'autres solutions sont aussi bien possiblesΓnous avons pris celle-ci car d'une part elle corresponde à l'intuition des utilisateurs (voir d'ailleurs le choix par défaut offerte par les outils industriels sur ce point) 0et d'autre part elle simplifie la validationΓen réduisant le non-déterminisme à l'exécution.

FinalementΓnous avons aussi regardé la traduction d'autres langages de spécification. Nous avons plus particulièrement considéré LOTOS [ISO88]Γnorme OSI pour la description des protocolesΓet PROMELAΓle langage natif du model-checker SPIN [Hol91]. Dans les deux casΓnous nous sommes intéressés principalement à la traduction des aspects temporels.

Une approche pour modéliser et valider les spécifications LOTOS est d'utiliser des réseaux de PetriΓplutôt que des automates communicantsΓcomme représentation intermédiaire [GS90a]. CependantΓl'expérience montre queΓen généralΓles spécifications des protocoles décrivent bien des compositions parallèles de plusieurs processus séquentiels. Cette observation a motivé aussi l'emploi des techniques de génération et vérification compositionelles dans le contexte du langage LOTOS [KM97]. De plusΓles extensions temporelles introduites dans E-LOTOS [Que98]ΓET-LOTOS [LL97] et LOTOS-NT [Sig99] sont similaires à celles existantes en IF. Les actions ont des échéances définies implicitement par leurs types : les exceptions et les actions internes sont urgentesΓtant que les autres ne le sont pas. La traduction de LOTOS en IF est donc envisageable : d'une partΓla composition parallèle avec synchronisation est possible en IFΓet d'autre partΓun processus LOTOS peut être modélisé par un automate temporisé étendu IF.

PROMELA est un langage de description intermédiaire à base d'automates communicants asynchrones. Il a beaucoup de succès grâce à la disponibilité et l'efficacité du model-checker SPIN [Hol91]. Plusieurs extensions temporelles de PROMELA ont été proposéesΓpar exempleΓ celle de [CT96]Γà base d'automates temporisésΓou celle de [BD98]Γqui utilise des temporisations à la SDL. Les modèles sous-jacents de PROMELA et de IF sont relativement proches et rendent possible la traduction entre ces deux formalismes. ConcrètementΓune traduction de IF vers PROMELA a été déjà définie et mise en œuvre par une équipe de l'Université d'EindhovenΓdans le cadre du projet européen VIRES [BDHS00].

Deuxième partie

Outils

Chapitre 4

Analyse statique

Dans la validation à base de modèlesΓles variables et les files du programme constituent l'une des principales sources de l'explosion des états. En particulier dans le contexte de la communication asynchroneΓla gestion de files d'attente introduit une complexité très importante du fait que les files mémorisentΓà part les messages transmisesΓde l'information sur l'ordre d'exécution. Afin de contourner ce problème nous nous sommes intéressés à l'utilisation *en amont* d'analyses statiques : des techniques permettant le calcul d'informations pertinentes sur le comportement dynamique d'un programmeΓsans l'exécuter.

Nous nous intéressons plus particulièrement à des techniques d'analyse de flot de donnéesΓ utilisées dans des domaines comme l'optimisation de code [ASU86ΓMuc97]. Les analyses statiques que nous présentons concernent notamment le calcul d'informations sur les variables et parfois les files du programme. Ces informations incluent l'activitéΓles domaines des valeursΓles invariantsΓetc. GénéralementΓces informations sont utiles à la fois pour simplifier ou abstraire syntaxiquement le programmeΓou encore pour améliorer dynamiquement les performances du processus de validation.

Nous nous intéressons aux programmes IF représentés sous la forme d'automates temporisés communicants étendus ATCDΓtels qu'ont été définis dans la section 2.2. MaisΓpour la clarté de la présentation nous allons considérer uniquement un sous-ensemble des ATCDΓ plus exactement :

- nous ignorons les aspects temporels i.eΓles gardes temporellesΓles affectations d'horloges ou encore les échéances associées aux transitions. En faitΓdes analyses statiques d'horlogesΓcomme par exemple l'activité ou l'égalitéΓont été développées dans [Daw98] et peuvent être reprises sans restrictions dans le cadre des ATCD.
- nous considérons des transitions avec une forme simplifiée i.eΓcomportant uniquement une garde et une actionΓcette dernière étant soit une entréeΓ soit une sortieΓ soit une synchronisationΓsoit une affectation multiple. Néanmoins les techniques d'analyse statique peuvent être étendues sans restriction au cadre général;
- la composition parallèle d'automates n'est pas prise en compteΓnous nous limitons à

des techniques définies au niveau d'un seul automate. Mais en généralΓles techniques présentées sont dans la plupart compositionnelles i.eΓdes résultats au niveau d'une composition parallèle d'automates peuvent être obtenus en combinant les résultats locauxΓobtenus pour chaque automate.

Soit $P = (D, X, S, B, G, C, Q, T)$ un ATCD de cette forme simplifié. Par la suiteΓles transitions de P sont notées simplement par $q \xrightarrow{h\ a} q'$ avec $q, q' \in Q\Gamma\, h \in BExp[X]$ et $a \in In[X, S, B] \cup Out[X, S, B] \cup Sync[X, G] \cup Rst[X]$.

4.1 Analyse d'activité

4.1.1 Variables utilisées et variables définies

D'abord nous introduisons les notions de *variable utilisée* et de *variable définie* dans une transition. Ces sont des points de départ pour des définitions plus compliquées concernant l'utilité des variables au niveau d'un programme.

IntuitivementΓles variables sont utilisées dans les gardesΓles émissions ou en partie droite des affectations. Elles sont définies dans les réceptions ou en partie gauche des affectations. Plus formellementΓles ensembles de variables respectivement utilisées (Use) et définies (Def) par une transition $q \xrightarrow{\alpha} q'$ sont donnés par la table suivante :

α	$Use(\alpha)$	$Def(\alpha)$
$h\ \ b?s(x_1, \ldots\ x_n)$	$vars(h)$	$\{x_1, \ldots\ x_n\}$
$h\ \ b!s(e_1, \ldots\ e_n)$	$vars(h) \cup \bigcup_{i=1}^{n} vars(e_i)$	$\emptyset$
$h\ \ g\ o_1 \ldots o_n$	$vars(h) \cup \bigcup_{o_i = !e_i} vars(e_i)$	$\bigcup_{o_i = ?x_i} \{x_i\}$
$h\ \ x_1 := e_1, \ldots\ x_n := e_n$	$vars(h) \cup \bigcup_{i=1}^{n} vars(e_i)$	$\{x_1, \ldots\ x_n\}$

A partir de l'utilité locale on peut définir les ensembles des variables utilisées et définies globalement dans P :

$$Use(P) = \bigcup_{q \xrightarrow{\alpha} q' \in T} Use(\alpha) \qquad Def(P) = \bigcup_{q \xrightarrow{\alpha} q' \in T} Def(\alpha)$$

Nous considérons le sous-ensemble des *variables propres* de P comme étant l'intersection $Use(P) \cap Def(P)$. Ces sont les variables réellement utilisées de la spécification : à la fois utilisées et définies par les transitions. GénéralementΓles autres variables peuvent être éliminées de P sans perte d'information carΓau moins une des situations suivantes doit se présenter :

- la variable n'est définie dans aucune transition : c'est une variable constanteΓqui garde sa valeur initiale pendant toute l'exécution. Ces variables peuvent être donc remplacées par leurs valeurs initiales et ensuite éliminées de la spécification.
- la variable n'est utilisée dans aucune transition : elle peut être affectée par ailleurs mais sa valeur n'est jamais prise en compte. Ces variables et leurs affectations peuvent aussi être éliminées de la spécification.

4.1.2 Variables actives

Une notion plus fine est la notion d'activité : une variable est *active* (*live*Γen anglais) dans un état si et seulement si elle peut être utilisée avant d'être redéfinie dans une séquence d'exécution qui commence dans cet état.

FormellementΓles ensembles de variables actives sont définis comme la plus petite solution du système d'équations suivantΓpour chaque état :

$$\forall q \in Q \quad Live(q) = \bigcup_{q \xrightarrow{\alpha} q'} Use(\alpha) \cup \big(Live(q') \setminus Def(\alpha)\big) \tag{4.1}$$

PratiquementΓce système peut être résolu de manière itérative. On commence par des ensembles vides et on itère les équations jusqu'à trouver le point fixe :

```
pour chaque état q faire
   Live(q) := ∅;
répéter
   pour chaque état q faire
      Live(q) := ⋃_{q -α-> q'} Use(α) ∪ (Live(q') \ Def(α))
jusqu'à convergence sur les Live
```

La convergence de cet algorithme repose sur la monotonie des équations 4.1 par rapport à l'inclusion d'ensemblesΓet sur le fait que l'ensemble des variables est fini. En généralΓon dit par abus de langage qu'un système d'équations est monotone dans un treillisΓsi l'opérateur qu'il décrit est monotone dans ce treillis.

Exemple 4.1 *Dans l'automate présenté de la figure 4.1, les ensembles de variables actives sont respectivement* $Live(q_1') = \{x_1\}$, $Live(q_2') = \{x_2\}$ *et* $Live(q_1) = Live(q_2) = \emptyset$.

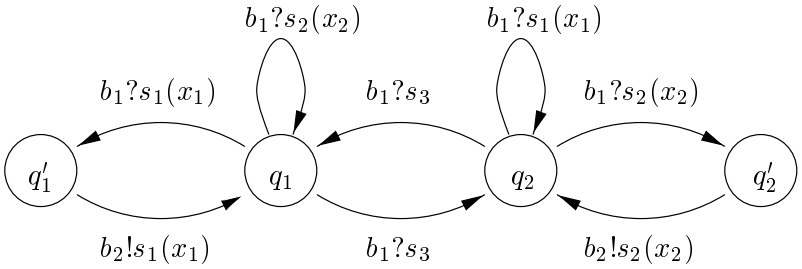

FIG. 4.1 – *Exemple de variables actives.*

L'activité de variables peut être prise en compte à deux niveaux différents. PremièrementΓau niveau syntaxiqueΓl'activité permet la réduction du nombre de variables par recouvrement. DeuxièmementΓau niveau sémantique l'activité permet la définition d'une bisimulation forte sur les états du système de transitions étiquetées associé au programme IF [BFG99].

4.1.3 Recouvrement d'activité

Soit $P = (D, X, S, B, G, C, Q, T)$ un ATCD et Y un ensemble de variables. Une application $\eta : Q \to X \to Y$ est *un recouvrement d'activité* des variables de X par des variables de Y dans P si et seulement si les conditions suivantes sont satisfaitesΓpour tout état $q \in Q$:

1. préservation de types : $\forall x \in Live(q) \quad \Rightarrow \quad type(x) = type(\eta(q)(x))$
2. préservation de l'activité : $\forall x, x' \in Live(q),\ x \neq x' \quad \Rightarrow \quad \eta(q)(x) \neq \eta(q)(x')$

IntuitivementΓsi P est un ATCD et $\eta : Q \to X \to Y$ est un recouvrement d'activité alors P peut être complètement réécrit avec les variables de YΓavec la préservation complète de la sémantique. La transformation consiste dans le renommage local des variables activesΓ transition par transitionΓen fonction des renommages $\eta(q) : X \to Y$ associés aux états $q \in Q$.

Soit $q \xrightarrow{\alpha} q'$ une transition de P fixée. La transformation est une double substitution de αΓ qui utilise $\eta(q)$ pour les variables utilisées et $\eta(q')$ pour les variables définies par α. De plusΓ un nombre d'affectations supplémentaires s'avèrent parfois nécessaires pour mettre à jour les variables actives de q et de q'Γinchangées par αΓmais renommées de manière différente dans

q et q'. FormellementΓnotons par $e^{\eta(q)}$ et $x^{\eta(q)}$ l'application des renommages aux expressions et aux variables :

$$e^{\eta(q)} = \eta(q)(e) \qquad \forall e \in Exp[X]$$

$$x^{\eta(q)} = \begin{cases} \eta(q)(x) & \text{si } x \in Live(q) \\ any & \text{sinon} \end{cases} \qquad \forall x \in X$$

La substitution de α par rapport au ηΓnotée par α^ηΓest définie par les règles suivantes :

$$\frac{h \;\; b?s(x_1, \ldots \; x_n)}{h^{\eta(q)} \;\; b?s(x_1^{\eta(q')}, \ldots \; x_n^{\eta(q')})} \qquad \frac{h \;\; g \; o_1 \ldots o_n \quad o_i' = \begin{cases} ?x_i^{\eta(q')} & \text{si } o_i = ?x_i \\ !e_i^{\eta(q)} & \text{si } o_i = !e_i \end{cases}}{h^{\eta(q)} \;\; g \; o_1' \ldots \; o_n'}$$

$$\frac{h \;\; b!s(e_1, \ldots \; e_n)}{h^{\eta(q)} \;\; b!s(e_1^{\eta(q)}, \ldots \; e_n^{\eta(q)})} \qquad \frac{h \;\; x_1 := e_1, \ldots \; x_n := e_n}{h^{\eta(q)} \;\; x_1^{\eta(q')} := e_1^{\eta(q)}, \ldots \; x_n^{\eta(q')} := e_n^{\eta(q)}}$$

L'affectation auxiliaire de α par rapport au ηΓnotée par α_0^ηΓest définie de manière suivante :

$$\alpha_0^\eta = \{x^{\eta(q)} := x^{\eta(q')} \mid x \in Live(q) \cap Live(q') \setminus Def(\alpha), \; x^{\eta(q)} \neq x^{\eta(q')}\}$$

FinalementΓle recouvrement d'un automate $P = (D, X, S, B, G, C, Q, T)$ par rapport au recouvrement d'activité $\eta : Q \to X \to Y$ est l'automate $P^\eta = (D, Y, S, B, G, C, Q, T^\eta)$ où $T^\eta = \{q \xrightarrow{\alpha^\eta \cup \alpha_0^\eta} q' \mid q \xrightarrow{\alpha} q'\}$. La construction garantit que P et P^η sont équivalents i.eΓ les systèmes de transitions étiquetées associés sont fortement bisimilaires.

Exemple 4.2 *Les deux variables de l'automate de la figure 4.1 peuvent être recouvertes à l'aide d'une seule variable y. L'automate ainsi obtenu, est présenté dans la figure 4.2.*

4.1.4 Bisimulation d'activité

Nous définissons la relation d'*équivalence sur les contextes de variables*Γinduite par l'activité de manière suivante : deux contextes sont équivalents dans un état si et seulement si ils affectent les mêmes valeurs aux variables actives de l'état. FormellementΓpour chaque état q nous définissons l'équivalence $\sim_q^{live}$ par :

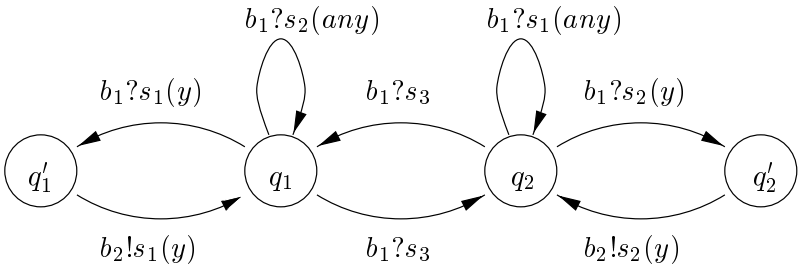

FIG. 4.2 – *Recouvrement de l'automate 4.1.*

$$\rho_1 \sim_q^{live} \rho_2 \quad - \quad \forall x \in Live(q)\ \rho_1(x) = \rho_2(x) \tag{4.2}$$

Nous considérons des équivalences similaires définies sur les contextes des files. Intuitivement deux contextes sont équivalents si et seulement si ils permettent de franchir les mêmes séquences de transitions et avec le même effet. Plus précisément deux contextes sont équivalents si récursivement pour toute entrée atteignable les files concernées offrent les mêmes possibilités d'évolution : soit les deux sont vides soit les deux contiennent en tête le bon signal avec les mêmes paramètres pour les variables actives soit les deux contiennent en tête des signaux inattendus.

Formellement nous définissons l'*équivalence sur les contextes de files* $\approx_q^{live}$ comme la plus grande solution du système d'équations suivant pour chaque état q :

$$\delta_1 \approx_q^{live} \delta_2 \quad -_\nu \bigwedge_{q \xrightarrow{\alpha} q'} \begin{cases} \alpha \text{ est une entrée } h\ \ b?s(x_1, \ldots\ x_n)\ \Rightarrow \\ \quad \begin{cases} \delta_1(b) = \epsilon\ \wedge\ \delta_2(b) = \epsilon\ \wedge \\ \quad\quad \delta_1 \approx_{q'}^{live} \delta_2 \\ \quad \vee \\ \quad \delta_1(b) = s(v_1^1, \ldots\ v_n^1).w_1\ \wedge\ \delta_2(b) = s(v_1^2, \ldots\ v_n^2).w_2\ \wedge \\ \quad\quad \forall i\ x_i \in Live(q') \Rightarrow v_i^1 = v_i^2 \\ \quad\quad \wedge \\ \quad\quad \delta_1[w_1/b] \approx_{q'}^{live} \delta_2[w_2/b] \\ \quad \vee \\ \quad \delta_1(b) = s_1(v_1^1, \ldots\ v_n^1).w_1\ \wedge\ \delta_2(b) = s_2(v_1^2, \ldots\ v_m^2).w_2\ \wedge \\ \quad\quad s_1 \neq s\ \wedge\ s_2 \neq s \end{cases} \\ \alpha \text{ n'est pas une entrée } \Rightarrow \\ \quad \delta_1 \approx_{q'}^{live} \delta_2 \end{cases} \tag{4.3}$$

Les équations sont monotonesΓétant donnée l'absence de négations et la monotonie des opérateurs de disjonction et conjonction. Il en résulte ainsi que la définition est consistanteΓ conformément à la théorème de TarskiΓle plus grand point fixe existe et il est unique.

Exemple 4.3 *Soit à nouveau l'automate présenté dans la figure 4.2. Les deux contextes suivants :*

$$\begin{aligned} \delta_1 &= \langle\ s_2(0).s_3.s_1(0).s_2(4)/b_1,\ s_1(0)/b_2\ \rangle \\ \delta_2 &= \langle\ s_2(7).s_3.s_1(5).s_2(4)/b_1,\ \epsilon/b_2\ \rangle \end{aligned}$$

sont équivalents aux états q_1 et q_1'. Par contre, ils ne le sont pas aux états q_2 et q_2'. De plus, on remarque que l'équivalence repose uniquement sur les contenus de la file b_1. Les contenus de b_2 sont toujours équivalents en absence d'entrées de b_2 dans l'automate.

Nous définissons maintenant l'*équivalence d'activité* sur les états du système de transitions : deux états sont équivalents si et seulement si ils ont le même état de contrôle q et si respectivement les contextes sur les variables et sur les files sont équivalents à q :

$$(q,\rho_1,\delta_1) \approx^{live} (q,\rho_2,\delta_2) \;-\; \rho_1 \sim_q^{live} \rho_2 \;\wedge\; \delta_1 \approx_q^{live} \delta_2 \tag{4.4}$$

Théorème 4.1 *L'équivalence $\approx^{live}$ est une bisimulation.*

Preuve: La preuve consiste à montrer que l'équivalence $\approx^{live}$ satisfait la définition d'une bisimulation forte. Plus exactementΓnous allons prouver que si $(q,\rho_1,\delta_1) \approx^{live} (q,\rho_2,\delta_2)$ alors :

$$\forall\ (q',\rho_1',\delta_1')\ \ (q,\rho_1,\delta_1) \xrightarrow{a} (q',\rho_1',\delta_1')\ \Rightarrow$$
$$\exists\ (q',\rho_2',\delta_2')\ \ (q,\rho_2,\delta_2) \xrightarrow{a} (q',\rho_2',\delta_2')\ \wedge\ (q',\rho_1',\delta_1') \approx^{live} (q',\rho_2',\delta_2')$$

$$\forall\ (q',\rho_2',\delta_2')\ \ (q,\rho_2,\delta_2) \xrightarrow{a} (q',\rho_2',\delta_2')\ \Rightarrow$$
$$\exists\ (q',\rho_1',\delta_1')\ \ (q,\rho_1,\delta_1) \xrightarrow{a} (q',\rho_1',\delta_1')\ \wedge\ (q',\rho_1',\delta_1') \approx^{live} (q',\rho_2',\delta_2')$$

Soit donc $(q,\rho_1,\delta_1) \approx^{live} (q,\rho_2,\delta_2)$ et $q \xrightarrow{\alpha} q'$ une transition. Nous distinguons plusieurs cas différentsΓen fonction de α :

1. $\alpha = h\ \ b?s(x_1,\ldots,x_n)$

   $\forall k = 1,2\ \ \delta_k(c) = s(v_1^k,\ldots\ v_n^k).w_k,\ \rho_k(h) = true\ \Rightarrow$

   $$(q,\rho_k,\delta_k) \xrightarrow{b?s(v_1^k,\ldots\ v_n^k)} (q',\rho_k',\delta_k'),\ \rho_k' = \rho_k[v_1^k/x_1,\ldots\ v_n^k/x_n],\ \delta_k' = \delta_k[w_k/b]$$

   $\delta_1 \approx_q^{live} \delta_2\ \Rightarrow\ \forall x_i \in Live(q')\ v_i^1 = v_i^2,\ \ \delta_1[w_1/b] \approx_{q'}^{live} \delta_2[w_2/b]$

   $\rho_1 \sim_q^{live} \rho_2\ \Rightarrow\ \rho_1(h) = \rho_2(h),\ \ \rho_1[v_1^1/x_1,\ldots\ v_n^1/x_n] \approx_{q'}^{live} \rho_2[v_1^2/x_1,\ldots\ v_n^2/x_n]$

2. $\alpha = h\ \ b!s(e_1,\ldots e_n)$

   $\forall k = 1,2\ \ \delta_k(c) = w_k,\ \forall i = 1,n\ \rho_k(e_i) = v_i^k,\ \rho_k(h) = true\ \Rightarrow$

   $$(q,\rho_k,\delta_k) \xrightarrow{b!s(v_1^k,\ldots\ v_n^k)} (q',\rho_k,\delta_k'),\ \delta_k' = \delta_k[w_k.s(v_1^k,\ldots\ v_n^k)/b]$$

   $\rho_1 \sim_p^{live} \Rightarrow\ \rho_1(h) = \rho_2(h),\ \forall i = 1,n\ \rho_1(e_i) = \rho_2(e_i),\ \rho_1 \approx_{q'}^{live} \rho_2$

   $\delta_1 \approx_q^{live} \delta_2\ \Rightarrow\ \delta_1 \approx_{q'}^{live} \delta_2\ \Rightarrow\ \delta_1[w_1.s(v_1^1,\ldots\ v_n^1)/b] \approx_{q'}^{live} \delta_2[w_2.s(v_1^2,\ldots\ v_n^2)/b]$

3. $\alpha = h\ \ g\ o_1\ldots\ o_n$

   $\forall k = 1,2\ \ \forall o_i = !e_i\ \rho_k(e_i) = v_i^k,\ \rho_k(h) = true\ \Rightarrow$

   $$(q,\rho_k,\delta_k) \xrightarrow{g\ !v_1^k\ldots!v_n^k} (q',\rho_k',\delta_k),\ \rho_k' = \rho_k[v_i^*/x_i\ \forall o_i = ?x_i]$$

   $\rho_1 \sim_q^{live} \rho_2\ \Rightarrow\ \rho_1(h) = \rho_2(h),\ \forall o_i = !e_i\ \rho_1(e_i) = \rho_2(e_i),\ \rho_1' \sim_{q'}^{live} \rho_2'$

   $\delta_1 \approx_q^{live} \delta_2\ \Rightarrow\ \delta_1 \approx_{q'}^{live} \delta_2$

4. $\alpha = h \;\; x_1 := e_1, \ldots \; x_n := e_n$

$\forall k = 1,2 \;\; \forall i = 1,n \;\; \rho_k(e_i) = v_i^k, \;\; \rho_k(h) = true \;\; \Rightarrow$

$(q, \rho_k, \delta_k) \xrightarrow{\tau} (q', \rho_k, \delta_k), \;\; \rho'_k = \rho_k[v_1^k/x_1, \ldots \; v_n^k/x_n]$

$\rho_1 \sim_p^{live} \rho_2 \;\; \Rightarrow \;\; \rho_1(h) = \rho_2(h), \;\; \forall i = 1,n \;\; \rho_1(e_i) = \rho_2(e_i),$

$\rho_1[v_1^1/x_1, \ldots \; v_n^1/x_n] = \rho_2[v_1^2/x_1, \ldots \; v_n^2/x_n]$

$\delta_1 \approx_q^{live} \delta_2 \;\; \Rightarrow \;\; \delta_1 \approx_{q'}^{live} \delta_2$

■

Ce résultat s'avère très utile dans la pratique pour la génération des systèmes de transitions étiquetées associés aux programmes IF. En fait, étant donné son caractère statique, le test de l'équivalence d'activité ne nécessite pas l'exploration des successeurs. Elle peut se faire localement, en tenant compte des informations sur l'activité. Cette propriété permet la *réduction à la volée*, pendant la génération, des modèles associés aux programmes IF. En fait, une procédure de génération classique utilise l'égalité forte entre contextes pour décider si un nouvel état a été rencontré. Désormais, il est tout à fait possible de remplacer le test d'égalité par le test d'équivalence d'activité $\approx^{live}$ ce qui donnera comme résultat la génération directe du modèle quotient par rapport à $\approx^{live}$. Il faut noter aussi que cette réduction est exacte i.e, elle préserve toutes les propriétés du modèle, car $\approx^{live}$ est une bisimulation forte.

Finalement, il existe aussi une possibilité de rendre (partiellement) cette réduction au niveau syntaxique. Il s'agit de rajouter systématiquement des affectations remettant les variables inactives à des valeurs prédéfinies. Concrètement, cette transformation consiste à remplacer chaque transition $q \xrightarrow{\alpha} q'$ par :

$$q \xrightarrow{\alpha \;\; \{x := v_0 \mid x \in Live(q) \setminus Live(q')\}} q'$$

De cette manière, deux contextes sur les variables ne seront jamais différenciés à cause de variables inactives. Par contre, cette technique ne peut pas s'appliquer aux contextes sur les files dont une analyse plus profonde des contenus est nécessaire.

4.2 Analyse des domaines

La deuxième catégorie de techniques d'analyse statique concerne les domaines de valeurs des variables. Il s'agit de techniques permettant de déterminer statiquement des propriétés sur les ensembles de valeurs prises par les variables au moment de l'exécution.

Parmi ces techniques nous mentionnons la propagation de constantes [Kil73, ASU86, WZ91], la propagation d'intervalles [Cou78, Bou92], la propagation de polyèdres [CH78, Hal79], et la propagation d'invariants [BBM97, BL99]. Le principe de base de ces techniques est généralement le même : il s'agit de propager des contraintes sur les variables (i.e, constantes,

intervallesΓpolyèdresΓassertions) sur les transitions de l'automateΓjusqu'à ce que un certain point fixe est atteint.

En généralΓles résultats obtenus par l'analyse de domaines peuvent être utilisées de façons multiples. D'une partΓils peuvent servir au niveau syntaxiqueΓpour simplifier éventuellement le programme. D'autre partΓils peuvent aussi servir au niveau sémantique pour améliorer les phases ultérieures de validation e.gΓsimplifier la représentation des états ou des ensembles d'états.

Comme exemplesΓnous allons brièvement illustrer la technique de propagation de constantes [Kil73] et une technique syntaxique de génération d'invariants.

4.2.1 Propagation de constantes

Une variable est *constante* dans un état de l'automate si sa valeur peut être déterminée statiquement à cet état. L'objectif de la propagation de constantes est de trouver les variables constantesΓpour tous les états de contrôle de la spécification.

Soit $V = \{v_1, v_2 \dots\}$ l'ensemble de toutes les valeurs possibles des variables du programme. Nous notons par $\mathcal{V} = V \cup \{\bot, \top\}$ l'ensemble des valeurs étendu avec deux valeurs spéciales : $\bot$Γ*la valeur non-initialisée*Γet $\top$Γ*la valeur non-constante*. Sur $\mathcal{V}$ on considère la relation $\sqsubseteq$ définie comme le plus petit ordre partiel tel que $\bot \sqsubseteq v$ et $v \sqsubseteq \top$ pour toute valeur $v \in V$. On peut vérifier facilement que $(\mathcal{V}, \sqsubseteq)$ est un treillis complet de profondeur 3. Nous notons par $v_1 \sqcup v_2$ le plus petit majorant de v_1 et v_2 dans $(\mathcal{V}, \sqsubseteq)$.

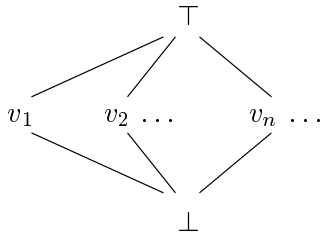

FIG. 4.3 – *Le treillis de valeurs étendues.*

Les contraintes manipulées par l'algorithme de propagation de constantes sont des *contextes étendus* sur les variables i.eΓdes applications $C : X \to \mathcal{V}$ associant une valeur constante $v \in V$ ou l'une des valeurs spéciales $\bot$ et $\top$ à chaque variable.

La relation d'ordre partiel $\sqsubseteq$ peut s'étendre naturellement aux contextes étendus sur les variables. On considère $C_1 \sqsubseteq C_2$ si et seulement si $C_1(x) \sqsubseteq C_2(x)$ pour toute variable $x \in X$. Là-aussiΓon peut vérifier que l'ensemble de contextes étendus $\mathcal{C}$ muni de l'ordre $\sqsubseteq$ est un treillis completΓde profondeur $3 \cdot |X|$. Nous allons noter aussi par $C_1 \sqcup C_2$ le plus petit majorant de contextes C_1 et C_2 ; en particulierΓon peut vérifier $(C_1 \sqcup C_2)(x) = C_1(x) \sqcup C_2(x)$ pour toute variable $x \in X$.

Pour chaque transition $q \xrightarrow{\alpha} q'$Γnous définissons le transformateur de contextes $Post^c(\alpha)$: $\mathcal{C} \to \mathcal{C}$Γdénotant l'effet de l'application de α :

$$Post^c(\alpha)(C) = C' \quad \text{où} \quad C'(x) = \begin{cases} C(e) & \text{si } x := e \in \alpha \\ C(x) & \text{si } x \notin Def(\alpha) \\ \top & \text{sinon} \end{cases} \quad \forall x \in X$$

Avec $C(e)$ nous avons notée le résultat de l'évaluation partielle de e dans le contexte étendu C : valeur non-constante $\top$ si au moins une des variables de e est non-constante constante v si toutes les variables de e sont constantes est e s'évalue à v et non-initialisée $\bot$ sinon.

Si $Const_0(q)$ dénote les valeurs initiales connues sur les variables à l'état q la propagation de constantes revient à chercher la plus petite solution du système d'équations suivant défini sur des contextes étendus :

$$\forall q' \in Q \quad Const(q') = \bigsqcup_{q \xrightarrow{\alpha} q'} Post^c(\alpha)(Const(q)) \tag{4.5}$$

Ce système peut être résolu de manière itérative : on commence avec des contextes associant la valeur non-initialisée pour toute variable. Ensuite on itère les équations jusqu'à ce que un point fixe est atteint. La convergence est assurée car pour toute action α l'opérateur $Post^c(\alpha)$ est monotones par rapport à l'ordre $\sqsubseteq$ et la profondeur du treillis de contextes étendus est bornée.

pour chaque état q `faire`
  $Const(q) := [\bot/x_1, \ldots \bot/x_n]$;
`répéter`
  `pour chaque état` q' `faire`
    $Const(q') = \bigsqcup_{q \xrightarrow{\alpha} q'} Post^c(\alpha)(Const(q))$
`jusqu'à convergence sur les` $Const$

Il existe des algorithmes plus efficaces de propagation qui exploitent la structure du graphe de contrôle. En général on peut itérer uniquement les équations sur les états dont les prédécesseurs ont été modifiées à l'itération précédente [ASU86] ou encore utiliser des stratégies plus efficace comme celle proposée par [Bou92]. De plus pour améliorer le résultat il est possible de prendre en compte des informations sur les variables constantes induites par les gardes [WZ91].

Exemple 4.4 *Pour l'automate présenté dans la figure 4.4, la propagation de constantes fournit comme résultat* $Const(q_0) = \langle \bot/x, \bot/y, \bot/z, \bot/t \rangle$, $Const(q_1) = \langle \top/x, \top/y, \bot/z, \bot/t \rangle$, $Const(q_2) = \langle 2/x, \top/y, \bot/z, \bot/t \rangle$, $Const(q_3) = \langle \top/x, 10/y, \bot/z, \bot/t \rangle$.

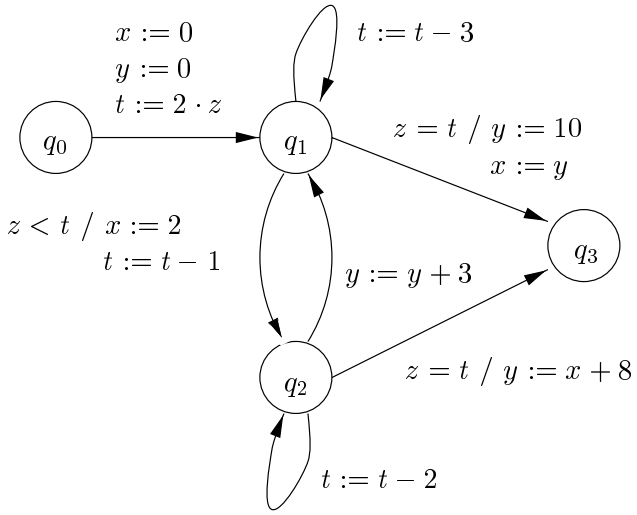

FIG. 4.4 – *Exemple de propagation de constantes.*

4.2.2 Propagation d'invariants

Un prédicat sur les variables est un *invariant* de l'automate dans un état si pour toute exécution il sera satisfait par les valeurs de variables à cet état. Nous nous intéressons à une catégorie particulière d'invariants générés de manière purement syntaxique à partir des gardes et des affectations de l'automate.

Pour chaque transition $q \xrightarrow{\alpha} q'$ nous définissons l'ensemble $\mathcal{L}(\alpha)$ de littéraux booléens produits par α i.e les expressions booléennes sur les variables toujours vraies après l'exécution de α :

$$\begin{aligned} \mathcal{L}(\alpha) \quad = \quad & \{\ h \mid h \in \alpha,\ vars(h) \cap Def(\alpha) = \emptyset\ \} \quad \cup \\ & \{\ x = e \mid x := e \in \alpha,\ vars(e) \cap Def(\alpha) = \emptyset\ \} \end{aligned}$$

Nous notons avec $\mathcal{L} = \cup_\alpha \mathcal{L}(\alpha)$ l'ensemble de tous les littéraux de l'automate. Nous cherchons des invariants I sous forme normale disjonctive $I \in \vee\wedge\mathcal{L} = \{\vee_i \wedge_j l_{ij} \mid l_{ij} \in \mathcal{L}\}$.

Nous munissons l'ensemble $\vee\wedge\mathcal{L}$ de la relation d'ordre partiel $\Rightarrow$ correspondante à l'implication syntaxique (sans interpréter les littéraux e.g $x < 1 \Rightarrow x < 1 \vee x < 2$ mais $x < 1 \not\Rightarrow x < 2$). On peut vérifier facilement que $(\vee\wedge\mathcal{L}, \Rightarrow)$ est un treillis complet de profondeur bornée à $2^{|\mathcal{L}|}$ avec la disjonction comme borne supérieure.

Pour chaque transition $q \xrightarrow{\alpha} q'$ nous définissons le transformateur d'invariants $Post^l(\alpha)$: $\vee\wedge\mathcal{L} \to \vee\wedge\mathcal{L}$:

$$Post^l(\alpha)(\vee_i \wedge_j l_{ij}) = \bigvee_i Post^l(\alpha)(\wedge_j l_{ij})$$

$$Post^l(\alpha)(\wedge_j l_{ij}) = \bigwedge_j \{ l_{ij} \mid vars(l_{ij}) \cap Def(\alpha) = \emptyset \} \wedge \bigwedge \{ l \mid l \in \mathcal{L}(\alpha) \}$$

Le calcul d'invariants revient à chercher la plus petite solution du système d'équations suivant :

$$\forall q' \in Q \quad Invrt(q') = \bigvee_{q \xrightarrow{\alpha} q'} Post^l(\alpha)(Invrt(q)) \tag{4.6}$$

Le système peut être résolu de manière itérativeΓpar un algorithme similaire au celui de la propagation de constantes. L'algorithme convergeΓétant donné que les équations sont monotones par rapport à l'implication et que la profondeur du treillis de prédicats considérés est bornée.

```
pour chaque état q faire
   Invrt(q) := false;
répéter
   pour chaque état q' faire
      Invrt(q') = ⋁_{q --α--> q'} Post^p(α)(Invrt(q));
jusqu'à convergence sur les Invrt
```

Exemple 4.5 *Pour l'automate présenté dans la figure 4.4, la propagation d'invariants fournit comme résultat* $Invrt(q_0) = false$, $Invrt(q_1) = x = 0 \wedge y = 0 \vee x = 2$, $Invrt(q_2) = x = 2$ *et* $Invrt(q_3) = x = 2 \wedge y = x + 8 \wedge z = t \vee x = y \wedge y = 10 \wedge z = t$.

Par rapport à d'autres techniques de générationΓcelle présentée ici est relativement faible : elle ne génère que des invariants syntaxiquesΓd'une forme très particulières. Les autres méthodes travaillent habituellement avec des invariants généraux Γsous forme de prédicats quelconques définis sur les variables du programme. En revancheΓelles doivent faire appel à des démonstrateurs automatiques pour réaliser toutes les opérations nécessaires e.gΓtester les implicationsΓou calculer la transformation à travers les transitions de l'automate.

Les invariants sont utilisés depuis longtemps par les méthodes déductives de validation de programmes séquentielsΓcomme des moyens auxiliaires pour réaliser ou simplifier les preuves intermédiaires. Plus récemmentΓdes techniques de génération d'invariants ont été définies et appliquées dans le cadre de la vérification déductive de programmes parallèles [BBM97Γ

BL99]. CependantΓles invariants peuvent être utilisés aussi dans le cadre de la vérification algorithmique – ils peuvent servir autant pour simplifier la représentation de l'espace des étatsΓ que pour améliorer certaines techniques d'explorationΓ comme par exemple dans [HD93]. Quelques exemples d'utilisation dans le contexte de l'analyse des programmes IF seront présentés dans le chapitre 5.

4.3 Calcul d'abstractions

L'abstraction est une technique qui consiste à construire un programme dans lequel les informations non-pertinentes pour la propriété à vérifier ont été éliminées. Cette construction préserve la satisfaction de la propriété si elle garantit une certaine relation entre le programme abstrait et le programme initial. GénéralementΓla relation qui lie le programme concret et le programme abstrait est une simulation [Mil80] et préserve la satisfaction des propriétés de sûreté.

Par la suiteΓnous allons décrire une technique très simple d'abstraction qui consiste simplement à éliminer de la spécification un certain nombre de variables. La démarche suivie est basée sur l'observation suivante : lorsque l'on simule de manière exhaustive une spécification sans évaluer toutes les valeurs des variables lors de la simulation on obtient un sur-ensemble du comportement de ce programme. En effetΓles gardes n'étant pas complètement évaluéesΓ beaucoup plus de transitions autorisées par la partie contrôle du programme sont exécutables. Par suiteΓil est possible de vérifier certaines propriétés sur le comportement ainsi obtenuΓet en particulier s'il offre bien *au moins* l'ensemble du service attendu. L'intérêt de ces vérifications est alors leur faible coût puisqu'elles sont effectuées sur un graphe de petite taille (seuls les états de contrôle de la spécification et certaines variables sont pris en compte).

ConcrètementΓsoit $P = (D, X, S, B, G, Q, T)$ un ATCD et $X^a \subseteq X$ un ensemble de variables à abstraire initialement donné. L'abstraction que nous proposons *élimine les variables de* X^a *et toutes leurs dépendances* en avant de la spécification P. Elle peut être vue comme une forme de *slicing* [Tip94] de P par rapport aux variables de $X \setminus X^a$.

Calcul de variables abstraites

Une variable sera dite *abstraite* dans un état si et seulement si soit elle appartient à l'ensemble X^aΓsoit elle a été définie en fonction d'une variable de X^a. Les variables abstraites sont définies comme la plus petite solution du système d'équations suivantΓpour chaque état et chaque transition :

$$\forall q' \in Q \quad Abstract(q') = X^a \cup \bigcup_{q \xrightarrow{\alpha} q'} Abs(q, \alpha) \cup \big(Abstract(q) \setminus Def(\alpha)\big) \tag{4.7}$$

où $Abs(q, \alpha) = \{x \mid x := e \in \alpha \ \wedge \ vars(e) \cap Abstract(q) \neq \emptyset\}$ est l'ensemble de variables abstraites par l'action α en connaissant les variables abstraites au départ. Le calcul de variables

abstraites pour chaque état peut se faire de manière itérativeΓpar l'algorithme suivant :

```
pour chaque état q faire
   Abstract(q) := X^a;
répéter
   pour chaque état q' faire
      Abstract(q') = X^a ∪ ⋃_{q -α-> q'} Abs(q, α) ∪ (Abstract(q) \ Def(α));
jusqu'à convergence sur les Abstract
```

Transformations sur le programme

Soit maintenant $q \xrightarrow{\alpha} q'$ une transition de P fixée. L'abstraction consiste à éliminer la garde et certaines des affectations de αΓsi elles utilisent des variables abstraites dans q. FormellementΓ avec les notations e^a et x^a pour dénoter le résultat de l'abstraction appliquée aux expressions et aux variables :

$$e^a = \begin{cases} e & \text{si } vars(e) \cap Abstract(q) = \emptyset \\ any & \text{sinon} \end{cases} \qquad \forall e \in Exp[X]$$

$$x^a = \begin{cases} x & \text{si } x \notin Abstract(q) \\ any & \text{sinon} \end{cases} \qquad \forall x \in X$$

l'action α devient α^aΓcas par cas :

$$\frac{h \;\; b?s(x_1, \ldots \; x_n)}{h^a \;\; b?s(x_1^a, \ldots \; x_n^a)} \qquad \frac{h \;\; g \; o_1 \ldots o_n \quad o_i' = \begin{cases} ?x_i^a & \text{si } o_i = ?x_i \\ !e_i^a & \text{si } o_i = !e_i \end{cases}}{h^a \;\; g \; o_1' \ldots \; o_n'}$$

$$\frac{h \;\; b!s(e_1, \ldots \; e_n)}{h^a \;\; b!s(e_1^a, \ldots \; e_n^a)} \qquad \frac{h \;\; x_1 := e_1, \ldots \; x_n := e_n}{h^a \;\; x_1^a := e_1^a, \ldots \; x_n^a := e_n^a}$$

Les gardes *any* et les affectations de *any* sont éliminées de α^a. FinalementΠl'abstraction de P étant donné l'ensemble X^a est l'ATCD $P^a = (D, X \setminus X^a, S, B, G, C, Q, T^a)$ où $T^a = \{q \xrightarrow{\alpha^a} q' \mid q \xrightarrow{\alpha} q'\}$.

Théorème 4.2 *L'élimination de variables est une abstraction.*

Preuve: On peut vérifier qu'il existe une relation de simulation définie sur le produit des états du système concret P et du système abstrait P^a. Plus exactementΓnous considérons la relation $\sqsubseteq$ entre les états de P et les états de P^aΓdéfinie de manière suivante :

$$(q,\rho,\delta) \sqsubseteq (q,\rho^a,\delta) \quad - \quad \forall x \notin Abstract(q)\ \rho(x) = \rho^a(x) \tag{4.8}$$

Cette relation est une simulationΓformellement si $(q,\rho,\delta) \sqsubseteq (q,\rho^a,\delta)$ nous avons

$$\begin{array}{l} \forall (q',\rho',\delta')\ \ (q,\rho,\delta) \xrightarrow{a} (q',\rho',\delta')\ \Rightarrow \\ \quad \exists (q',\rho^{a\prime},\delta')\ \ (q,\rho^a,\delta) \xrightarrow{a} (q',\rho^{a\prime},\delta')\ \wedge\ (q',\rho',\delta') \sqsubseteq (q',\rho^{a\prime},\delta') \end{array}$$

Soit donc $(q,\rho,\delta) \sqsubseteq (q,\rho^a,\delta)$ et $q \xrightarrow{\alpha} q'$ une transition. Nous considérons ci-après le cas où α est une affectation. Les autres cas sont similaires.

1. $\alpha = h\ \ x_1 := e_1, \ldots\ x_n := e_n$

   $\rho(h) = true,\ \ \forall i = 1,n\ \rho(e_i) = v_i\ \ \Rightarrow$

   $(q,\rho,\delta) \xrightarrow{\tau} (q',\rho',\delta),\ \ \rho' = \rho[v_1/x_1, \ldots v_n/x_n]$

   $\rho^a(h^a) = true,\ \ \forall i = 1,n\ \rho^a(e_i^a) = v_i^a\ \ \Rightarrow$

   $(q,\rho^a,\delta) \xrightarrow{\tau} (q',\rho^{a\prime},\delta),\ \ \rho^{a\prime} = \rho^a[v_1^a/x_1^a, \ldots v_n^a/x_n^a]$

   MaisΓen sachant $\forall x \notin Abstract(q)\ \rho(x) = \rho^a(x)$ nous avons :

   $\rho(h) = true\ \ \Rightarrow\ \ \rho^a(h^a) = true$

   $x \notin Abstract(q')\ \ \Rightarrow$

   - $x = x_i \in Def(\alpha)\ \ \Rightarrow\ \ x_i \notin Abs(q,\alpha)\ \wedge\ \rho^{a\prime}(x) = v_i^a = v_i = \rho'(x)$
   - $x \notin Def(\alpha)\ \ \Rightarrow\ \ x \notin Abstract(q)\ \wedge\ \rho^{a\prime}(x) = \rho^a(x) = \rho(x) = \rho'(x)$

En particulierΓtoute séquence d'exécution de P se retrouve parmi les séquences d'exécution de P^a. ∎

Exemple 4.6 *L'élimination de z dans l'automate 4.4 produit comme résultat l'automate de la figure 4.5.*

L'abstraction est en général indispensable pour la validation de programmes de grande taille. En particulierΓdes abstractions basées sur l'élimination de variablesΓaussi connues sous nom d'évaluations partiellesΓsuscitent actuellement beaucoup d'intérêt. Comme premier exempleΓ nous mentionnons le travail de [CGJ98] dans le contexte de la validation de programmes ouverts. Afin de trouver des blocages sous l'hypothèse de le plus général environnementΓ on élimine toute variable non-contrôlée par le programme i.eΓune variable dont la valeur

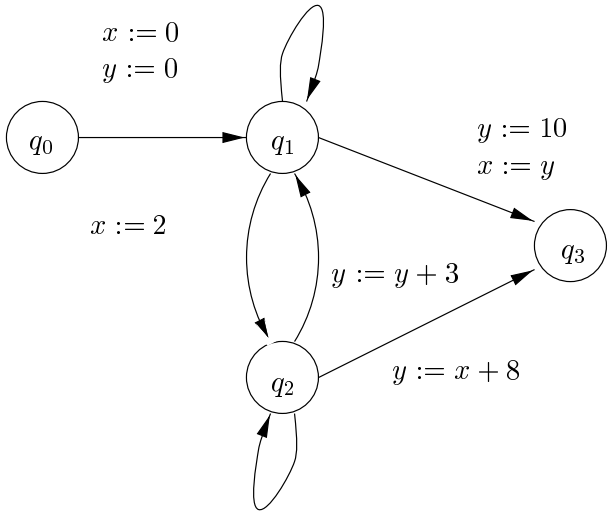

FIG. 4.5 – *Résultat d'élimination de variables.*

peut dépendre des valeurs reçues à travers des canaux de communication ouverts. Une autre approche est présentée dans [DH99] et consiste à éliminer des variables en fonction d'une propriété LTL à vérifierΓde telle sorte qu'elle reste préservée par l'abstraction. IciΓà partir des variables mentionnées explicitement dans la propriété on calcule les variables dont elles dépendent syntaxiquement dans le programmeΓet par suite on élimine les autres.

Chapitre 5

Simulation

En généralΓpar *simulation* nous entendons construire le modèle sémantique i.eΓle système de transitions étiquetéesΓd'une spécification IF. Cette construction est basée sur la sémantique dynamique de IF.

La place de la simulation dans la chaîne de validation à base de modèles est centrale. En faitΓ tous les algorithmes de validation travaillent au niveau du modèle et donc reposent implicitement sur l'existence d'un *simulateur* capable d'explorer ou de construire complètement ce modèle.

La simulation a comme exigence principale *l'efficacité* par rapport aux ressources standards de calculΓla mémoire et le temps nécessaire pour la réaliser. AinsiΓd'une part on cherche des représentations compactes pour des états et des ensembles d'états du modèle de programme IF. D'autre partΓon cherche des méthodes efficaces pour le calcul des transitionsΓi.eΓtrouver les successeurs ou les prédécesseurs d'un état ou d'un ensemble d'étatsΓadaptées aux représentations choisies. Ce ne sont pas de problèmes triviaux car les états sont des objets relativement compliquésΓcomportant des parties hétérogènesΓavec des fonctionnalités très différentes. De plusΓles ensembles d'états manipulés sont souvent infinisΓce qui entraîne la définition de représentations et d'opérations finies afin de garantir l'automatisation du processus de simulation.

Le principe que nous avons adopté pour la mise en œuvre de la simulation des programmes IF est illustré dans la figure 5.1. A partir d'un programme IF concret nous construisons une *bibliothèque de simulation* qui fournit un certain nombre de primitives dont les plus importantes sont :

1. une représentation concrète des états et des ensembles d'états
2. une manière concrète de calculer les transitions entre ces états

IntuitivementΓces primitives peuvent être définies à l'aide d'une *bibliothèque générique de représentations*Γcomportant des modules spécialisés pour la manipulation et le stockage efficace de certains objets intensivement utilisés. D'autre partΓafin d'offrir de meilleures

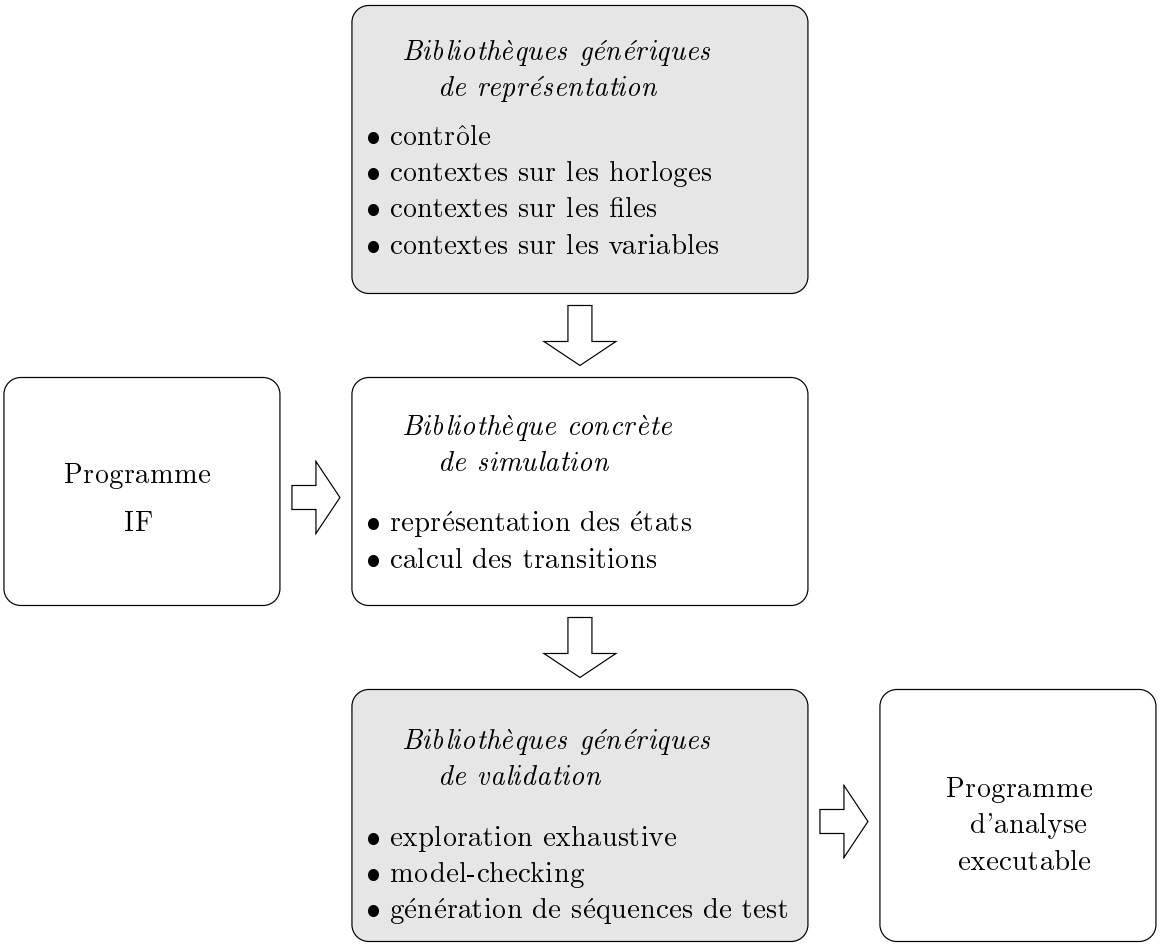

FIG. 5.1 – *Principe de la mise en œuvre.*

performancesΓles primitives doivent exploiter tout résultat utile obtenu par *analyse statique* sur le programme initial. Et finalementΓelles doivent être conçues en prenant en compte le fait qu'elles seront potentiellement utilisées par des *algorithmes génériques de validation* comme par exemple la simulation exhaustiveΓle model-checking de propriétés ou la génération de séquences de test.

Pour IFΓnous avons défini et expérimenté trois modes de simulation complémentaires. La *simulation énumérative* est basée sur une représentation individuelle des étatsΓpar des structures comportant respectivement le point de contrôle et les contextes sur les horlogesΓles files et les variables. La *simulation mixte* utilise une représentation des ensembles d'états où on combine une représentation symboliqueΓà base de contraintes arithmétiquesΓpour les horloges et une représentation énumérative pour le reste. FinalementΓla *simulation symbolique* est basée sur l'utilisation uniforme de diagrammes de décision binaires pour la représentation des ensembles quelconques d'états et de transitions.

5.1 Simulation énumérative

Afin de réaliser la simulation énumérativeΓnous avons défini une manière explicite pour représenter les états et pour calculer l'ensemble de leur successeurs.

5.1.1 Représentation des états

Les états du modèle sont représentés explicitement par une structure $(q, \rho, \delta, \gamma)$ contenant respectivement l'état de contrôle q et les contextes sur les horloges γΓsur les files δ et sur les variables ρ.

Contrôle

La représentation du contrôle ne pose pas de problème particulier. Comme l'ensemble d'états de contrôle Q est finiΓon peut supposer sans perte de généralité $Q = \{\ 1, 2, \ldots, m\ \}$. Un état individuel sera alors représenté par un entier $q \in Q$.

Horloges

Les contextes sur les horloges sont des applications $\gamma : C \to \mathbb{R}$ où C est l'ensemble d'horloges et $\mathbb{R}$ dénote l'ensemble des nombres réels. Il est clair que l'ensemble des contextes est infiniΓ dense et non-énumérableΓce qui fait que des représentations énumératives directes par des vecteurs de nombre réels $\langle \gamma(c_1), \ldots\ \gamma(c_n) \rangle$ sont mal adaptées pour la simulation exhaustive.

La solution adoptée est la *discrétisation* du temps et ainsi le transfert du problème de représentation vers un autre espaceΓcette fois ci discret et énumérable. Etant fixé un pas de discrétisation $\Delta \in \mathbb{R}^+$Γles contextes sur les horloges sont *arrondis* par des applications $\gamma_\Delta : C \to \mathbb{R}_\Delta$ où $\mathbb{R}_\Delta = \{\ k\Delta \mid k \in \mathbb{Z}\ \}$Γassociant à chaque horloge une valeur dans l'espace

réel discrétisé par Δ. Sans perte de généralité on peut considérer $\Delta = 1$ et les contextes deviennent simplement des applications $\gamma_1 : C \to \mathbb{Z}$ associant des valeurs entières aux horloges.

Une fois discrétiséesΓles horloges sont des variables entièresΓavec un comportement particulier. On peut même les considérer bornées car toutes les valeurs plus grandes que la plus grande constante testée dans le système sont équivalentes. Ces valeurs ne peuvent pas être distinguées à l'aide des gardes existantes dans le programme et sont habituellement fusionnées en une seule valeur spécialeΓl'infini ∞. Les contextes discrétisés seront alors représentés par des vecteurs d'entiers bornés $\langle \gamma_1(c_1), \ldots \gamma_1(c_n) \rangle$. Les primitives nécessaires à la manipulation sont respectivement l'évaluation de gardes $\underline{eval}(\gamma, h)$Γl'exécution des remises à zéro $\underline{assign}(\gamma, r)$Γet la progression du temps $\underline{timep}(\gamma, k)$.

FinalementΓnous remarquons que du point de vue sémantiqueΓla discrétisation du temps entraîne une perte d'information par rapport au cas dense. HeureusementΓcette perte est minime et même inexistante pour certains types d'automates temporisés [HMP92ΓAMP98].

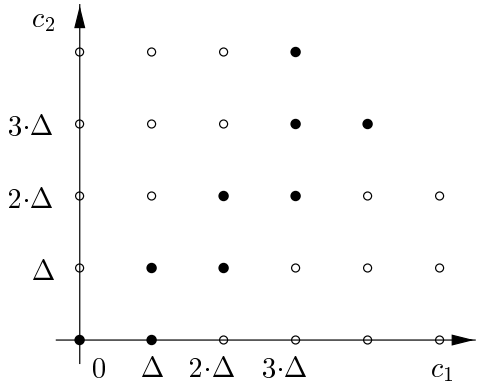

FIG. 5.2 – *Contextes discrétisés définis avec deux horloges c_1 et c_2.*

Files

GénéralementΓles contextes sur les files sont des applications $\delta : B \to S^*$ où B est l'ensemble de files et S est l'ensemble de signaux. Si les files ne sont pas bornées ou éventuellement si l'ensemble de signaux est infini alors l'ensemble de contextes est lui aussi infini.

La technique de représentation choisie consiste à voir chaque contexte δ comme un vecteur de mots de taille finie sur l'ensemble de signaux i.eΓ $\langle \delta(b_1), \ldots \delta(b_n) \rangle$ et de le représenter explicitement. Cette technique est partielle car elle permet uniquement la représentation d'un ensemble fini de contextes. Les primitives nécessaires de manipulation sont respectivement l'accès au signal en tête de files $\underline{front}(\delta, b)$ et l'exécution d'entrées $\underline{input}(\delta, b)$ et de sorties $\underline{output}(\delta, b, w)$.

Pour plus d'efficacitéΓnous nous sommes inspirés des techniques énumératives classiques pour la représentation des mots finis et des ensembles de mots finis. Nous avons expérimenté *la représentation avec partage de suffixes*Γqui nous a semblé particulièrement appropriée pour

la représentation de contextes. Le principe est illustré dans la figure 5.3. Même si au pireΓ cette représentation est exponentielle $\mathcal{O}(m^n)$ où m est le nombre de signaux et n est la taille maximale des filesΓles résultats pratiques ont montré l'intérêt d'avoir une telle représentation. Ces résultats sont présentés en détail dans la section 7.1.

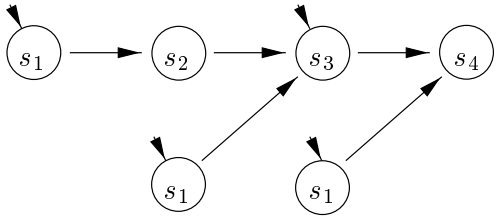

FIG. 5.3 – *Représentation de mots $s_1s_2s_3s_4$, $s_1s_3s_4$, s_3s_4 et s_1s_4 avec partage de suffixes.*

Variables

Les contextes sur les variables sont des applications $\rho : X \to V$ où X est l'ensemble de variables et V est l'ensemble des valeurs. Chaque contexte ρ est représenté alors explicitement en énumérant les valeurs $\langle \rho(x_1), \ldots \rho(x_n) \rangle$Γpour toutes les variables. Les primitives nécessaires de manipulation sont l'évaluation des expressions $\underline{eval}(\rho, e)$ et l'exécution des affectations $\underline{assign}(\rho, r)$.

Contrairement aux contextes sur les horloges ou filesΓle domaine de valeurs V est relativement amorphe : dans le cas le plus général on suppose qu'il est constitué de valeurs des types abstraits algébriques quelconques. Cela nous empêche de définir des techniques de représentation généralement efficaces carΓde telles techniques sont fortement dépendantes des types concrets utilisés. CependantΓon peut fournir quelques principes généraux en vue de simplifier les représentationsΓau vue des informations obtenues par l'analyse statique :

- une première source de simplification est l'activité des variables. Cela conduit à représenter uniquement les valeurs des variables actives dans qΓet ignorer les autres. ConcrètementΓsi $Live(q) = \{x_{i_1}, \ldots\ x_{i_n}\}$ on représente uniquement $\langle \rho(x_{i_1}), \ldots \rho(x_{i_n}) \rangle$.
- une deuxième source de simplification sont les informations sur les domaines de valeurs. D'une partΓon peut représenter uniquement les valeurs des variables non-constantes et se débarrasser des autres car leurs valeurs peuvent toujours être retrouvées à partir de l'état du contrôle courant. AinsiΓon représente uniquement $\langle \rho(x_{i_1}) \cdots \rho(x_{i_k}) \rangle$ pour toute variable x_{i_j} telle que $Const(q)(x_{i_j}) = \top$. D'autre partΓon peut utiliser des invariants particuliers pour simplifier encore les contextes. Par exempleΓon peut se servir de dépendances linéaires entre les variables i.eΓdes égalités $x_i = e_i$ invariantes pour certains états. Dans ce casΓla valeur de x_i peut être enlevée de la représentation car on peut toujours la calculer à partir des autres intervenant dans e.

5.1.2 Représentation des transitions

Chaque transition t du programme sera représentée par une procédure $Fire^e[t]$ qui l'interprète : cette procédure prend en entrée un état du modèle (contrôle plus contextes) et met à jour l'ensemble de ses successeurs *succs* ainsi que la condition booléenne de progression du temps *tpc*.

Le fonctionnement d'une telle procédure comporte les phases suivantes :

1. *évaluation des gardes*

   Les gardes portant sur les horloges $guard^c(t)$ et sur les variables discrètes $guard^x(t)\Gamma$ ainsi que sur les contenus de files dans les cas de transitions asynchronesΓsont évaluées dans les contextes courants pour détecter si la transition est ou non franchissable. Si nonΓla transition ne produit pas de successeurs et ne modifie pas la condition de progression du temps.

2. *calcul des successeurs*

   Les actions d'entrées-sorties $action(t)\Gamma$puis les affectations $reset^c(t)$ et $reset^x(t)$ sont exécutées dans les contextes courants pour calculer l'état ou les états successeurs. Pour une transition asynchroneΓdes signaux sont consommés ou ajoutés dans les files. Pour une transition synchroneΓdes valeurs sont échangées par l'intermédiaire d'une porte de synchronisation. On remarque aussi dans ce cas le non-déterminisme produit au niveau des variables présentes. FinalementΓles étiquettes correspondantes aux successeurs sont aussi synthétisées.

3. *mise à jour de la condition de progression du temps*

   La condition de progression du temps *tpc* sera mise à fausse si la transition est urgente ou si elle est retardable mais infranchissable plus tard à cause de sa garde temporelle. SinonΓla condition de progression du temps restera inchangée.

FinalementΓune procédure $Fire^e[time]$ est aussi définie pour représenter les transitions temporelles. Son fonctionnement consiste uniquement à incrémenter les valeurs d'horloges d'une unité de temps.

Transitions asynchrones

```
procédure Fire^e[t](in (q, ρ, δ, γ), in/out tpc, in/out succs) ≡

   (* évaluation de gardes *)
   si source(t) ≠ q
      retourne;
   si eval(γ, guard^c(t)) = false alors
      retourne;
   si eval(ρ, guard^x(t)) = false alors
      retourne;
   si ∃b?s(x⃗) ∈ action(t) tel que front(δ, b) ≠ s(v⃗) alors
      retourne;

   (* mise à jour de successeurs *)
   soit (q', ρ', δ', γ') := (target(t), ρ, δ, assign(γ, reset^c(t)));
   soit a := ε;
   pour chaque entrée b?s(x⃗) ∈ action(t) faire
      soit s(v⃗) := front(δ', b);
      δ' := input(δ', b);
      ρ' := assign(ρ', {x⃗ := v⃗});
      a := a · b?s(v⃗);
   pour chaque sortie b!W ∈ action(t) faire
      soit w := eval(ρ', W);
      δ' := output(δ', b, w);
      a := a · b!w;
   ρ' := exec(ρ', reset^x(t));
   succs := succs ∪ {(a, (q', ρ', δ', γ'))};

   (* mise à jour de tpc *)
   si urgency(t) = eager alors
      tpc := false;
   si urgency(t) = delayable alors
      tpc := tpc ∧ ¬eval(timep(γ, 1), guard^c(t));
```

Transitions synchrones

```
procédure Fire^e[t](in (q, ρ, δ, γ), in/out tpc, in/out succs) ≡

   (* évaluation de gardes *)
   si source(t) ≠ q
      retourne;
   si eval(γ, guard^c(t)) = false alors
      retourne;
   si eval(ρ, guard^x(t)) = false alors
      retourne;

   (* mise à jour de successeurs *)
   soit (q', ρ', δ', γ') := (target(t), ρ, δ, assign(γ, reset^c(t)));
   soit a := ε;
   pour chaque valuation v_1, ... v_n telle que
         v_i = eval^ρ(ρ, e_i) si o_i =!e_i ∈ action(t) et
         v_i ∈ domain(x_i) si o_i =?x_i ∈ action(t) faire
      soit ρ'' := assign(ρ, {x_i := v_i | o_i =?x_i};
      ρ' := assign(ρ'', reset^x(t));
      a := g !v_1 ... !v_n;
      succs := succs ∪ {(a, (q', ρ', δ', γ'))};

   (* mise à jour de tpc *)
   si urgency(t) = eager alors
      tpc := false;
   si urgency(t) = delayable alors
      tpc := tpc ∧ ¬eval(timep(γ, 1), guard^c(t));
```

Transitions temporelles

```
procédure Fire^e[time](in (q, ρ, δ, γ), in/out tpc, in/out succs) ≡

   succs := succs ∪ {(time(1), (q, ρ, δ, timep(γ, 1)))};
```

FinalementΓces procédures peuvent être générées directement dans un langage exécutableΓ comme par exemple C. De cette manièreΓle code généré peut être encore optimisé à l'aide des informations calculées par analyse statique. De plusΓétant basée sur la compilation et l'exécution du code engendréΓcette approche est généralement plus efficace en temps que d'autres basées sur l'interprétation dynamique des règles sémantiques

5.1.3 Exploration énumérative

Les procédures construites à partir de transitions sont groupées dans une primitive globale $Post^e$ permettant le calcul de successeurs d'un état par toutes les transitions du programme. Ça sera normalement la seule primitive accessible à l'extérieur :

```
procédure Post^e(in (q, ρ, δ, γ), out succs) ≡

   soit tpc := true;
   pour chaque transition t faire
      Fire^e[t]((q, ρ, δ, γ), tpc, succs);
   si tpc = true alors
      Fire^e[time]((q, ρ, δ, γ), tpc, succs);
```

Son fonctionnement consiste à exécuter d'abord les procédures associées aux transitions discrètesΓpuis celles des transitions temporellesΓsi c'est possible. De cette manière pendant l'exécution des transitions discrètes on récupère à la fois les successeurs et aussi la condition de progression du temps globalement induite. Si finalementΓcette condition est fausseΓelle bloquera le franchissement des transitions temporelles.

5.2 Simulation mixte

La deuxième méthode de simulation mise en œuvre pour des programmes IF combine des techniques énumératives et symboliques aussi bien pour la représentation des ensembles d'états que pour le calcul de transitions entre ces ensembles.

5.2.1 Représentation des états

Nous considérons des ensembles d'états (ou *états symboliques*) de la forme $(q, \rho, \delta, \Gamma)$ où q est l'état de contrôleΓρ est un contexte sur les variablesΓδ est un contexte sur les files et Γ est un ensemble de contextes sur les horloges. Les parties contrôleΓfiles et variables sont représentées explicitementΓde la même manière que pour la simulation énumérative. Les ensembles de

contextes sur les horloges sont des *zones* i.eΓdes polyèdres convexes particuliers dans l'espace dense de contextesΓadmettant une représentation efficace.

Horloges

Les techniques existantes de vérification pour des automates temporisés sont basées implicitement ou explicitement sur la construction du *graphe de régions* [ACD93]. IntuitivementΓil existe une relation d'équivalence unifiant les contextes sur les horloges qui rendent possibles les mêmes séquences de transitions discrètes. Cette équivalenceΓnotée par $\sim$Γest définie formellement de manière suivante :

Définition 5.1 ($\gamma_1 \sim \gamma_2$) *Soit $x \in \mathbb{R}$ un nombre réel. Nous notons par $\lfloor x \rfloor$ (resp. par $\lceil x \rceil$) la partie entière (resp. fractionnaire) de x i.e, le plus grand entier plus petit que x (resp. $x - \lfloor x \rfloor$). Deux contextes γ_1, γ_2 sont équivalents si et seulement si :*

1. $\forall c_i \in C \quad \lfloor \gamma_1(c_i) \rfloor = \lfloor \gamma_2(c_i) \rfloor$
2. $\forall c_i, c_j \in C \quad \lceil \gamma_1(c_i) \rceil = \lceil \gamma_1(c_j) \rceil - \lceil \gamma_2(c_i) \rceil = \lceil \gamma_2(c_j) \rceil$
3. $\forall c_i, c_j \in C \quad \lceil \gamma_1(c_i) \rceil < \lceil \gamma_1(c_j) \rceil - \lceil \gamma_2(c_i) \rceil < \lceil \gamma_2(c_j) \rceil$

Les classes d'équivalence de $\sim$ sont appelées *régions*. Le graphe des régions d'un automate temporisé est défini comme le quotient du système de transitions étiquetées associé à l'automate par la sémantique opérationnelle modulo l'équivalence $\sim$. Ce graphe est d'état fini et ses transitions correspondent à des combinaisons de transitions temporelles et de transitions discrètes. Il constitue une abstraction finie et exacte de l'ensemble des comportementsΓce qui rend possible l'analyse exhaustive des automates temporisés.

CependantΓle graphe des régions est relativement grandΓsa taille est de l'ordre $\mathcal{O}(K^n \cdot n!)$ où n est le nombre d'horloges et K est la plus grande constante testée dans le programmeΓce qui fait que sa construction directe est souvent impossible dans des exemples réels. Dans la pratique de vérification des automates temporisés on préfère l'utilisation de *zones*Γc'est-à-dire des unions convexes de plusieurs régions. En faitΓl'utilisation de zones n'impose pas la fragmentation à priori de l'espace des contextesΓainsi des régions comportant des contextes avec les mêmes comportements futurs restent groupés à l'intérieur d'une même zone pendant l'analyse du système.

Les zones et les régions construites à base de l'équivalence $\sim$ sont en fait des polyèdres qui peuvent se représenter par des contraintes linéaires particulières sur les horloges. Il s'agit de contraintes de la forme $c_i \# k$ ou $c_i - c_j \# k$ où c_i, c_j sont des horlogesΓ$\#$ est une relation parmi $<, \leq, =, \geq, >$ et k est une constante entière. Un tel ensemble de contraintes peut être manipulé soit directementΓsoit mis sous la forme d'une *matrices de bornes de différences* (DBM) [ACD93]. Ces matrices sont une représentation canonique de taille $\mathcal{O}(n^2)$Γoù n est le nombre d'horlogesΓet permettent une implantation efficace des opérations comme l'intersection $\cap$ et la progression du temps $\nearrow$. MalheureusementΓelles ne peuvent pas représenter des ensembles non-convexes et donc des opérations comme l'union $\cup$ ou la complémentation

¬ nécessitent l'utilisation explicite des unions de plusieurs zones. C'est néanmoins la représentation la plus utilisée par les outils de vérification dédiés aux systèmes temporisés comme par exemple KRONOS [Yov97ΓBDM+98] et UPPAAL [LPY97].

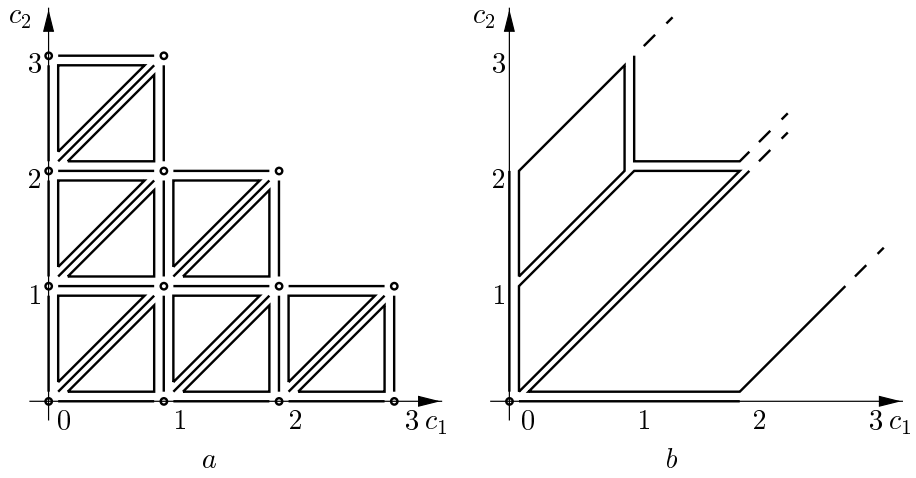

FIG. 5.4 – *Zones (a) et régions (b) définies avec deux horloges c_1 et c_2.*

Plus récemmentΓ*les diagrammes de décision temporisés*Γou CDDs (*Clocked Decision Diagrams*)Γont été proposés dans [BLP+99]. IntuitivementΓil s'agit de graphes orientés dont les nœuds sont étiquetés par des différences d'horloges $c_i - c_j$ et les arcs par des intervalles de valeurs $[k_i, k_j]$. Cette représentation n'est plus limitée aux ensembles convexes et toutes les opérations nécessaires ($\cap\Gamma\cup\Gamma\neg\Gamma\nearrow\Gamma$etc) peuvent se réaliser efficacement avec des algorithmes sur des graphes. Par contreΓce n'est pas une représentation canoniqueΓce qui introduit un surcoût non négligeable pour tester l'égalité.

5.2.2 Représentation des transitions

De même manière que pour la simulation énumérativeΓon représente chaque transition t par une procédure $Fire^m[t]$ qui prend en entrée un état symbolique $(q, \rho, \delta, \Gamma)$ et met à jour l'ensemble de ses successeurs *succs* plus un ensemble de *zones de progression du temps tpz*. IntuitivementΓl'union de ces zones sera l'ensemble (à priori non-convexe) de contextes atteignables à partir de Γ par des transitions temporelles tant que les urgences des transitions le permettent.

Le fonctionnement de la procédure est relativement le même que avant. Quelques différences concernent notamment l'utilisation de zones et sont détaillées par la suiteΓpour chacune de phases :

1. *évaluation de gardes*

   L'évaluation de la garde temporelle $guard^c(t)$ revient maintenant au test de l'intersection vide avec la zone de départ i.eΓΓ $\cap\, guard^c(t) \neq \emptyset$.

2. *calcul de successeurs*

   La zone de chaque successeur symbolique est obtenue en appliquant les remises à zéro $reset^c(t)$ dans l'intersection $\Gamma \cap guard^c(t)$. Si le résultat n'est pas une zone il faut le décomposer en plusieurs zones et générer plusieurs successeurs.

3. *mise à jour de zones de progression du temps*

   Les zones de progression du temps locales tpz_t induites par la transition courante dénotent l'ensemble de contextes atteignables à partir de Γ par des transitions temporelles tant que l'urgence de t le permet. Il s'agit des ensembles $\nearrow (\Gamma \cap \neg guard^c(t)) \cap \neg guard^c(t)$ si t est urgente$\Gamma \nearrow (\Gamma \cap \neg guard^c(t)\downarrow) \cap \neg guard^c(t)\downarrow$ si t est retardable et respectivement $\nearrow \Gamma$ si t est paresseuse.

   Les zones de progression de temps locales tpz_t sont *synchronisées* avec les zones de progression globales tpz. Il s'agit en fait de trouver l'ensemble de contextes atteignables par des transitions temporelles en prenant en compte les contraintes induites par plusieurs transitions. IntuitivementΓles transitions temporelles permises doivent se synchroniser. FormellementΓnous avons la définition suivante pour l'opération de synchronisation $\otimes$ sur les zones de progression du temps :

$$tpz_1 \otimes tpz_2 = \{\ \Gamma_1 \cap \Gamma_2 \mid \Gamma_1 \in tpz_1,\ \Gamma_2 \in tpz_2\ , \Gamma_1 \cap \Gamma_2 \neq \emptyset\ \}$$

FinalementΓla procédure $Fire^m[time]$ encode l'exécution de transitions temporelles. Etant donné un ensemble de zones de progression de temps $tpz\Gamma$ elle consiste à construire des successeurs symboliques ayant les mêmes composantes discrètes et une zone de progression pour les contextes sur les horloges.

Transitions asynchrones

```
procédure Fire^m[t](in (q, ρ, δ, Γ), in/out tpz, in/out succs) ≡

   (* évaluation de gardes *)
   si source(t) ≠ q
      retourne;
   si Γ ∩ guard^c(t) = ∅ alors
      retourne;
   si eval(ρ, guard^x(t)) = false alors
      retourne;
   si ∃b?s(x⃗) ∈ action(t) tel que front(δ, b) ≠ s(v⃗) alors
      retourne;

   (* mise à jour de successeurs *)
   soit (q', ρ', δ', Γ') := (target(t), ρ, δ, assign(Γ ∩ guard^c(t), reset^c(t)));
   soit a := ε;
   pour chaque entrée b?s(x⃗) ∈ action(t) faire
      soit s(v⃗) := front(δ', b);
      δ' := input(δ', b);
      ρ' := assign(ρ', {x⃗ := v⃗});
      a := a · b?s(v⃗);
   pour chaque sortie b!W ∈ action(t) faire
      soit w := eval(ρ', W);
      δ' := output(δ', b, w);
      a := a · b!w;
   ρ' := assign(ρ', reset^x(t));
   succs := succs ∪ {(a, (q', ρ', δ', Γ'))};

   (* mise à jour de tpz *)
   si urgency(t) = eager alors
      soit tpz_t :=↗ (Γ ∩ ¬guard^c(t)) ∩ ¬guard^c(t);
   si urgency(t) = delayable alors
      soit tpz_t :=↗ (Γ ∩ ¬guard^c(t)↓) ∩ ¬guard^c(t)↓;
   si urgency(t) = lazy alors
      soit tpz_t := {↗ Γ};
   tpz := tpz ⊗ tpz_t;
```

Transitions synchrones

```
procédure Fire^m[t](in (q, ρ, δ, Γ), in/out tpz, in/out succs) ≡

   (* évaluation de gardes *)
   si source(t) ≠ q
      retourne;
   si Γ ∩ guard^c(t) = ∅ alors
      retourne;
   si eval(ρ, guard^x(t)) = false alors
      retourne;

   (* mise à jour de successeurs *)
   soit (q', ρ', δ', Γ') := (target(t), ρ, δ, assign(Γ ∩ guard^c(t), reset^c(t)));
   soit a := ε;
   pour chaque valuation v_1, ... v_n telle que
         v_i = eval(ρ, e_i) si o_i =!e_i ∈ action(t) et
         v_i ∈ domain(x_i) si o_i =?x_i ∈ action(t) faire
      soit ρ'' := assign(ρ, {x_i := v_i | o_i =?x_i};
      ρ' := assign(ρ'', reset^x(t));
      a := g !v_1 ... !v_n;
      succs := succs ∪ {(a, (q', ρ', δ', Γ'))};

   (* mise à jour de tpz *)
   si urgency(t) = eager alors
      soit tpz_t :=↗ (Γ ∩ ¬guard^c(t)) ∩ ¬guard^c(t);
   si urgency(t) = delayable alors
      soit tpz_t :=↗ (Γ ∩ ¬guard^c(t)↓) ∩ ¬guard^c(t)↓;
   si urgency(t) = lazy alors
      soit tpz_t := {↗ Γ};
   tpz := tpz ⊗ tpz_t;
```

Transitions temporelles

```
procédure Fire^m[time](in (q, ρ, δ, Γ), in/out tpz, in/out succs) ≡

   pour chaque zone Γ' ∈ tpz faire
      succs := succs ∪ {(time, (q, ρ, δ, Γ'))};
```

5.2.3 Exploration mixte

FinalementΓles procédures $Fire^m[t]$ construites à partir de transitions sont groupées dans une seule primitive globale $Post^m$ calculant l'ensemble de tous les successeursΓdiscrètes ou temporellesΓpour un état symbolique donné. Le fonctionnement de cette procédure est similaire au cas énumératif :

```
procédure Post^m(in (q, ρ, δ, Γ), out succs) ≡

   tpz := {↗ Γ};
   pour chaque transition t faire
      Fire^m[t]((q, ρ, δ, Γ), tpz, succs);
   si tpz ≠ ∅ alors
      Fire^m[time]((q, ρ, δ, Γ), tpz, succs);
```

IntuitivementΓon itère d'abord les transitions discrètesΓpour calculer d'une part les successeurs discrètes et d'autre part pour identifier les zones de progression du temps. FinalementΓ si de telles zones existentΓon génère aussi des successeurs temporels.

Il faut noter que cette primitive peut être utilisée pour en construire d'autresΓplus adaptées pour la vérification temporelle de spécifications. Par exempleΓon peut facilement définir une primitive qui fait d'abord l'avancement maximal du tempsΓensuite tire une transition discrèteΓou le contraireΓpar des appels successifs à notre procédure $Post^m$.

5.3 Simulation symbolique

L'approche suivie pour la simulation symbolique est centrée autour de l'utilisation de représentation symbolique à base de *diagrammes de décision binaires* [Bry86] (*Binary Decision*

Diagrams BDDs). Cette approche est limitée aux *programmes* IF *finis* : l'idée principale est de coder symboliquement les ensembles d'états et de transitions d'un programme IF par des prédicats portant sur le point de contrôleΓles valeurs d'horloges et de variables et sur les contenus de files.

5.3.1 Représentation des états

Nous représentons des ensembles d'états par de prédicats de la forme $R^s(q, X, B, C)$ où q est une variable de contrôleΓX est l'ensemble de variables discrètesΓB est l'ensemble de files et C est l'ensemble de horloges. Ces prédicats sont obtenus en composant des prédicats élémentaires définis pour chacune de composantes :

- *contrôle* : le contrôle est représenté par une variable finie $q \in Q$. Des ensembles d'états sont représentés par des prédicats portant sur cette variable q.
- *horloges* : avec l'interprétation discrète du temps (section 5.1)Γles horloges C sont vues comme des variables entières bornées. Les gardes temporelles h^c sont codées naturellement par des prédicats $h^c(C)$. Les remises à zéro r^c sont codées par des prédicats $r^c(C,C')$ portant sur deux copies de l'ensemble d'horlogesΓC et C'Γdénotant les valeurs avant et après l'exécution. FinalementΓla progression du temps est aussi codée par un prédicat $timep(C, C')$.
- *files* : les files B sont vues comme des variables structurées finies. En particulierΓon considère les primitives $in(X, B, X', B')$ et $out(X, B, B')$ codant respectivement l'effet d'une entrée *in* ou d'une sortie *out* sur les contenus de files BΓB' et sur les valeurs de variables discrètes XΓX'Γavant et après l'exécution.
- *variables* : des prédicats sur les variables discrètes sont définies d'une manière naturelle. En particulierΓnous considérons la représentation des gardes h^x par de prédicats $h^x(X)$ et des affectations parallèles r^x par de prédicats $r^x(X, X')$.

Afin d'obtenir des meilleurs performances lors de la manipulationΓnous disposons d'un certain nombre d'options généralement applicablesΓdont les plus importantes sont :

- *ordre sur les variables de codage*

  L'ordre est un paramètre essentiel de la performance tant du point de vue de la représentation des étatsΓque de l'explorationΓcar il détermine directement la taille des BDDs manipulés. GénéralementΓtrouver l'ordre optimal est un problème NP-complet [FS90] ce qui fait queΓpour trouver un bon ordre on utilise plutôt des heuristiquesΓsouvent automatisablesΓcomme par exemple celle proposée par [Kam90]. SinonΓon peut envisager l'utilisation des techniques d'ordonnancement dynamique de variables [BRB90].

- *informations calculées par analyse statique*

  Les informations obtenues par analyse statique peuvent être utilisées pour simplifier toutes ces représentations. D'abordΓles variables inactives peuvent être éliminées par

des quantifications existentielles car elles n'apportent aucune information utile. Ensuite les informations sur les domaines effectifs des variables peuvent servir pour réduire le nombre des variables booléennes nécessaires au codage. Finalement les variables constantes ou encore tout invariant connu sur les variables peuvent servir pour simplifier les BDDs pendant leur manipulation [HD93].

5.3.2 Représentation des transitions

Chaque transition t se représente par un prédicat $Trans[t]$ portant sur deux ensembles de variables les valeurs avant $q X B C$ et après $q' X' B' C'$ l'exécution de la transition. Ces prédicats sont construits à partir de prédicats élémentaires définis sur les composantes. Généralement il s'agit de composer des prédicats correspondants aux gardes et aux actions contenues dans la transition :

Transitions asynchrones

$$
\begin{array}{l}
\texttt{relation}\ Trans^s[t](q, X, B, C, q', X', B', C') \ \equiv \\
\\
\quad q = source(t) \ \wedge \\
\quad guard^c(t)(C) \ \wedge \\
\quad guard^x(t)(X) \ \wedge \\
\quad \exists X'' \ \exists B'' \\
\qquad input(t)(X, B, X'', B'') \ \wedge \\
\qquad output(t)(X'', B'', B') \ \wedge \\
\qquad reset^x(t)(X'', X') \ \wedge \\
\quad reset^c(t)(C, C') \ \wedge \\
\quad q' = target(t)
\end{array}
$$

Transitions synchrones

$$
\begin{array}{l}
\texttt{relation}\ Trans^s[t](q, X, B, C, q', X', B', C') \ \equiv \\
\\
\quad q = source(t) \ \wedge \\
\quad guard^c(t)(C) \ \wedge \\
\quad guard^x(t)(X) \ \wedge \\
\quad \exists X'' \\
\qquad sync(t)(X, X'') \ \wedge \\
\qquad reset^x(t)(X'', X') \ \wedge \\
\quad reset^c(t)(C, C') \ \wedge \\
\quad q' = target(t)
\end{array}
$$

Une réduction significative est obtenue par la projection des variables inactives de la représentation des transitions et des ensembles d'états. Les valeurs de ces variables n'apportent aucune information utile pour l'exploration du système. La prise en compte de l'activité est très simple. Lorsque on code les transitions élémentaires on considèreΓnormalementΓque les variables affectées prennent des nouvelles valeursΓtant que les autres gardent leurs valeurs initiales. MaintenantΓen fonction l'activité à l'état d'arrivé nous considérons que les variables inactives *oublient* leurs valeursΓce qui revient à les éliminer par une quantification existentielle. ConcrètementΓpour chaque transition t si $Live(target(t)) = \{x_{i_1}, \ldots\ x_{i_n}\}$ on représente uniquement :

$$\exists x'_{i_1}\ \ldots\ \exists x'_{i_n}\ Trans^s[t](q, X, B, C, q', X', B', C')$$

Transitions temporelles

FinalementΓun prédicat $Trans[time]$ encode toutes les transitions temporelles. Pour simplifier sa définitionΓnous considérons que les transitions sont soit urgentes soit paresseuses. Ce prédicat est défini de manière suivante :

$$\texttt{relation}\ Trans^s[time](q,X,B,C,q',X',B',C') \equiv$$

$$\begin{array}{l} \forall q''\ \forall X''\ \forall B''\ \forall C'' \\ \quad \bigwedge\{\neg Trans^s[t](q,X,B,C,q'',X'',B'',C'') \mid urgency(t) = eager\} \\ \wedge \\ q = q' \wedge \\ X = X' \wedge \\ B = B' \wedge \\ timep(C,C') \end{array}$$

Transition globale

La relation de transition globale associée au programme IF est obtenue en faisant la disjonction de prédicats associés aux transitions. En faitΓil s'agit de faire l'union de toutes les transitions définies dans le système :

$$\texttt{relation}\ Trans^s(q,X,B,C,q',X',B',C') \equiv$$

$$\begin{array}{l} \bigvee_t Trans^s[t](q,X,B,C,q',X',B',C') \vee \\ Trans^s[time](q,X,B,C,q',X',B',C') \end{array}$$

Nous avons les options suivantes concernant la manipulation de la transition globale :

- *partitionnement de la relation de transition*

  La relation de transition globale peut être représentée soit par un seul BDDΓou partitionnée en plusieurs BDDs. La première solution s'avère plus efficace si le BDD global a une taille raisonnable et rend possible le calcul de successeurs. SinonΓla deuxième solution est recommandée car elle permet toujours de représenter la relation de transitionΓ et introduit un surcoût relativement faible dans le calcul de successeurs.

- *fermeture transitive*

  Une deuxième technique en vue d'accélérer l'exploration c'est la fermeture transitive de la relation de transition. L'idée est simpleΓune fois la relation de transition $Trans^s$ codéeΓil est envisageable de calculer les relations composées ${Trans^s}^2\Gamma{Trans^s}^3\Gamma\ldots$

$Trans^{s*}$. En outre si la transition globale est décomposée on peut faire la même chose pour ses composantes. L'utilisation de la fermeture transitive peut permettre si on peut la calculer des gains considérables pour le calcul des états atteignables.

5.3.3 Exploration symbolique

Le BDD $Trans^s$ est utilisé pour calculer aussi bien les successeurs que les prédécesseurs d'un ensemble d'états (R^s) représenté aussi par un BDD de la manière suivante :

$$
\begin{array}{rcl}
Post^s(R^s)\ (q', X', B', C') & = & \exists q\ \exists X\ \exists B\ \exists C \\
 & & \quad R^s(q, X, B, C)\ \wedge \\
 & & \quad Trans^s(q, X, B, C, q', X', B', C') \\
 & & \\
Pre^s(R^s)\ (q, X, B, C) & = & \exists q'\ \exists X'\ \exists B'\ \exists C' \\
 & & \quad Trans^s(q, X, B, C, q', X', B', C')\ \wedge \\
 & & \quad R^s(q', X', B', C') \\
 & & \\
\widetilde{Pre^s}(R^s)\ (q, X, B, C) & = & \forall q'\ \forall X'\ \forall B'\ \forall C' \\
 & & \quad Trans^s(q, X, B, C, q', X', B', C')\ \Rightarrow \\
 & & \quad R^s(q', X', B', C')
\end{array}
$$

La simulation symbolique présente un certain nombre d'avantages. D'une part elle assure une représentation canonique et uniforme des ensembles quelconques d'états et de transitions. D'autre part des implantations efficaces existent aussi bien pour les opérations ensemblistes que pour le calcul des successeurs. Cette représentation nous a permis de traiter des exemples de taille raisonnable (voir par exemple la section 7.3 ou [ABK+97, BMPY97, BMT99]). Cependant, généralement on ne peut pas conclure que c'est la meilleure solution car :

- les horloges sont fortement synchronisées par les transitions temporelles ce qui fait que les variables binaires utilisées pour leur codage sont dépendantes ; de plus, le nombre des ces variables est strictement dépendant de la taille du domaine des valeurs d'horloges, et ainsi de valeurs de constantes utilisées dans la spécification et du pas de discrétisation choisi.
- malgré l'avantage théorique par rapport à une représentation énumérative, à priori exponentielle, les représentations symboliques de contenus de queues explosent aussi dès qu'on considère une taille raisonnable de files. En fait, il semble absolument nécessaire de prendre en compte des informations auxiliaires, comme par exemple l'activité ou certains invariants calculés par analyse statique, afin de simplifier les BDDs pendant leur manipulation.

5.4 Discussion

La simulation représente le moteur de tout algorithme de validation basé sur des modèles. Nous avons présenté dans ce chapitre trois modes de simulation complémentaires pour des programmes IF respectivement énumérativeΓmixte et symbolique. Dans la suiteΓnous allons présenter comment la simulation s'intègre de manière effective dans la chaîne de validation à base de modèles.

Chapitre 6

Validation

L'une des principales motivations pour la définition de la représentation intermédiaire IF a été de construire un environnement *ouvert* de validationΓintégrant dans un cadre uniforme des outils de validation hétérogènes. Un tel environnement doit satisfaire les contraintes suivantes :

- premièrementΓil doit fournir *plusieurs techniques de validation* incluant la simulation interactiveΓla vérification automatique de propriétésΓle calcul d'abstractionsΓla génération automatique de séquences de testΓla génération automatique du codeΓetc. La solution est l'intégration de plusieurs outils au sein de l'environnementΓcar normalement les outils existants proposent seulement une partie de techniques mentionnées.
- deuxièmementΓl'environnement doit fournir *plusieurs représentations* possibles à la fois pour les programmes et pour leurs modèles sémantiques. GénéralementΓil est bien connu que la validation formelle des protocoles nécessite habituellement la combinaison de plusieurs techniques d'analyse avec des représentations associéesΓafin de pouvoir contourner le problème d'explosion d'état.

  ConcrètementΓnous disposons des techniques basées sur la représentation syntaxique des programmesΓcomme par exemple l'analyse statique ou le calcul d'abstractions. Des techniquesΓcomme par exemple la vérification à la volée ou la comparaison modulo une relation de simulation ou bisimulationΓtravaillent au niveau de la représentation du modèle sémantique de programmes i.eΓles systèmes de transitions étiquetées associés. Là aussiΓplusieurs représentations sont envisageablesΓénumératives et symboliques.
- troisièmementΓl'environnement doit être *ouvert* et *évolutif*. AinsiΓl'intégration des outils doit se faire en utilisant à la fois de formats d'échange des données et des interfaces de programmation (API) bien définies.

Par la suite de ce chapitre nous présentons d'abord les principes des techniques de validation basées sur les modèles et quelques outils qui implémentent ces techniques. Ensuite nous proposons une architecture pour intégrer ces techniques et outils dans un environnement de validation ouvertΓconstruit autour de la représentation intermédiaire IF.

6.1 Techniques

6.1.1 Simulation

La simulation est une technique élémentaire de validation. Elle consiste à explorer le modèle sémantique du programmeΓde manière interactive ou aléatoireΓen utilisant éventuellement des heuristiques plus ou moins fines pour choisir les états à visiter. Dans ce cadreΓla mise en défaut de certaines propriétés comme par exemple la présence d'un blocage met en évidence une erreur dans le programme.

La simulation est une technique de validation *partielle*. Si aucune erreur n'est détectéeΓcette méthode permet d'augmenter la confiance dans la correction du programme mais ne permet jamais d'affirmer que le programme satisfait ses spécifications.

6.1.2 Vérification comportementale

Les propriétés comportementales expriment le comportement attendu du système à un certain niveau d'abstraction. Une propriété comportementale peut être modélisée par un système de transitions étiquetées. La vérification comportementale consiste alors à comparer deux systèmes de transitions étiquetées : le modèle du programme et le modèle de la propriété.

Cette comparaison peut être formalisée soit par une relation de préordre soit par un relation d'équivalence : la première notion définit l'inclusion sémantique entre les ensembles de comportements alors que la seconde définit l'égalité. Les relations de préordre et d'équivalence utilisés sont basées respectivement sur les notions de *simulation* [Mil80] et *bisimulation* [Par81]. Comme exempleΓnous avons la simulation forte [Mil80] et le préordre de sûreté [BFG$^+$91] respectivement la bisimulation forte [Par81]Γla bisimulation observationnelle [Mil80]Γla bisimulation de branchement [vGW89]Γla bisimulation de sûreté [Rod88]Γ la bisimulation de délai [NMV90]. Une classification de ces différentes relations peut être trouvé dans [vG90ΓMou92].

Pour les relations de préordre et équivalence mentionnées ci-dessus il existeΓdans le cas de systèmes finisΓdes algorithmes de décision efficaces. On distingue d'une part *des algorithmes par analyse globale*Γqui nécessitent la construction préalable du modèle associé au programme. Nous mentionnons ici les algorithmes basés sur le raffinement de partition [PT87ΓKS90ΓGV90]. D'autre partΓon a *des algorithmes par analyse locale* où la vérification de la propriété se combine avec la construction du modèle du programme. Nous mentionnons ici les algorithmes de comparaison à la volée [FM90ΓFM91].

6.1.3 Vérification logique

Le propriétés logiques caractérisent des *propriétés globales* du systèmeΓtelles que l'absence de blocageΓl'exclusion mutuelle ou l'équité. Parmi les formalismes utilisésΓles *logiques temporelles* s'avèrent être bien adaptéesΓcar elles permettent de décrire globalement l'évolution du système dans le temps. Dans ce casΓla vérification consiste à *évaluer* la validité de l'ensemble de ces formules sur le système de transitions étiquetéesΓmodélisant le programme.

Nous distinguons les *logiques linéaires* qui expriment des propriétés sur les séquences d'exécution et les logiques arborescentes. qui expriment des propriétés sur les arbres d'exécution issus du programme. Comme exemples de logiques linéaires nous mentionnons PTL [MP92] et LTL [Lam80]. Comme exemples de logiques arborescentes nous mentionnons CTL [CES83]Γ ACTL [NV90] et le μ-calcul arborescent [Koz83]. Une classification de différentes logiques peut être trouvé dans [Mat98].

Pour décider la satisfaction d'une formuleΓdes techniques complètement automatisables ont été proposées et implémentées depuis longtemps dans le cadre des systèmes finis. Nous distinguons aussi des techniques d'évaluation globale [EL86ΓVL92ΓCS93] ou locales [CVWY90Γ JJ91ΓVL93]Γbasés sur de parcours en avant ou en arrièreΓavec des représentations énumérative ou symboliques du modèleΓetc.

6.1.4 Test de conformité

Le test de conformité est une technique complémentaire à la vérification formelle. Il a pour objectif de démontrer l'adéquation d'une implémentation aux spécifications de référence. Dans la pratiqueΓla conformité est testée à l'aide d'une suite de tests. Chaque test consiste en un procédure finie d'interactions entre le testeur et l'implémentation à tester et doit aboutir à un verdict. La norme ISO 9646 [ISO92] propose trois sortes de verdict : *pass* (réussi)Γ*fail* (raté) et *inconclusive* (non concluant). Une implémentation est non-conforme à ses spécifications s'il existe un test dont l'exécution conduit au verdict *Fail*. Des travaux de recherche actuels concernent notamment les fondements théoriques du test [Bri88ΓTre92ΓPha94] ainsi que la génération automatique de suite de tests [FJJV97ΓJM99].

6.2 Outils

6.2.1 GEODE

GEODE [Ver] est un outil industriel de conception autour de SDL réalisé par la société VERILOG.

GEODE comporte *des éditeurs* performants pour plusieurs formalismes et notamment SDL pour la description du contrôleΓOMT [RBP$^+$91] pour la description des données et MSC [IT94e]

pour la description de propriétés. Il comporte *un simulateur* interactif permettant l'exécution pas à pas de programmes SDL. A tout momentΓl'utilisateur peut choisir et exécuter une transitionΓrevenir en arrièreΓconsulter les valeurs de variables ou les contenus des filesΓetc.

GEODE comporte de *modules de vérification formelle* et *de génération de séquences de test.* Le principe de vérification de GEODE est le model-checking énumératif avec *des observateurs* [ALH95]. IntuitivementΓles observateurs sont des automates d'état fini décrivant la propriété que l'on veut vérifier. Ils s'exécutent en parallèle et *regardent* l'exécution du programme SDL. Avec GEODEΓles observateurs peuvent être décrits soit directementΓsoit générés à partir de MSC. Le générateur de séquences de test est le résultat de l'intégration de plusieurs techniques [KJG99]. Il permet d'une part la définition automatique des objectifs de test afin de assurer la couverture de la spécification. A partir de ces objectifsΓ il permet la construction automatique des séquences de testΓrespectivement leur mise en forme TTCN [ISO92].

GEODE comporte aussi *un générateur de code* exécutable à partir de spécifications SDL. Il permet la construction rapide de prototypesΓpour différentes plate-formesΓavec une intervention minimale de la part de l'utilisateur.

D'autre partΓGEODE offre *des interfaces de programmation* (API). Une première interfaceΓ SDL/APIΓest définie au niveau du compilateur de SDL. Elle offre l'accès en lecture à l'arbre abstrait construit par la compilation. Toute information contenue dans un programme SDL est ainsi accessible. Une deuxième interfaceΓSIM/APIΓest définie au niveau du simulateur de SDL. Elle offre l'accès à la représentation interne des états et aux fonctions de calcul de transitions utilisées par le simulateur.

6.2.2 INVEST

INVEST [BLO98] est un outil de calcul d'abstractions sur des systèmes infinis développé à VERIMAGΓen collaboration avec l'Université de Kiel et le laboratoire SRI de Stanford. Il constitue un excellent exemple d'intégration des techniques de vérification algorithmiquesΓ basées sur les modèlesΓavec des techniques de vérification déductivesΓbasées sur des preuves.

INVEST est réalisé à base du démonstrateur automatique PVS [ORSvH95]. Il prend en entrée des systèmes décrits par des compositions parallèlesΓsynchrones ou asynchronesΓd'automates étendus avec des variables. Aucune restriction n'est faite sur les variables : elles peuvent avoir n'importe quel type abstrait algébriqueΓêtre de domaine fini ou infiniΓexistant en PVS. INVEST calcule de manière automatique des systèmes abstraitsΓpar rapport à des fonctions d'abstraction fournies par l'utilisateur ou dérivées automatiquement à partir du système initial. Dans ce deuxième casΓdes heuristiques très finesΓà base d'une propriété concrète à vérifier sur le système initialΓguident la construction de la fonction d'abstraction. FinalementΓ on note que la construction du système abstrait est faite de manière compositionelle i.eΓ automate par automateΓsans passer par la construction du produit.

INVEST implémente aussi des heuristiques pour calculer des invariants structurels. Ces invariants servent à la fois pour vérifier des propriétés sur le systèmeΓainsi que pour réaliser des preuves intermédiaires pendant le calcul d'abstraction.

6.2.3 KRONOS

KRONOS [Yov97] est un outil d'aide à la conception des systèmes temps réel développé à VERIMAG. Il fournit un ensemble de techniques de vérification à intégrer dans les environnements de développement conçus pour différents catégories de systèmes temps-réel : des protocoles de communication temps réelΓde circuits asynchronesΓdes systèmes hybridesΓetc.

KRONOS prend en entrée des systèmes décrits par des compositions parallèles d'automates temporisés et des propriétés exprimées soit en logique temporelle TCTL [ACD93]Γsoit par des automates de Büchi temporisés. KRONOS implémente à la fois des techniques de *vérification* i.eΓvoir si un système satisfait ses propriétés et de *synthèse* i.eΓmodifier le système (e.gΓ trouver un sous-système) de tel sorte qu'il satisfait ses propriétés.

A part ces deux techniquesΓKRONOS permet la construction de l'ensemble des états atteignables d'un système temporisé ou le calcul du système minimal équivalent par rapport à des bisimulations d'abstraction du temps (des relations entre des états ayant le même comportement à l'exception des valeurs exactes de délais).

KRONOS utilise des techniques symboliques de représentations à base de DBMs. Il emploi aussi bien des méthodes d'exploration *en avant* que *en arrière* du modèle. FinalementΓil comporte des modules d'optimisation à base d'analyse statique sur les horloges.

6.2.4 CADP

CADP (*CAESAR-ALDEBARAN Development Package*) [FGK+96] est un environnement pour le développement des protocoles LOTOS [ISO88]. Il est développé par l'équipe VASY de l'INRIA Rhône-Alpes et le laboratoire VERIMAG.

Plusieurs logiciels composent CADP : ALDEBARANΓBCCΓCAESARΓCAESAR.ADTΓEVALUATORΓOPEN/CAESARΓXTLΓetc. Tous ces composants sont accessibles par l'intermédiaire d'une interface graphique conviviale développée dans le cadre du projet EUCALYPTUS. Nous allons décrire d'abord les fonctionnalités générales de CADP et ensuite brièvement chacune de composantes.

CADP prend en entrée des descriptions de protocoles écrites dans plusieurs formalismes. Il accepte des *spécifications formelles* de haut niveau écrites dans le langage LOTOS. CADP accepte aussi des descriptions de bas niveau i.eΓdes *systèmes de transitions étiquetées*. FinalementΓCADP accepte comme représentation intermédiaire les *systèmes de transitions étiquetées communicants* i.eΓdes systèmes de transitions étiquetées composés à l'aide des opérateurs de composition parallèle avec synchronisation.

Les fonctionnalités offertes par CADP incluent la simulation interactive ou aléatoireΓla vérification comportementale par rapport à des relations de simulation et bisimulationΓla vérification logique de propriétés exprimées en logique temporelles. Toutes les outils de vérification de CADP sont basés sur le même principe d'exploration du *modèle* i.eΓle système de transitions étiquetées représentant les comportements du protocole. Ce modèle peut être accédé à travers plusieurs représentations. La *représentation énumérative explicite* consiste

dans la liste d'états et de transitions. La *représentation énumérative implicite* consiste dans un ensemble des structures et de fonctions C permettant son exploration dynamique. Cette représentation est bien adaptée pour la vérification à la voléeΓnécessitant uniquement l'exploration d'une partie du modèle. FinalementΓla *représentation symbolique* consiste dans un ensemble de diagrammes de décision binaires codant les transitions du modèle. Cette représentation est normalement construite directement à partir du programme de haut niveau sans passer par la construction explicite du modèle.

CAESAR [Gar89a] et CAESAR.ADT [Gar89b] sont des compilateurs permettant la traduction des spécifications LOTOS vers des systèmes de transitions étiquetées. OPEN/CAESAR [Gar98] est une plate-forme qui permet l'implémentation des algorithmes de vérification à la volée. Il comporte des modules dédiées à la représentationΓle stockage et l'exploration du modèle. ALDEBARAN [Fer88] permet de minimiser ou de comparer des systèmes de transitions étiquetées modulo des relations d'équivalence et de préordre : la bisimulation forteΓla bisimulation observationnelleΓ la bisimulation de délaiΓ la bisimulation de de branchementΓ l'équivalence et le préordre de sûreté. Il implémente à la fois des algorithmes globaux basés sur le raffinement de partitions et des algorithmes locaux basés sur la comparaison à la volée. Dans les deux casΓil travaille aussi bien avec des représentations énumératives que avec des représentations symboliques du modèle. EVALUATOR est un model-checker à la volée de formules de μ-calcul arborescent sans alternation [Koz83]. XTL [Mat98] est un langage fonctionnel conçu pour faciliter l'implémentation des divers opérateurs de logique temporelleΓ évalués sur des modèles construits explicitement.

6.2.5 TGV

TGV (*Test Generation using Verification technology*) [FJJV97] est un prototype de génération automatique de tests de conformité développé en commun par l'équipe PAMPA de l'IRISA et le laboratoire VERIMAG. Ses algorithmes sont des adaptations d'algorithmes issus du domaine de la vérification.

TGV utilise des objectifs de test formalisés par des automates et sélectionne un test par objectif parmi les comportements possibles de la spécification. La sélection se fait par un parcours du produit entre l'objectif de test et le comportement visible de la spécificationΓ ses interactionsΓvia des points de contrôle et d'observation. TGV fournit en sortie un cas de test dans un format automate d'ALDEBARAN qui peut être ensuite traduit dans le langage TTCN.

TGV travaille complètement à la volée sur le modèle de la spécification. Il est indépendant du langage de spécification : actuellementΓil est utilisable à la fois sur des spécifications LOTOS par une connexion à l'environnement CADPΓet sur de spécifications SDL par une connexion à l'interface SIM/API du simulateur GEODE.

6.3 Environnement

Nous présentons ci-après un environnement de validation pour SDL construit par l'intégration des outils présentés auparavantΓà base du modèle intermédiaire IF. ConjointementΓces outils couvrent la plupart de techniques utilisées pour la validation de protocoles et notamment la simulation interactive ou aléatoire (GEODEΓCADP)Γl'analyse statique et le calcul d'abstractions (INVEST)Γ la vérification de propriétés de logique temporelle (KRONOSΓ CADP)Γla vérification comportementale à base de simulations ou bisimulations (CADP)Γla génération de séquences de test (GEODEΓTGV).

L'architecture globale de l'environnement est présentée dans la figure 6.1. L'environnement est centré autour de la représentation intermédiaire IF. Dans ce contexteΓIF assure le lien entre des formalismes de description de haut niveauΓ notamment des spécifications SDLΓ et des modèles sémantiques de bas niveauxΓ notamment les automates temporisés AT et systèmes de transitions étiquetées STE.

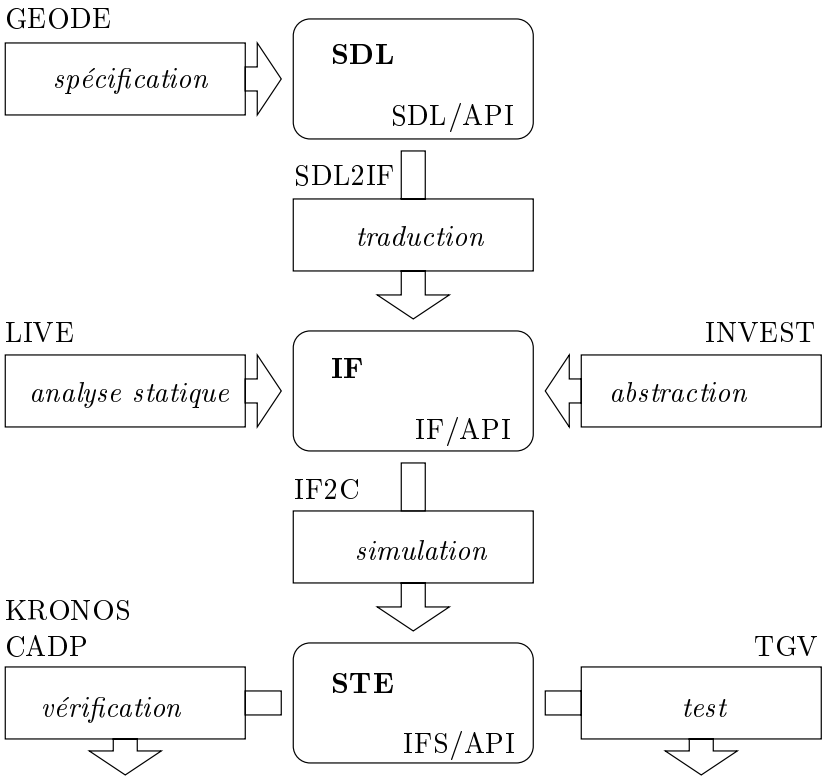

FIG. 6.1 – *Environnement de validation.*

6.3.1 SDL2IF

Le traducteur SDL2IF réalise la traduction des programmes SDL vers des programmes IF. Il implémente les principes présentés dans le chapitre 3. La version actuelle de SDL2IF

a été réalisée à l'aide de l'interface SDL/API existante au niveau du compilateur SDL de GEODE. De cette manièreΓnous avons évité le développement (assez laborieux) d'un nouveau compilateur pour SDL.

De plusΓcette connexion nous donne la possibilité d'utiliser GEODE en phases amont de la validation. Grâce à l'éditeur graphique et au simulateur interactifΓGEODE est un excellent outil pour la spécification et l'analyse préliminaire de protocoles.

D'un autre point de vueΓSDL2IF représente une deuxième connexion de GEODE aux outils de VERIMAG. La première a été réalisée au niveau de simulateur de GEODE et connecte les outils de validation énumérative de CADP [KRL97].

6.3.2 IF/API

IF/API est une bibliothèque pour la représentation et la manipulation syntaxique de programmes IF. Elle donne accès aux objets contenus dans l'arbre syntaxique construit par la compilation d'un programme IF et offre quelques primitives générales de manipulation.

Ce niveau de représentation sert pour connecter ou développer des outils d'analyse statique et d'abstractionΓétant donnée que toute information sur les horlogesΓfilesΓvariables est gardée explicitement. Parmi les outils entièrement développés avec IF/API figurent l'analyseur statique LIVE et le compilateur IF2C présentés ci-après. Des connexions ont été réalisées avec les outils INVEST et SPIN.

Nous avons défini et implémenté une traduction de IF vers le langage d'entrée d'INVEST et vice-versa [Saa99]Γavec la préservation complète de la sémantique entre ces deux formalismes. Cette connexion nous permet d'appliquerΓsur des programmes IFΓtoute technique d'abstraction et de calcul d'invariants existante actuellement dans INVEST.

IF/API a été utilisée aussi par une équipe de l'Université d'EindhovenΓdans le cadre du projet VIRESΓpour définir un traducteur de IF vers PROMELAΓle langage natif de l'outil SPIN [Hol91].

6.3.3 LIVE

L'outil LIVE implémente l'analyse de variables actives et les optimisations syntaxiques induites sur les programmes IF (voir section 4.1). Il comporte aussi une composante pour réaliser l'abstraction de variables (voir section 4.3). IntuitivementΓ il prend en entrée un programme IF et fournit en sortie un autre oùΓ toute redondance produite par les variables inactives ou inutiles est éliminée explicitement par l'introduction systematique de re-initialisations.

6.3.4 IF2C

IF2C est un compilateur qui produit les primitives de simulation en CΓpour des programmes IFΓtelles qu'elles ont été définies au chapitre 5. Il a été réalisé à l'aide de l'interface IF/API.

La représentation énumérative consiste dans un ensemble de structures de données et des fonctions C permettant l'exploration dynamique du modèle. Notamment on distingue la structure représentant l'état du modèle et la fonction primitive de calcul de successeurs $Post^e$.

De la même manière la représentation mixte consiste dans la structure C concrète des états symboliques et la fonction de calcul de successeurs adaptée $Post^m$. Pour les états symboliques nous utilisons une bibliothèque de représentation de zones par de DBMs de taille variable développée dans KRONOS.

Finalement la représentation symbolique consiste dans un ensemble de diagrammes de décision binaires codant la relation de transition du modèle. Ces diagrammes sont construites automatiquement à partir du programme avec la prise en compte d'un certaine nombre d'options. Elles sont manipulées via l'interface SMI [Boz96] qui permet l'utilisation de manière transparente des plusieurs implémentations concrètes de BDD dont les plus performantes sont les CUDD [Som] et les TiGeR BDDs [Cor94].

6.3.5 IFS/API

Finalement IFS/API est une bibliothèque pour la représentation et la manipulation de systèmes de transitions étiquetées issus de programmes IF. Elle assure d'une part l'interface avec le code ou les diagrammes de décision engendrés par le compilateur IF2C et d'autre part elle comporte certains modules auxiliaires basiques de manipulation et de stockage pour des états et des transitions.

Cette bibliothèque sert pour connecter ou développer des algorithmes de validation qui travaillent habituellement au niveau sémantique.

Par exemple nous avons connecté le simulateur mixte de IF aux algorithmes de validation à la volée de KRONOS [Tri98]. Cette connexion nous a donné à la fois l'opportunité de valider la technologie de KRONOS sur des exemples plus complexes et aussi de récupérer cette technologie pour des programmes IF et SDL.

Nous avons connecté le simulateur énumératif aux outils CADP travaillant de manière énumérative. Une première application a été la génération explicite du modèle système de transitions étiquetées associés aux programmes IF dans un format automate accepté par CADP. Par la suite tous les outils de vérification de CADP sont applicables sur les modèles ainsi générés. Une deuxième application a été l'adaptation de l'outil EVALUATOR de tel sorte qu'il travail directement à la volée sur des programmes IF.

D'autre part nous avons connecté le simulateur symbolique aux algorithmes symboliques de minimisation et comparaison existants dans ALDEBARAN [FKM93]. Aussi nous avons implémenté une version symbolique de l'outil EVALUATOR. L'évaluation est faite de manière classique en arrière sur le modèle pour des formules de μ-calcul générales.

Finalement un travail de connexion avec l'outil TGV est en cours.

Troisième partie

Applications

Chapitre 7

Applications

Dans ce chapitre nous présentons quelques applications concrètes de IF pour la validation de protocoles. Nous avons choisi trois exemples avec des fonctionnalités et complexités différentes : un protocole classique d'exclusion mutuelle basé sur la circulation d'un jeton sur un anneauΓun protocole de transfert de donnée de la pile ATM et un circuit asynchroneΓaussi de transfert de données.

Suite au travail sur ces protocolesΓnous avons dégagé quelques principes méthodologiques généraux à suivre pour aboutir dans la validation.

- Dans une première phase il est absolument nécessaire de comprendre le fonctionnement du protocole et ses propriétés. D'une part on peut procéder à des simulations interactives pour vérifier nos intuitions. D'autre partΓon doit construire et examiner des modèles plus abstraits du fonctionnementΓobtenus en utilisant des techniques plus ou moins fortes d'abstraction.

- Dans une deuxième phase on propose l'utilisation intensive de techniques d'analyse statique sur les programmes. Généralement on pense à l'application de toute technique syntaxique permettant la transformation du programme afin de réduire le vecteur d'étatΓou l'espace d'états de son modèleΓtout en préservant les propriétés à vérifier. A part un gain supplémentaire que ces analyses peuvent encore nous apporter sur la compréhension du protocoleΓles résultats fournis s'avèrent d'une importance capitale pour le succès des phases ultérieures de validation.

- EnsuiteΓon devrait choisir le mode de simulation le mieux adapté au protocole. Des structures de données complexesΓou encore un fonctionnement *peu régulier* du protocole limite généralement l'utilisation d'une simulation symbolique à base de BDDs. D'autre partΓun nombre élevé d'horlogesΓgénéralement activesΓlimite l'utilisation de la simulation mixte à base de régions.

- FinalementΓon devrait trouver le bon formalisme pour spécifier les propriétésΓet ensuite les outils les mieux adaptés pour leur vérification.

Pour chacun des exemples mentionnésΓaprès une courte présentation du fonctionnement et des propriétésΓnous décrivons les techniques d'analyse statique et de validation appliquéesΓ et les résultats obtenus.

7.1 Le protocole TRP

7.1.1 Présentation

Le protocole TRP (*Token Ring Protocol*) correspond à un algorithme classique permettant à un réseaux de processus de partager une ressource critique. Cet algorithmeΓproposé dans [Lan77] et amélioré par [CR79] repose sur une connexion de processus selon une structure d'anneau. Une marque spécialeΓappelée jeton (*token*) circule sur cet anneau ; le processus ayant en sa possession le jeton a le droit d'accéder à la ressource critique. Si le jeton est unique sur l'anneau alors l'exclusion mutuelle est assurée.

Une caractéristique importante de cet algorithme est sa résistance aux pannes. Il permet de traiter plusieurs types de pannes qui se caractérisent toutes par la perte du jetonΓcomme par exemple les pannes de transmission (si le jeton est perdu lors d'une transmission) ou panne de stations (si une station tombe en panne et si elle avait le jeton). Si le jeton est perduΓil faut le régénérer. Le problème est alors de choisir un et un seul processus parmi les processus encore actifs qui se chargera de cette régénération. Ce choix est fait par un algorithme d'élection.

7.1.2 Modélisation

Nous avons considéré une description de ce protocole en SDLΓavec $n = 4$ stations. IntuitivementΓil s'agit d'un seul bloc contenant l'anneauΓconstitué de processusΓcomme le montre la figure 7.1. Deux signaux circulent sur l'anneau : *token*Γdont le rôle a été précisé ci-dessus et *claim* utilisé dans l'algorithme d'élection.

Toutes les stations ont le même comportementΓprésenté dans la figure 7.2. La temporisation *worried* est armée quand la station commence attendre le jeton. Si elle expireΓla station suppose que le jeton est perdu et lance une élection en envoyant un signal de candidature *claim*Γparamétré par son numéro d'identification et le bit *round*. Le bit alterné *round* est utilisé pour distinguer entre les candidatures valides (émises pendant la phase courante d'élection) et invalides (annulées par le passage du jeton). BrefΓà l'état *idle*Γune station peut recevoir soit le jeton et ensuite accéder à la ressource critique (l'état *critical*)Γsoit le signal d'expiration de *worried* au cas où elle envoie sa candidature et lance une nouvelle électionΓ soit une candidature en provenance de son voisin. Dans ce dernier casΓcette candidature est *filtrée* (non-transmise) si le numéro d'identification est plus petitΓou transmise inchangée s'il est plus grand que le sien. Si une de ses propres candidatures est reçue la station devient élue et régénère le jeton.

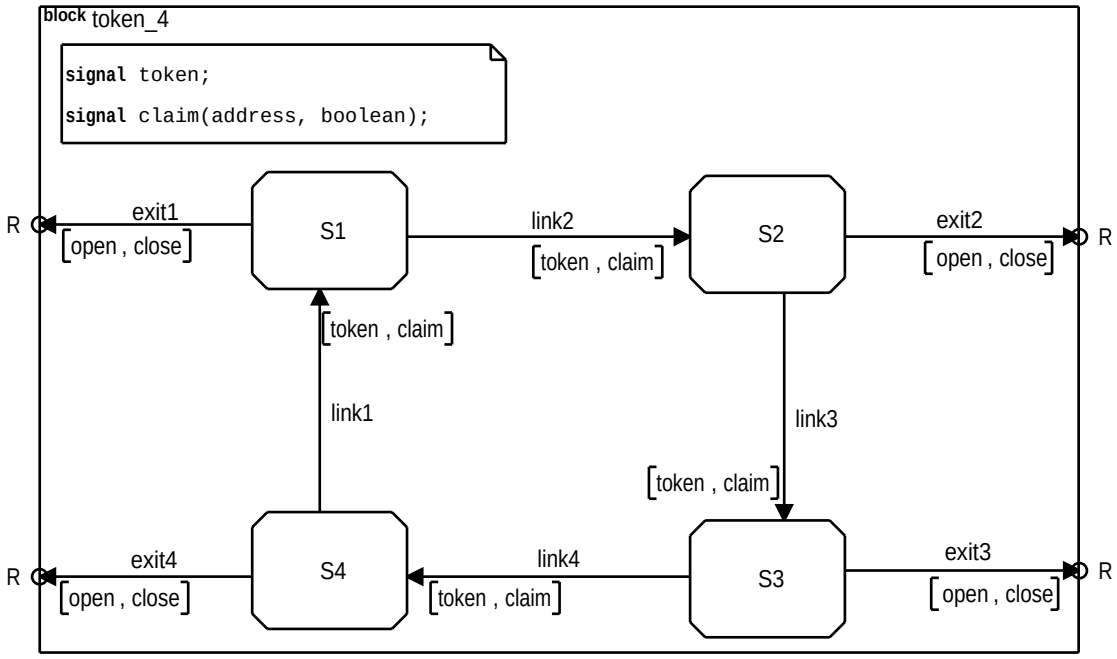

FIG. 7.1 – *La structure du protocole TRP en SDL.*

7.1.3 Validation

Génération de modèles

Nous avons considéré les cas suivants :

- *a) files fiables, urgence maximale.* Ce cas correspond à la sémantique habituelle de SDL où les files sont fiables et le temps avance quand le système est en attente (sans transitions discrètes franchissables). Ce cas est relativement restrictif car il comporte juste une phase d'élection (une fois le jeton généré il n'est sera plus jamais perdu) et ensuite il fonctionne correctement.

- *b) files non-fiables, urgence maximale.* Les files non-fiables entraînent la perte du jeton et des candidatures. Plusieurs élections sont possibles maisΓà cause de l'urgence maximale elles seront déclenchées uniquement quand le système est en attente. Il s'agit donc toujours que de *vraies* électionsΓlancées quand le jeton est perdu et jamais des *fausses* électionsΓlancées pendant que le jeton existe encore sur l'anneau.

- *c) files non-fiables, urgence affaiblie.* Nous avons affaibli l'urgence de la transition sortante de l'état critiqueΓen le considérant retardable d'une unité de temps. Ça permet au processus de passer un temps limité dans cet état. Si c'est le casΓpendant ce temps les autres peuvent lancer des électionsΓainsi plusieurs candidatures circulent sur l'anneauΓ

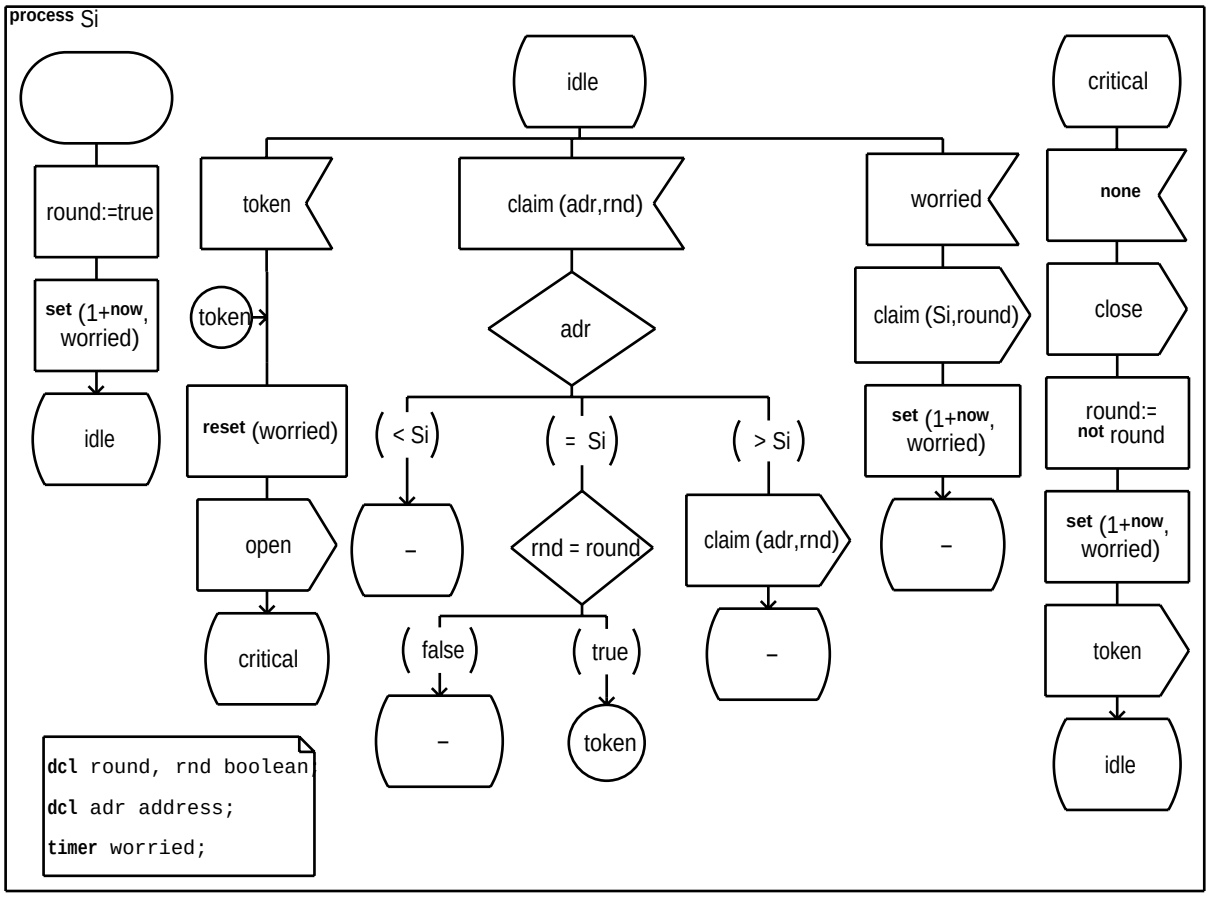

FIG. 7.2 – *Le comportement d'une station en SDL.*

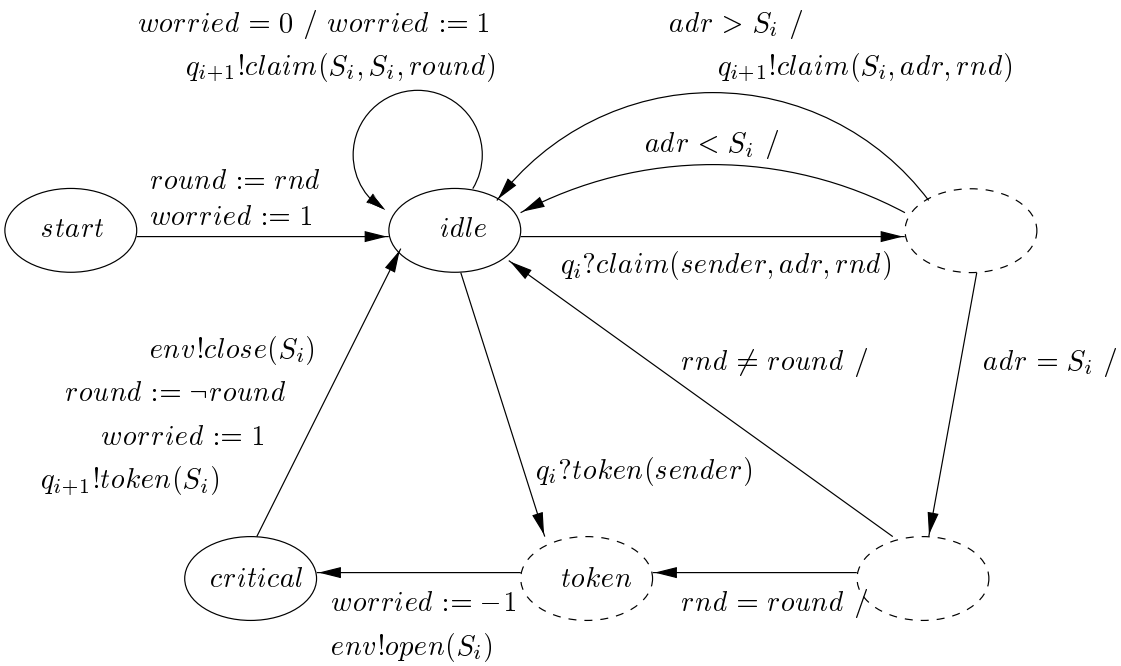

FIG. 7.3 – *Le comportement d'une station en IF.*

sans que le jeton soit vraiment perdu.

Cette solution n'est pas unique : on peut aussi bien considérer des retards différents au niveau de chaque station ou encore d'autres transitions retardables voir même paresseuses. Nous avons choisi celle-ci car à part le fait qu'elle autorise plus de comportements intéressants elle est encore assez forte pour garantir un modèle fini du protocole. Par exemple pouvoir passer un temps illimité à l'état critique ce qui correspondrait à mettre la transition de sortie paresseuse peut conduire au remplissage illimité des files. Toutes les candidatures issues par les autres stations pendant chaque unité de temps vont s'accumuler dans la file d'un processus toujours à l'état critique.

Pour chacun de cas nous avons considéré les deux situations extrêmes de numérotation de stations :

- *1)* les stations sont numérotées dans *l'ordre directe* de circulation de signaux sur l'anneau ; ça correspond au cas le plus favorable de l'algorithme de l'élection *linéaire* en nombre de signaux transmis par rapport au nombre de stations ;
- *2)* les stations sont numérotées dans *l'ordre inverse* à la circulation de signaux sur l'anneau ; ça correspond au cas le plus défavorable de l'algorithme d'élection cette fois-ci *quadratique* en nombre de signaux transmis par rapport au nombre de stations.

Nous avons généré les systèmes de transitions étiquetées associés à ces modèles en utilisant le simulateur GEODE et le simulateur IF sans et avec la prise en compte des résultats

		GEODE	IF	IF + AS
a	*1*	1731 / 3822	618 / 1256	292 / 756
		1.0	0.4	0.2
	2	3611 / 8092	1412 / 3060	934 / 2261
		1.0	0.9	0.6
b	*1*	3018145 / 7119043	537891 / 2298348	4943 / 19664
		18:07.00	9:07.23	4.8
	2	- / -	1670209 / 7738144	17295 / 66496
		-	29:54.46	15.2
c	*1*	- / -	- / -	54591 / 250016
		-	-	54.9
	2	- / -	- / -	1110431 / 4495904
		-	-	21:31.3

TAB. 7.1 – *Résultats obtenus pour le protocole TRP.*

d'analyse statique. Dans ce dernier cas nous avons considéré notamment l'optimisation due aux variables inactivesΓcalculées au niveau de chaque station. ConcrètementΓil s'agit principalement des variables *rnd* et *adr*Γjamais actives dans les états stables mais qui prennent plusieurs valeurs à l'exécution. Les résultats obtenus sont présentés dans la table 7.1. Pour chaque cas nous donnons la taille du système de transitions étiquetéesΓle nombre d'états et de transitions (n/m) ainsi que le temps[1] utilisateur nécessaire pour les générer.

Vérification logique

La propriété d'exclusion mutuelle a été vérifiée avec EVALUATORΓpour toutes les situations considérées:

† *pour chaque station* S_i*, tout accès à la ressource* $open(S_i)$ *ne peut être suivi d'aucune action observable autre que la sortie de la ressource* $close(S_i)$ *:*

$$\textbf{all}\ [open(S_i)]\ \textbf{not pot}_{\tau}\ \langle\overline{close(S_i)}\rangle\ tt$$

Vérification comportementale

Tous les graphes générés sont bisimilaires avec l'un des graphes présentés dans la figure 7.4. La comparaison a été faite modulo la *bisimulation de sûreté* [Rod88]Γen gardant visibles uniquement les actions qui marquent l'accès à la ressource critique. La comparaison a été faite à l'aide d'ALDEBARAN.

1. les tests ont été effectués sur un Sun Ultra SPARC 10 avec 128 MB

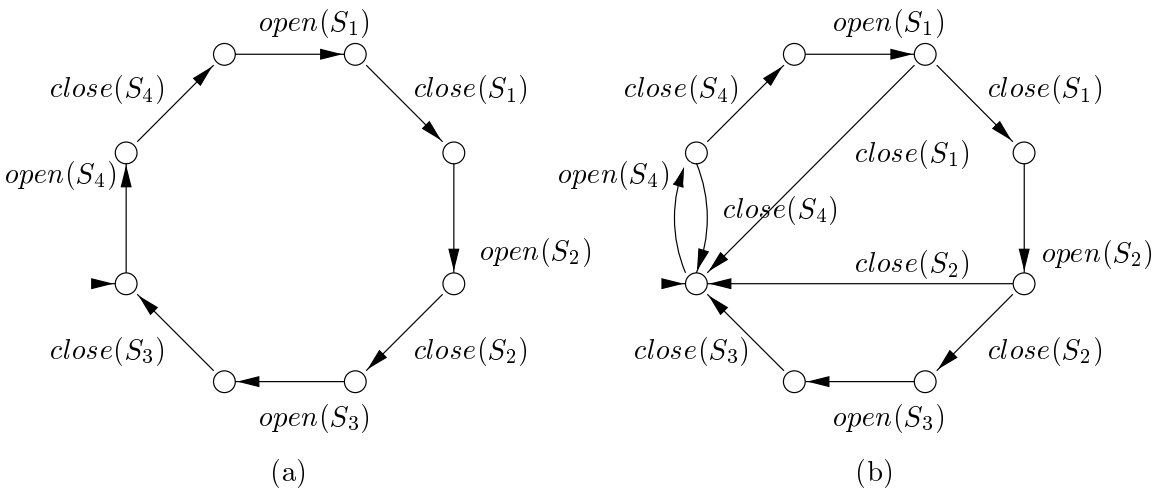

FIG. 7.4 – *Graphes réduites pour (a) files fiables et (b) files pas fiables.*

Chaque graphe réduit modélise la propriété d'exclusion mutuelle. MaisΓsur cet exempleΓon voit qu'il faut différents graphes pour exprimer cette propriétéΓen fonction du type de filesΓ alors qu'une seule formule de μ-calcul suffit.

7.1.4 Observations

Une synthèse sur la vérification algorithmique de différentes versions de TRP peut être trouvée dans [GM96]. Des travaux plus récents ont porté sur la vérification des versions paramétrés par le nombre de processusΓen utilisant des méthodes déductives [BLM97].

7.2 Le protocole SSCOP

7.2.1 Présentation

Le protocole SSCOP (*Service Specific Connection Oriented Protocol*) fait partie des protocoles de la pile ATM (*Asynchronous Transfer Mode*). SSCOP est normalisé sous la référence ITU-T Q2110 [IT94c]. A l'origine il est conçu pour transférer des données à haut débit entre deux entités large bande et ce d'une façon sûre. Bien que sa conception le rende apte à traiter des volumes de données importantsΓactuellement son usage est cantonné dans la couche signalisation de l'ATM. CependantΓil est raisonnable de penser qu'il sera employé pour le transfert de données importantes dans des applications futures. SSCOP est une des sous-couches de la couche AAL (*ATM Adaptation Layer*) qui a pour rôle essentiel d'adapter le service fourni par la couche ATM physique au type des données passant par la connexion établie entre deux extrémités.

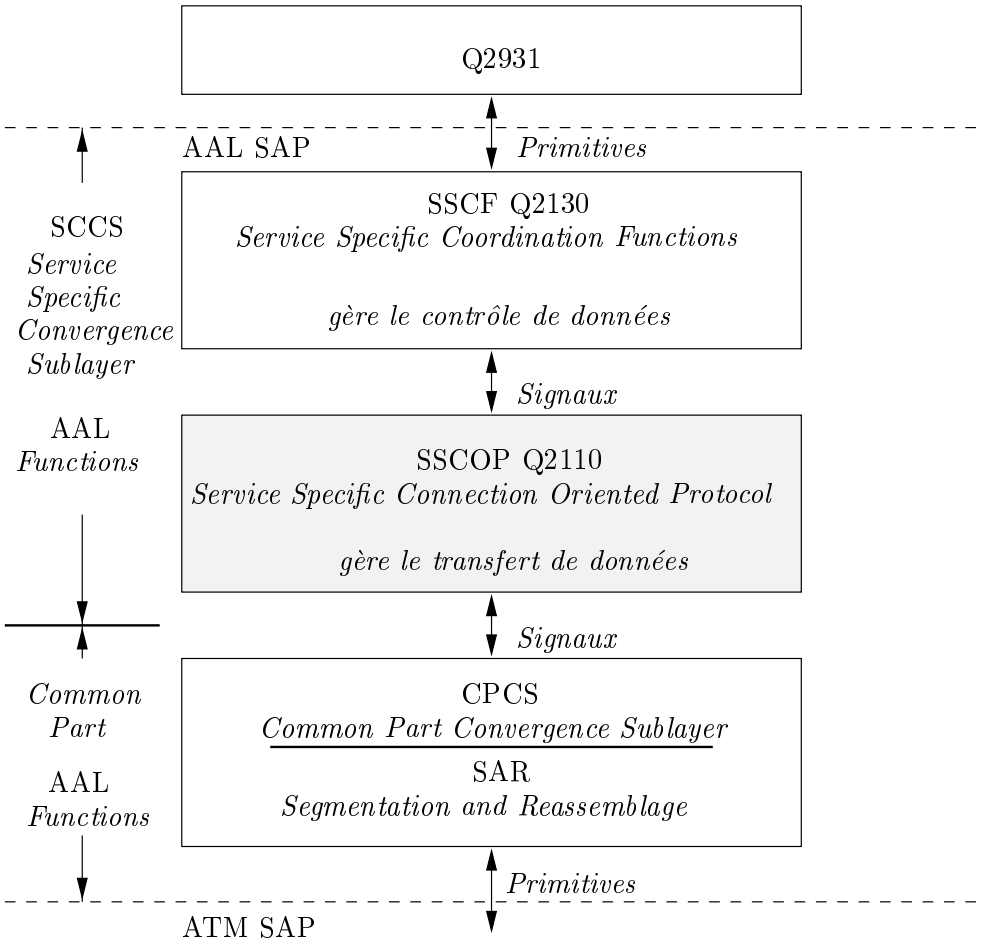

FIG. 7.5 – *Placement du SSCOP dans la pile ATM.*

Cette étude de cas a été fournie par le CNET [2] Lannion dans le cadre de l'action FORMA [3]. Nous disposons de la norme ITU-T décrivant le protocole et sa spécification en SDL ainsi qu'une spécification SDL sous forme électronique écrite par le CNET à partir de la norme.

La vérification de la spécification SDL fournie permet de s'assurer de sa validité du point de vue de propriétés d'accessibilité (absence de deadlockΓde livelockΓd'ambiguïtés)Γde sa validité par rapport à la norme ITU-T (la spécification doit décrire correctement un sous-ensemble des fonctionnalités de la norme).

7.2.2 Modélisation

En quelques motsΓSSCOP est un protocole de transfert de données à base d'une fenêtre glissante (*sliding window*). Sa description en SDL comporte un seul processus qui mixe les fonctionnalités en tant que émetteur ou récepteur du protocole. Ce processus comporte uniquement 10 états de contrôle mais il utilise un nombre important de variablesΓdont une grande partie structurées. En toutΓla description du processus est d'environ 2000 lignes de SDL. Elle couvre les fonctionnalités suivantes :

- *contrôle de la connexion* incluant établissementΓcontrôle de flux par commande du débit par le récepteurΓmaintien (même en l'absence prolongée de transfert de données)Γ relâchementΓet re-synchronisation ;
- *transfert de données* usager en mode garanti ou nonΓavec intégrité de séquencement (les paquets soumis par l'usager au protocole sont numérotés dans l'ordre où ils seront soumis pour transfert) ;
- *détection et recouvrement d'erreur* de protocole par retransmission sélective (détection par le récepteur de paquets manquants et demande de retransmission) ;
- *indication* d'erreur ou d'état (de la fenêtre de transfert) à la couche de gestion.

7.2.3 Validation

Nous décrivons maintenant l'ensemble des résultats obtenus sur cette étude de cas. Nous commençons par les résultats d'analyse statique de la spécification qui ont permis de comprendre le protocoleΓsimplifier sa spécificationΓcorriger quelques erreurs par rapport à la norme et surtout de réduire son graphe d'états de façon très significative. Le résultat de ces travaux préliminaires est une nouvelle spécification corrigée et simplifiée qui a ensuite servi à la vérification.

2. Centre National d'Etudes de Télécommunications de France Telecom

3. FORMA est une action soutenue par le programme DSP"STTC/CNRS/MENRT: "Maîtrise de systèmes complexes réactifs et sûrs"

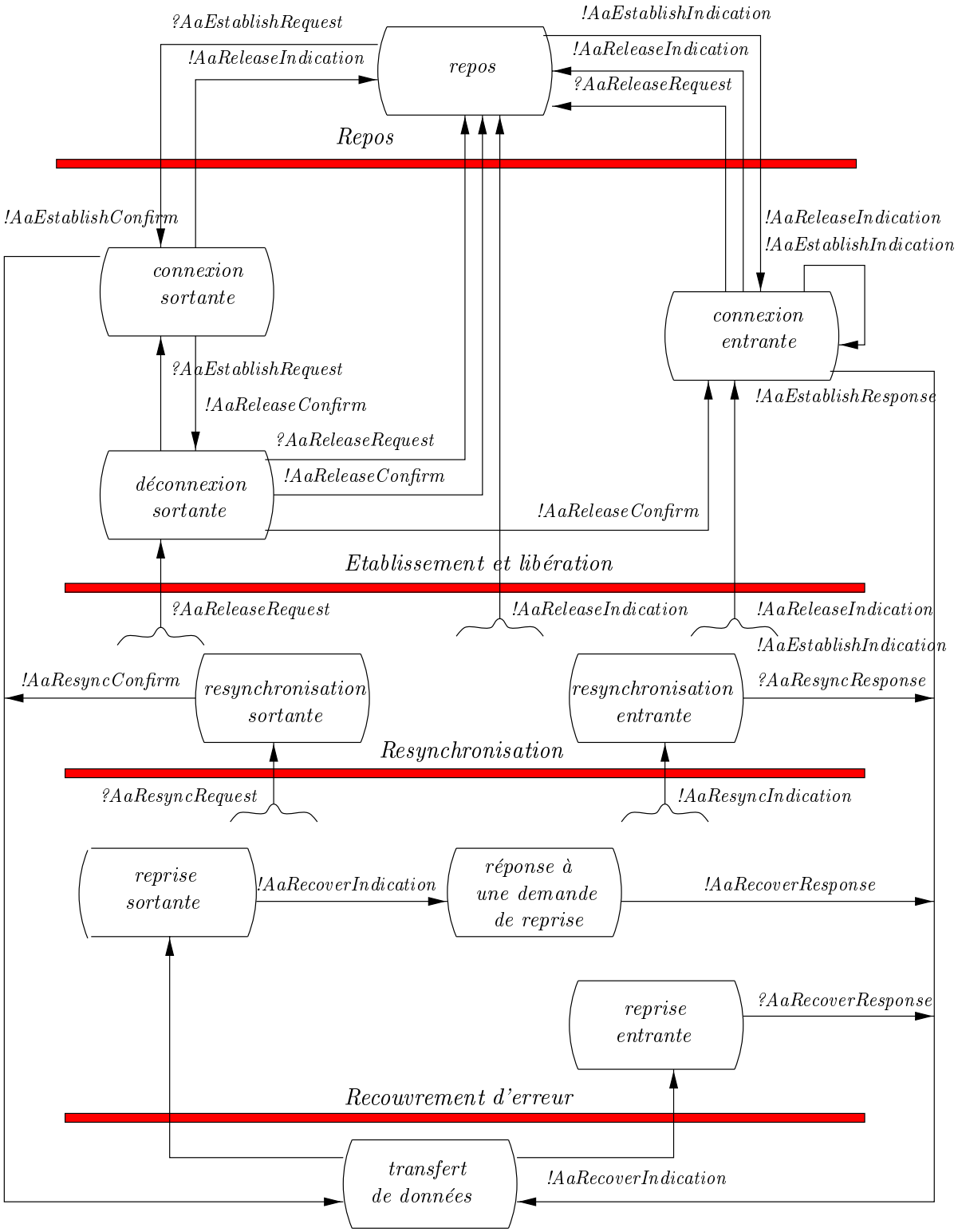

FIG. 7.6 – *Fonctionnement global du SSCOP.*

Analyse statique

L'objectif principal de cette première étape était d'effectuer un travail de simplification et d'abstraction sur la spécification SDL fournie par le CNET. En effetΓle rôle de cette spécification étant d'offrir une description détaillée du protocole (y compris dans des aspects très liés à son implémentation)Γil n'aurait pas été envisageable de l'utiliser telle quelle dans le contexte d'une vérification formelle ni même pour la génération de séquence de tests.

Un deuxième objectif était de faire un certain nombre de vérifications statiques sur cette spécificationΓde manière à détecter au moindre coût les erreurs ou les omissions flagrantes par rapport à la norme qu'elle est supposée décrire. Ces vérifications préliminairesΓmanuelles ou automatiquesΓne sauraient toutefois être exhaustives.

EnfinΓcette première analyse était également indispensable pour mieux appréhender le fonctionnement général du protocoleΓnécessaire dans la suite pour spécifier des propriétés ou interpréter des diagnostics produits par les outils de vérification et vérifier la validité des tests obtenus.

• *Abstraction de variables*

Dans une première phase nous avons fait l'abstraction de toutes les variables du protocole. On obtient ainsi un système de transitions d'environ 1000 états.

L'essentiel des vérifications menées sur ce système a consisté à le comparer avec les automates fournies dans la norme pour modéliser les interactions entre le protocole et ses couches adjacentes (notamment les séquences de signaux échangés). Ces comparaisons ont été effectuées avec l'outil ALDEBARAN.

Quelques erreurs ont ainsi été révélées dans la spécificationΓnotamment des permutations entre les états d'arrivée de différentes transitions. Ces anomalies ont pu être confirmés lors de simulations interactives ou de générations partielles de graphes d'états avec GEODE.

• *Recouvrement d'activité*

Dans une deuxième phase nous avons effectué des recouvrements des variables du protocoleΓ en nous basant sur leur activité.

D'abordΓune première transformation a consisté à supprimer les variables inutiliséesΓainsi que certains champs des structures concernant uniquement une implémentation (les champs réservés des PDUsΓles champs destinés a l'alignementΓetc.). EnsuiteΓcertaines parties de la spécification ont été réécrites pour supprimer également des variables *redondantes* (par exemple des variables utilisées localement dans un état SDL pour construire les différents structures avant leur émission). FinalementΓpour éviter de générer des états redondantsΓ nous avons appliqué la technique de ré-initialisation des variables inactives à une même valeur prédéfinie.

Les diverses simulations exhaustives menées par la suite ont montré l'intérêt de ces différentes transformations sur la taille et le nombre d'états générés. En particulier la ré-initialisation des variables inactives a donné lieu à des gains spectaculairesΓpermettant notamment de diviser par 200 la taille des graphes obtenus.

Vérification

Une fois les erreurs ou omissions évidentes détectées statiquementΓil reste alors à montrer la correction de la spécification de façon plus exhaustive. ToutefoisΓcompte-tenu de sa complexitéΓet surtout en l'absence d'un comportement de référence universel (i.e.Γvalide quelque soit l'environnement)Γil est évident que cette correction ne peut être établie que par rapport à des environnements particuliers du protocoleΓpour lesquels certaines propriétés bien précises doivent être vérifiées.

Il a donc été choisi d'effectuer cette vérification en considérant dans un premier temps deux entités du protocoleΓreliées entre elles par des files d'attentes bornées et fiables (modélisant ainsi une couche inférieure fiableΓsans perte de signaux)Γet susceptibles d'accepter différents signaux de la couche supérieure. Par suiteΓen fixant cet ensemble de signaux pour chacune des deux entitésΓil devient possible de générer les graphes d'états correspondant aux comportements de différentes phases du protocole (l'établissement de la connexionΓle transfert de donnéesΓetc.).

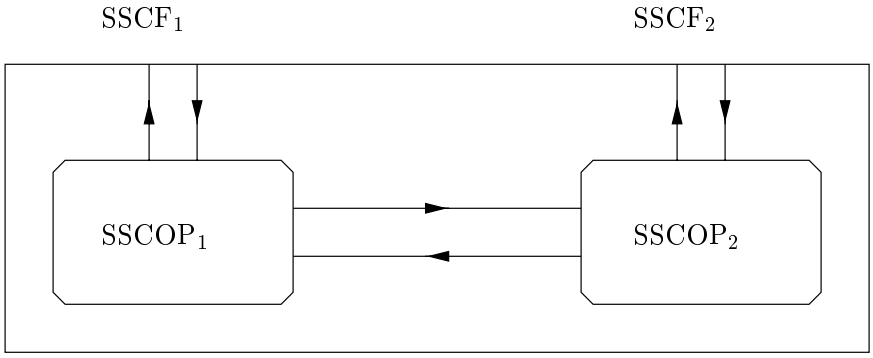

FIG. 7.7 – *Deux entités SSCOP communicantes.*

On donne dans la suite quelques uns des scénarios qui ont pu être ainsi examinés en indiquant les problèmes rencontrés.

• *Etablissement d'une connexion*

Les signaux acceptés de la couche supérieure par chacune des deux entités sont la demande d'établissement d'une connexion (*AaEstablishRequest*) et la réponse à une demande de connexion (*AaEstablishResponse*). Le graphe généré comportait alors 15000 d'étatsΓpuis 5000 après réduction. Les propriétés suivantes ont été spécifiées en logique temporelle (μ-calcul arborescent) et vérifiées avec EVALUATOR :

† *toute demande de connexion (AaEstablishRequest) d'une entité peut être suivie d'une confirmation de connexion (AaEstablishConfirm) par cette même entité :*

all [$SSCOP_1!AaEstablishRequest$] **pot** ⟨ $SSCOP_1!AaEstablishConfirm$ ⟩ tt

† *toute demande de connexion (AaEstablishRequest) d'une entité n'est pas inévitablement suivie d'une confirmation de connexion (AaEstablishConfirm) par cette*

même entité :

all [$SSCOP_1!AaEstablishRequest$] **inev** ⟨ $SSCOP_1!AaEstablishConfirm$ ⟩ tt

En examinant le diagnostic obtenu pour cette seconde propriété on constate que la connexion ne s'établit pas lorsque l'une des temporisations (*TimerCC*) expire sur l'une ou l'autre des entités (ce qui conduit à un abandon de la tentative de connexion en coursΓconforme à la norme). Cette propriété a donc pu être formulée de la manière suivante et vérifiée avec EVALUATOR : *toute demande de connexion d'une entité non suivie d'une expiration de TimerCC sur l'une ou l'autre des entités est inévitablement suivie d'une confirmation de connexion par cette même entité.*

Il en résulte qu'en l'absence de pertes de signauxΓet si les temporisations sont convenablement réglésΓalors l'établissement correct d'une connexion est garanti.

• *Libération d'une connexion*

Les signaux permettant une demande de libération de connexion (*AaReleaseRequest*) ont été ajoutés aux ensembles précédents. Le graphe généré comportait alors 30000 d'étatsΓpuis 8000 après réduction. Les propriétés vérifiées avec EVALUATOR sont les suivantes :

† *toute demande de libération de la connexion (AaReleaseRequest) d'une entité est inévitablement suivie d'une indication de libération (AaReleaseIndication) émanant de l'autre entité :*

all [$SSCOP_1!AaReleaseRequest$] **inev** ⟨ $SSCOP_2!AaReleaseIndication$ ⟩ tt

† *toute demande de libération de la connexion (AaReleaseRequest) d'une entité peut être suivie d'une confirmation (AaReleaseConfirm) de libération par cette même entité :*

all [$SSCOP_1!AaReleaseRequest$] **pot** ⟨ $SSCOP_1!AaReleaseConfirm$ ⟩ tt

† *toute demande de libération de la connexion (AaReleaseRequest) d'une entité n'est pas inévitablement suivie d'une confirmation de libération (AaReleaseConfirm) par cette même entité :*

all [$SSCOP_1!AaReleaseRequest$] **inev** ⟨ $SSCOP_1!AaReleaseConfirm$ ⟩ tt

Là encoreΓen examinant le diagnostic obtenu pour cette dernière propriété on constate qu'une demande de libération n'est pas confirmée lorsque la connexion a été rompue entre temps (soit parce que l'autre entité avait également demandée la libérationΓsoit sur expiration de la temporisation *NoResponse*).

• *Transfert de données en mode assuré*

Les signaux utilisées pour ce scénario sont la demande de l'établissement de la connexion de la part de l'entité 1 et la réponse à cette demande de la part de l'entité 2Γainsi que la demande de transfert de 2 messages de données (*AaDataRequest*) de l'entité 1 vers l'entité 2.

Le graphe généré comportait alors 4 millions d'états puis 33000 après réduction. La propriété vérifiée avec EVALUATOR est la suivante :

> † *il n'est pas inévitable qu'une demande d'émission (AaDataRequest) de message reçu par l'entité 1 de sa couche supérieure soit suivie d'une indication de réception (AaDataIndication) de ce même message par l'entité 2 : le transfert correct de message n'est pas garanti :*
>
> **all** [$SSCOP_1!AaDataRequest$] **inev** ⟨ $SSCOP_2!AaDataIndication$ ⟩ tt

Le diagnostic associé à cette dernière propriété montre ce qui semble être une anomalie dans la spécification : lorsque le nombre de message transmis par l'entité 1 à atteint la valeur du crédit initial autorisé par l'entité 2 alors l'entité 1 n'émet plus aucun message (ce qui est conforme à la norme). Par contre cette valeur de crédit n'est jamais mise à jour ce qui bloque définitivement l'émission de message de la part de l'entité 1 (jusqu'à ce que l'expiration d'une temporisation rompe la connexion et qu'une nouvelle connexion s'établisse).

L'analyse de cette série de propriétés montre bien l'intérêt de ce type de vérification que ce soit pour améliorer la compréhension du comportement du protocole ou pour détecter des anomalies dans son fonctionnement. Il resterait donc à poursuivre ce travail en examinant d'autres propriétés (concernant par exemple la re-synchronisation d'une connexion la restitution de données locales etc.) ou encore en effectuant ces vérifications en considérant un environnement plus hostile pour le protocole (avec pertes de signaux pannes de l'entité homologue etc.).

Dans cette perspective il est vraisemblable que l'approche utilisée jusqu'ici ne puisse être appliquée pour certains scénarios en raison de la taille du graphe à générer. D'autres alternatives pourront alors être utilisées comme la vérification à la volée ou l'utilisation de représentations symboliques pour modéliser le comportement du protocole.

7.2.4 Observations

Les résultats complets sur la validation de SSCOP peuvent être trouvés dans [BFG+98]. L'étude d'une spécification SDL de cette taille montre bien l'importance du travail réalisé *statiquement* en amont de la phase de vérification exhaustive en particulier pour optimiser la spécification afin de réduire la taille du graphe d'états qui la représente. En effet les propriétés qui ont été vérifiées automatiquement par la suite n'auraient certainement pas pu l'être à partir de la spécification initiale. De nombreuses perspectives restent donc à explorer dans ce sens tant pour déterminer de nouveaux critères d'optimisation que pour en automatiser certaines transformations.

7.3 Le circuit STARI

7.3.1 Présentation

STARI (*Self Timed at Receiver's Input*) [Gre93Γ Gre] est une nouvelle approche pour la communication à large bande combinant des techniques synchrones et asynchrones. Il s'agit de faire communiquer un émetteur et un récepteur qui fonctionnent de manière synchroneΓ et à la même fréquence d'horlogeΓvia une file asynchrone. Le rôle de la file est de maintenir la synchronisation entre l'émetteur et le récepteur lorsque la vitesse de l'un d'entre eux varie sur une courte période. Un protocole interne à la fileΓ à base d'acquittementsΓassure le fonctionnement sans perte ou duplication des données.

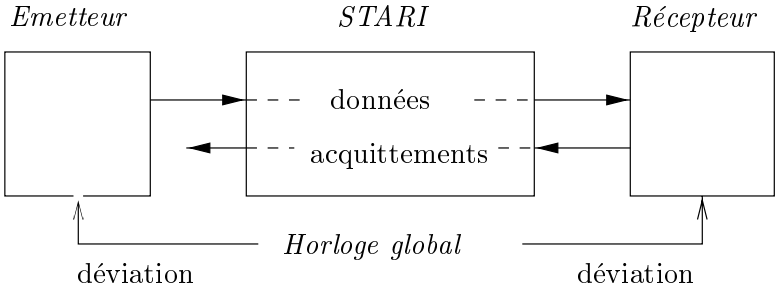

FIG. 7.8 – *Présentation du circuit STARI.*

Le fonctionnement du STARI est basé sur une idée simple. La file est initialisée à moitié pleine. EnsuiteΓpendant chaque cycle d'horlogeΓl'émetteur insère une valeur en début et le récepteur enlève une autre en fin de la file. Etant donné la nature complémentaire de ces actionsΓaucun contrôle supplémentaire n'est nécessaire pour tester si la file est vide ou pleine. En outreΓles variations de vitesse sur une courte période entre l'émetteur et le récepteur sont atténuées par la fileΓrespectivement en insérant ou effaçant plus de valeurs.

Les deux propriétés suivantes doivent être vérifiées afin de garantir le bon fonctionnement :

† *chaque valeur envoyée par l'émetteur doit être copiée dans la file avant d'envoyer la suivante*

† *une nouvelle valeur doit sortir de la file chaque fois que le récepteur envoie un acquittement.*

Dans la suite nous présentons l'implantation concrète de STARI proposée dans [TB97]. Les valeurs transmises sont les valeurs booléennes *true* et *false*Γséparées par une valeur auxiliaire *empty*. La valeur *empty* est nécessaire pour distinguer entre deux valeurs booléennes identiques et une valeur transmise pendant plus d'un cycle d'horloge. Chacune de ces valeurs x est codée à l'aide des deux booléens $x.t$Γ$x.f$Γcomme le montre la figure 7.9.

Le circuit est constitué d'une séquence de n tranches identiquesΓchacune capable de stocker une valeur x. Le principe de fonctionnement d'une tranche k est le suivant : *elle copie la*

x	*true*	*false*	*empty*
$x.t$	1	0	0
$x.f$	0	1	0

FIG. 7.9 – *Codage des valeurs (dual rail encoding).*

valeur de la tranche précédente ($x_k := x_{k-1}$) si et seulement si la tranche suivante a copié et acquitté sa valeur courante ($x_k = x_{k+1}$). Avec le codage binaire considéréΓce comportement peut être obtenu avec deux portes de Muller-CΓqui stockent les composantes $x.t$ et $x.f$ d'une valeurΓet une porte NOR qui calcule l'acquittement (voir figure 7.10).

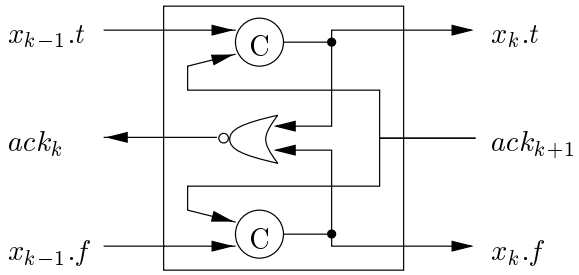

FIG. 7.10 – *Le k^{eme} tranche du circuit STARI.*

Une porte de Muller-C fonctionne de manière suivante : si ses deux entrées ont la même valeur alorsΓaprès un certain délaiΓla sortie prend aussi cette valeurΓsinon elle reste inchangée.

Par exempleΓconsidérons la situation où les tranches k et $k+1$ contiennent la valeur *empty*Γ la tranche $k-1$ la valeur *true* et l'acquittement ack_{k+1} est encore 0. Quand ack_{k+1} devient 1Γla porte de Muller-C de $x_k.f$ reste inchangée car ses entrées sont différentesΓmais celle du $x_k.t$ passe au 1Γses entrées étant toutes les deux 1. En effetΓla valeur *true* est ainsi copiée dans la tranche k.

7.3.2 Modélisation

Afin de pouvoir vérifier le bon fonctionnement du circuitΓétant données les contraintes temporelles sur les vitesses de fonctionnementΓSTARI et son environnement ont été modélisés par un réseaux de $3n+3$ automatesΓoù n est le nombre de tranches. Toute porte de Muller-C ou NOR est modélisée par un automate temporisé comme décrit dans [MP95]. Trois autres automates décrivant respectivement l'émetteurΓle récepteur et une horloge globale constituent l'environnement. Tous ces automates sont complètement asynchronesΓet communiquent à travers des variables partagées.

Conformément à [MP95]Γune porte calculant une fonction $\mathcal{F}$Γavec un délai arbitraire dans un intervalle $[l,u]$ se modélise par un automate à deux états comportant une variable x et une horloge CΓcomme le montre la figure 7.11. La variable x est la sortie courante de la porte ; C mesure le délai dans le calcul de x à partir de $\mathcal{F}$.

IntuitivementΓl'état *stable* corresponde à $x = \mathcal{F}$ i.eΓla sortie courante de la porte est bien

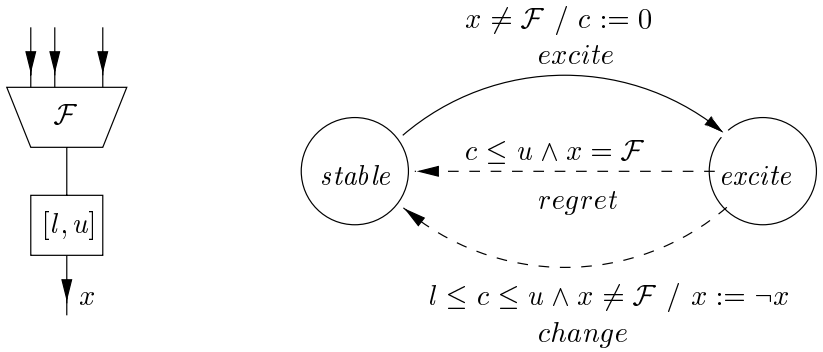

FIG. 7.11 – *Une porte générique avec délai.*

la valeur de la fonction. Dans cet étatΓun changement des entréesΓayant comme effet la modification de $\mathcal{F}$ déclenche la transition urgente vers l'état *excite*. Deux possibilités de retour sont ensuite possiblesΓles deux retardables jusqu'à u unités de temps : soit les entrées changent à nouveau tel que $\mathcal{F}$ revient à l'ancienne valeur (*regret*)Γsoit le changement persiste après l unités de tempsΓet entraîne ainsi le changement de la sortie x (*change*).

Soit k une tranche du circuitΓ$1 \leq k \leq n$. Les deux portes de Muller-C contrôlent les variables booléennes $x_k.t$ et $x_k.f$. Chacune a son horlogeΓrespectivement $c_k.t$ et $c_k.f$Γpour assurer un délai compris dans un intervalle $[l_c, u_c]$. Les fonctions calculées sont les suivantes :

$$\mathcal{F}_k.t \;=\; \begin{cases} x_{k-1}.t & \text{si } x_{k-1}.t - ack_{k+1} \\ x_k.t & \text{sinon} \end{cases}$$

$$\mathcal{F}_k.f \;=\; \begin{cases} x_{k-1}.f & \text{si } x_{k-1}.f - ack_{k+1} \\ x_k.f & \text{sinon} \end{cases}$$

La porte NOR contrôle la variable booléenne ack_k. Elle a sa propre horloge $c_k.a$Γpour assurer un délai compris dans un intervalle $[l_n, u_n]$. La fonction calculée est la suivante :

$$\mathcal{F}_k.a \;=\; \neg(x_k.t \vee x_k.f)$$

BrefΓchaque tranche est modélisée par 3 automates à deux étatsΓcomportant 3 horloges et 3 variables booléennes.

L'horloge global cyclique est modélisée par l'automate temporisé de la figure 7.12 a. Il a une horloge c qui est remise a zéro chaque fois qu'elle atteint la période p du cycle. Cette transition est annotée par l'action *tick* et est urgente.

L'émetteur est modélisé par l'automate temporisé à trois états de la figure 7.12 b. Pendant chaque cycle d'horloge il envoie une nouvelle valeur dans la fileΓce qui revient à changer les entrées de la première tranche $x_0.t$ et $x_0.f$. Le changement peut avoir lieu avec une variation

par rapport à la période exacte pΓbornée par une constante s_t. Toutes les transitions de l'émetteur sont retardables. Elles sont annotées par l'action putΓparamétrée par la valeur envoyée.

Le récepteur est modélisé par l'automate temporisé de la figure 7.12 c. Pendant chaque cycle d'horlogeΓil acquitte la sortie courante de la file en modifiant la valeur du ack_{n+1}Γavec une variation bornée par une constante s_r. Là-aussi les transitions sont retardables. Elles sont annotées par l'action getΓparamétrée par la valeur reçue.

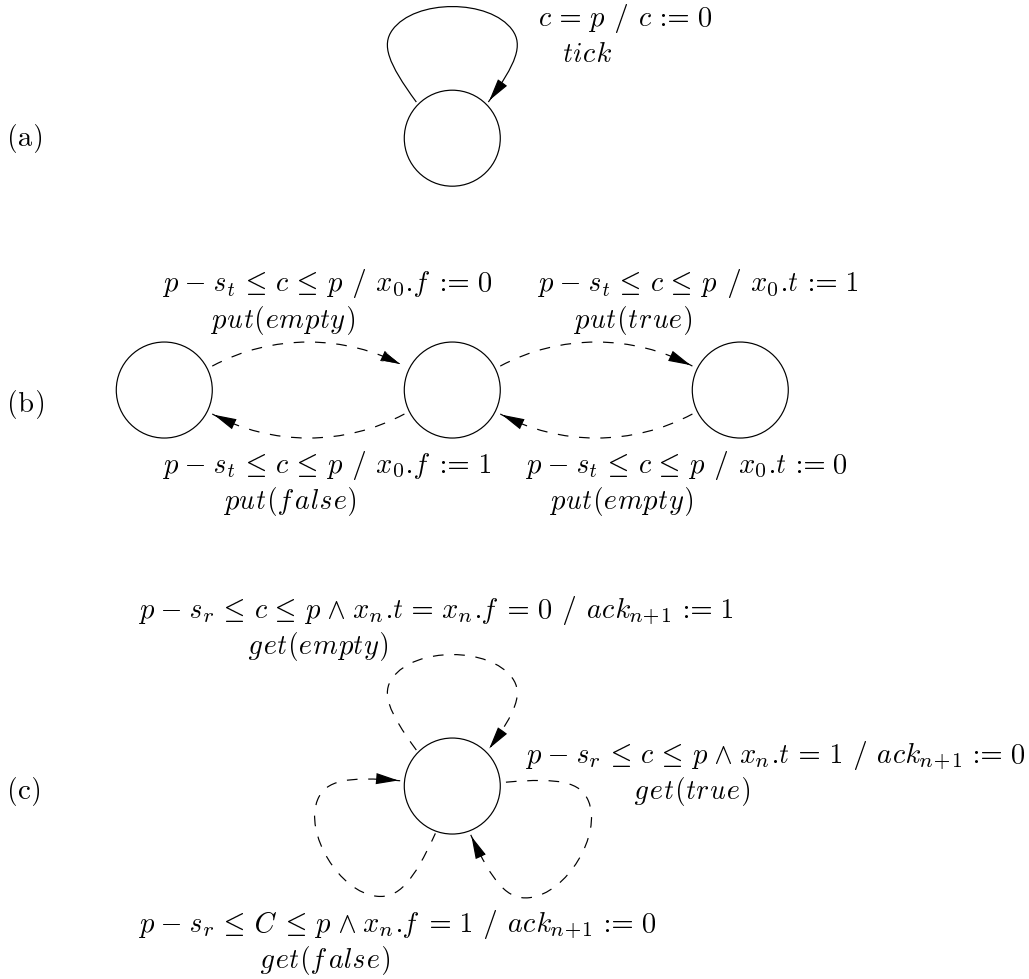

FIG. 7.12 – *(a) L'horloge globale, (b) l'émetteur et (c) le récepteur.*

7.3.3 Validation

Temps discret

Nous avons analysé ce modèle pour différents nombres n de tranches. Pour tous les casΓ nous avons composé les automatesΓexploré symboliquement le graphe d'états du produitΓ et minimisé ce graphe en gardant comme observables uniquement les actions *put* et *get*. Ces actions sont suffisantes pour vérifier le bon fonctionnement du circuitΓcaractérisé par l'absence de pertes et de duplications à l'intérieur de la file.

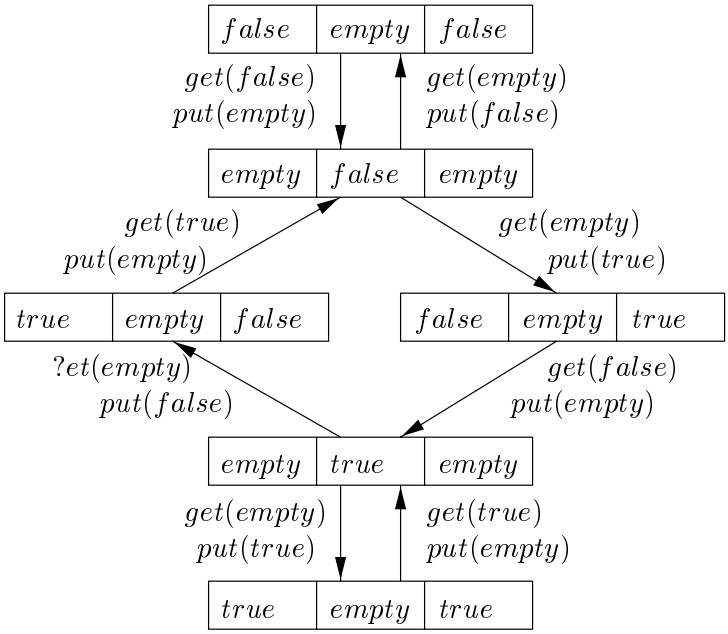

FIG. 7.13 – *Une file idéale de taille $m = 3$.*

Nous avons utilisés les valeurs $[l_c, u_c] = [l_n, u_n] = [2, 4]$ pour les délaisΓune période d'horloge $p = 12$ et les bornes de déviations maximales $s_t = s_r = 2$. Nous avons réussi à vérifier que toute instance de STARI avec $n \leq 18$Γbien initialisés avec $m \sim n/2$ valeursΓsimule une file idéal de taille m (voir figure 7.13). Plus exactementΓnous avons vérifié que le système de transitions du circuit et la file idéale sont équivalentes par rapport à la bisimulation de branchement [vGW89]. L'équivalence a été vérifiée à l'aide d'ALDEBARANΓen utilisant l'algorithme symbolique de minimisation décrit dans [FKM93]. Les performances en temps et en mémoire sont présentées dans la figure 7.14Γpar rapport au nombre de tranches.

Pour avoir une idée plus préciseΓl'instance du circuit avec $n = 18$ tranches est modélisée par 57 automatesΓcomportant 55 horloges. Elle nécessite 286 variables booléennes pour être représentée symboliquement à l'aide de BDDs. Son espace d'états atteignable est de l'ordre de 10^{15} états.

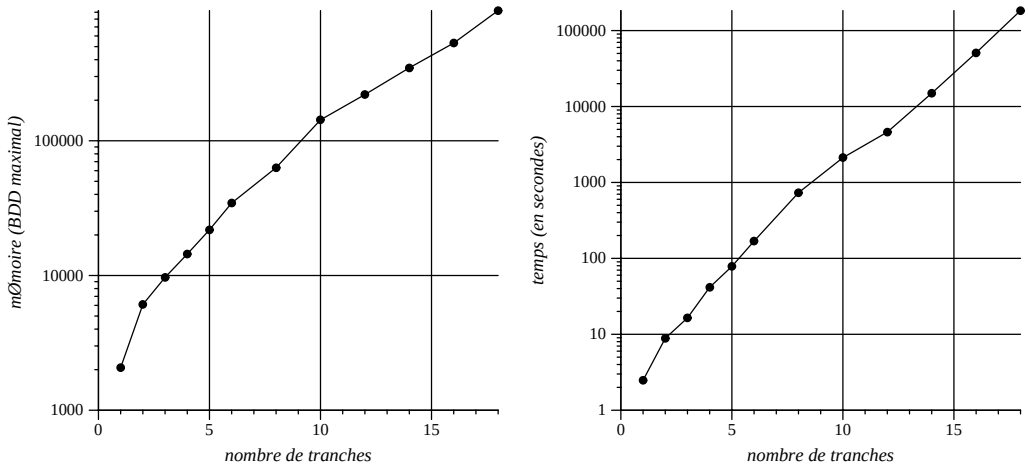

FIG. 7.14 – *Performances de la vérification à base de BDDs.*

Temps dense

Avec les même paramètresΓnous avons effectué aussi des vérifications en prenant l'interprétation du temps dense. Ces vérifications ont été réalises à l'aide de l'outil KRONOS et ont été présentées dans [Tri98].

Pour éviter l'explosion d'étatsΓnous avons pu efficacement exploiter l'activité des horloges. En faitΓà part l'horloge globaleΓtoutes les autres sont utilisées uniquement quand les portes associées changent de valeur. L'intuition sur le fonctionnement du circuit nous a fait penser que il y a peu de portes *actives* en même temps. Cette intuition a été vérifiée dans la pratique et nous a permis de traiter des instances du circuit avec $n \leq 8$ tranchesΓcomportant 25 horloges. En effetΓdans la plupart des états explorés il y a que 6 horloges qui sont activesΓ et jamais plus de 9. La distribution exacte de l'activité est montrée dans la figure 7.15.

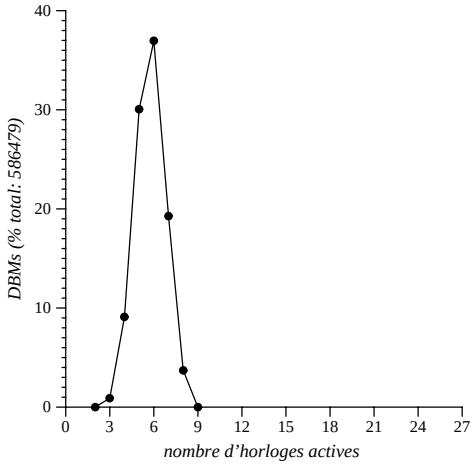

FIG. 7.15 – *Distribution de l'activité d'horloges.*

7.3.4 Observations

Une première vérification de la correction de STARI a été faite par [Gre93]. Il s'agit d'une preuve assistée par un démonstrateur et a nécessité une importante intervention de la part de l'utilisateur. Une autre approcheΓcette fois automatiqueΓa été suivi par [HBAB93]. La correction a été vérifiée en utilisant une technique à base de réseaux de PetriΓqui consiste à trouver la durée de séparationΓminimale et maximaleΓentre deux événements. FinalementΓ une vérification à base d'automates temporisés à été réalisée par [TB97]. D'abordΓils ont conçu une abstraction d'une tranche par un automate temporisé avec un seule horloge. Cette abstraction a été ensuite vérifie pour $n \leq 8$ tranches à l'aide de l'outil Timed-COSPAN.

Conclusion

Bilan

Ce travail se situe dans le cadre de l'utilisation de méthodes formelles pour la conception et la validation de protocoles de télécommunication. Nous avons poursuivi trois objectifs : la définition d'une représentation intermédiaire expressive pour décrire le fonctionnement des protocolesΓl'emploi de l'analyse statique pour la validationΓet la mise en œuvre d'un environnement robusteΓcomplet et ouvert de validation.

Formalismes de description

D'abordΓnous avons étudié la sémantique dynamique du langage de spécification SDLΓactuellement le plus utilisé dans le domaine des télécommunications. Nous nous sommes intéressés plus particulièrement aux aspects temporelsΓde plus en plus importants pour établir le bon fonctionnement des protocoles. Les conclusions retenues ont été le manque d'une sémantique adéquate pour le tempsΓainsi que l'absence d'outils d'analyse et de validation correspondants.

Nous avons défini par suite la représentation intermédiaire IF. Cette représentation est construite à base d'automates temporisés à échéances et permet ainsi la représentation non-ambiguë de l'urgence et d'autres contraintes temporelles. De plusΓIF comporte à la fois la communication asynchrone par différents types de files d'attenteΓet la communication synchrone par rendez-vous. FinalementΓcette représentation permet l'utilisation explicite des données.

Telle que cette représentation a été définieΓIF assure le lien entre des formalismes de description formelle standardisés et des formalismes mathématiques utilisés pour la validation. Grâce aux primitives de description fourniesΓla représentation IF d'un protocole reste en amont de problème d'explosion des états. De plusΓces primitives sont suffisamment expressives pour traduire un sous-ensemble statiqueΓsans création dynamique de processusΓde SDL. FinalementΓIF a une sémantique opérationnelle bien définie en termes de systèmes de transitions étiquetéesΓqui permet la mise en œuvre de techniques de compilation et de simulation très efficace.

Analyse statique

L'utilisation de l'analyse statique est désormais nécessaireΓétant donnée la complexité accrue des protocoles. Nous avons adapté quelques techniques classiques d'analyse de flot de données au cadre de la validation basée sur les modèles. En faitΓdes techniques comme l'analyse de variables actives ou la propagation de constantesΓhabituellement employées pour l'optimisation de codeΓpeuvent être re-utilisées pour réduire le modèle et ainsi améliorer considérablement les performances de la validation. Malgré leur simplicitéΓles résultats obtenus par l'application de ces techniques sur des protocoles ont été spectaculaires. En particulierΓl'exploitation de l'activité des variablesΓincluant la propagation à l'intérieur de filesΓpeut donner lieu à des réductions très importants du modèle.

Environnement de validation

EnfinΓles techniques d'analyse statique mentionnées et d'autres déjà existantes ont été intégrées au sein d'un environnement pour la validation de protocoles. Nous avons contribué au développement d'un important nombre de logicielsΓautant des bibliothèques que des outils. Il s'agit principalement de la compilation de IFΓincluant l'analyse statique et la simulation. Nous avons aussi contribué à la réalisation du traducteur de SDL vers IFΓet à des connexions avec d'autres plate-formes de validation. Ce travail englobe plus de 25000 lignes de code C et C++. ActuellementΓces outils ont atteint une certaine maturité et font partie des logiciels distribués par le laboratoire VERIMAG.

Pendant tout ce temps nous avons travaillé sur l'application effective de nos méthodes et outils sur des études de cas. Ce travail a contribué de manière essentielle à identifier les vrais problèmesΓet à valider notre technologie. Pour illustrer concrètement l'applicabilité de IF nous avons présenté ici les résultats concrets obtenus sur trois protocoles de communication. De plusΓnotons que IF et les outils associés sont actuellement utilisés dans plusieurs projets menés au sein du laboratoire VERIMAG comme par exemple le projet européen VIRESΓsur la spécification et la vérification d'un protocole ATM sans fil (MASCARA)Γet le projet PROUSTΓsur l'étude des aspects temporisés et évaluation de performance en SDL.

Perspectives

Trouver le modèle idéal pour la représentation et l'analyse des systèmes asynchronesΓet en particulier des protocoles de communicationΓest un problème encore ouvert. Un certain nombre de perspectives intéressantes restent à explorer. Nous les avons groupées en deux catégoriesΓla première concerne la représentation intermédiaire et la deuxième les méthodes et les techniques d'analyse.

La représentation intermédiaire

Un certain nombre des concepts s'avèrent encore nécessaires pour la modélisation et l'analyse des protocoles. Parmi celles-ci nous mentionnons à titre d'exemple les priorités discrètes associées aux transitionsΓles files avec délaisΓet la relâche des contraintes statiques.

Une extension simple est l'introduction de priorités entre les transitions d'un processus. Ces priorités compléteraient de manière très naturelle les échéances existantesΓqui définissent uniquement l'urgence de transitions par rapport à la progression du temps. De plusΓSDL comporte des priorités implicites entre différents types d'entrées et aussi des priorités explicites entre des entrées continues. La traduction dans un modèle sans priorités peut être à priori effectuée ; cependant elle peut donner lieu à des modèles complexes et même parfois illisibles à cause des gardes supplémentaires. La prise en compte des priorités au niveau du modèle intermédiaire doit être donc envisagée. CertainementΓau niveau d'un seul automateΓ les priorités n'apportent pas plus d'expressivité et ne posent aucun problème. En revancheΓ leur utilisation s'avère plus délicate face à la composition parallèle avec synchronisation.

Les délais dans les files sont nécessaires pour modéliser les retards habituels existants dans les lignes de communication. En ce sensΓles canaux de SDL peuvent avoir un délai de transmission arbitraire. AinsiΓun signal entrant le canal au moment t_0 devient disponible en sortie à un moment $t \in [t_0, \infty)$. Le fonctionnement d'un tel canal peut être simulé simplement à l'aide d'un processus supplémentaire. CependantΓnous voulons définir des modèles plus expressifs comme par exemple des canaux avec délai de type intervalle $[l, u]$. Un signal envoyé via un tel canal au moment t_0 devrait ainsi être disponible en sortie de manière arbitraire à un moment $t \in [t_0 + l, t_0 + u]$. La difficulté principale est qu'un tel modèle peut exiger l'utilisation d'un nombre non-borné d'horloges : un pour chaque signal en transit dans les files.

Des extensions plus ambitieuses concernent la définition d'une représentation intermédiaire dynamique. ActuellementΓpour des raisons d'efficacitéΓaucune forme de création dynamique n'est autorisé en IF. Cette contrainte est cependant très restrictive et on pourra envisager la définition autorisant la création dynamique des processus et des files. En outreΓles files avec delais peuvent exiger aussi la création dynamique d'horloges. Une telle extension peut limiter l'application de certaines techniques de validationΓcomme par exemple l'analyse symbolique à base de BDDs. En revancheΓelle peut toujours bénéficier de l'application de techniques d'analyse statique ou des techniques de validation à la volée.

Méthodes et techniques d'analyse

Il existe bien d'autres techniques d'analyse statique qui semblent pouvoir fournir de l'aide pour la validation. Nous en avons expérimenté quelques unesΓmais le spectre d'analyses effectuées par les compilateurs en comporte beaucoup d'autres qui méritent d'être explorées. Nous voyons deux directions d'applications. La première consiste à identifier les manières les plus appropriées d'utiliser ces techniques pour la validation. Nous nous sommes en particulier intéressé à la réduction du modèle pendant la simulationΓmais bien d'autres solutions sont

envisageables. Par exemple [KLM+98] présente une combinaison intéressante de l'analyse statique avec une méthode de réduction d'ordre partiel. Comme deuxième direction nous devrons considérer aussi la spécialisation des techniques d'analyse statique par rapport à des propriétés à vérifier. Des résultats existent déjà est sont basés principalement sur le calcul d'abstractions plus ou moins fines en fonction des propriétés. D'autre solutions sont cependant envisageables. Par exemple nous avons défini la bisimulation d'activité. Cette bisimulation est trop forte : elle préserve toutes les propriétés du modèle initial. On pourra envisager étant donné une propriété Φ la définition d'une bisimulation plus faible à base de l'activité et de Φ autorisant plus de réduction mais qui préserve strictement Φ.

Une autre perspective concerne la validation de systèmes infinis. Par nature les protocoles peuvent avoir des modèles infinis : les horloges les files et mêmes les variables ont toutes des domaines de valeurs à priori infinis même si le contrôle est fini. Pour des raisons comme l'efficacité et l'automatisation nous nous sommes concentré sur la validation de protocoles ayant des modèles finis. Cependant aujourd'hui il est tout à fait possible et réaliste de s'attaquer aussi à des protocoles avec modèles infinis. D'une part on a des résultats récents pour la vérification algorithmique de systèmes infinis d'une forme particulière comme par exemple des systèmes à files avec pertes ou des systèmes à compteurs. De manière plus générale une autre approche comporte l'utilisation des méthodes déductives. Ces méthodes s'utilisent de plus en plus pour la validation conjointement avec des méthodes basées sur les modèles. Nous avons expérimenté une telle connexion avec l'outil INVEST. Les résultats obtenus ont été très bons mais il reste plusieurs pistes à explorer.

Finalement une dernière perspective sera la mise en œuvre de techniques pour l'évaluation de performances. Ces techniques occupent une place bien définie dans la chaîne de conception de protocoles. Elles sont basées sur des modèles mathématiques probabilistes du système à vérifier et consistent à calculer certaines mesures importantes (e.g le temps moyen d'exécution d'une tâche). Dans ce contexte nous avons travaillé pour la définition d'une représentation symbolique adéquate et l'analyse du comportement asymptotique des chaînes de Markov [BM99].

Bibliographie

[AB] Telelogic AB. *SDT Reference Manual*. `http://www.telelogic.se`.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data Structures for the Verification of Timed Automata. In O. Maler, editor, *Proceedings of HART'97 (Grenoble, France)*, volume 1201 of *LNCS*, pages 346–360. Springer, April 1997.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model Checking in Dense Real Time. *Information and Computation*, 104(1):1–34, 1993.

[AD94] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[ALH95] B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *Proceedings of SDL FORUM'95*. Elsevier, 1995.

[AMP98] E. Asarin, O. Maler, and A. Pnueli. On the Discretization of Delays in Timed Automata and Digital Circuits. In R. de Simone and D. Sangiorgi, editors, *Proceedings of CONCUR'98*, volume 1466 of *LNCS*, pages 470–484. Springer, 1998.

[ASU86] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Readings, MA, 1986.

[BB88] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *ISDN*, 14(1):25–29, January 1988.

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic Generation of Invariants and Intermediate Assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} states and beyond. In *Proceedings of LICS'90 (Philadelphia, USA)*, pages 428–439, June 1990.

[BD88] S. Budkowski and P. Dembinski. An Introduction to ESTELLE: A Specification Language for Distributed Systems. *ISDN*, 14, January 1988.

[BD98] D. Bošnački and D. Dams. Integrating Real Time into Spin: A Prototype Implementation. In *Proceedings of the FORTE/PSTV'98 (Paris, France)*Γpages 423–438ΓOctober 1998.

[BDHS00] D. BošnačkiΓD. DamsΓL. HolenderskiΓand N. Sidorova. Model Checking SDL with Spin. In S. Graf and M. SchwartzbachΓeditorsΓ*Proceedings of TACAS'2000 (Berlin, Germany)*ΓLNCS. SpringerΓMarch 2000. to appear.

[BDM+98] M. BozgaΓC. DawsΓO. MalerΓA. OliveroΓS. TripakisΓand S. Yovine. Kronos: a Model-Checking Tool for Real-Time Systems. In A. Hu and M. VardiΓeditorsΓ *Proceedings of CAV'98 (Vancouver, Canada)*Γvolume 1427 of *LNCS*Γpages 546–549. SpringerΓJune 1998.

[BFG+91] A. BouajjaniΓJ.Cl. FernandezΓS. GrafΓC. RodriguezΓand J. Sifakis. Safety for Branching Time Semantics. In *Proceedings of ICALP'91*Γvolume 510 of *LNCS*. SpringerΓJuly 1991.

[BFG+98] M. BozgaΓJ.Cl. FernandezΓL. GhirvuΓC. JardΓT. JéronΓA. KerbratΓP. MorelΓ and L. Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*Γ1998. to appear.

[BFG99] M. BozgaΓJ.Cl. FernandezΓand L. Ghirvu. State Space Reduction based on Live Variables Analysis. In A. Cortesi and G. FilέΓeditorsΓ*Proceedings of SAS'99 (Venice, Italy)*Γvolume 1694 of *LNCS*Γpages 164–178. SpringerΓSeptember 1999.

[BFKM97] M. BozgaΓ J.Cl. FernandezΓ A. KerbratΓ and L. Mounier. Protocol Verification with the Aldebaran Toolset. *Software Tools for Technology Transfer*Γ 1(1+2):166–183ΓDecember 1997.

[BL99] S. Bensalem and Y. Lakhnech. Automatic Generation of Invariants. *Formal Methods in System Design*Γ15(1):75–92ΓJuly 1999.

[BLM97] N. BjørnerΓU. LernerΓand Z. Manna. Deductive Verification of Parameterized Fault Tolerant Systems: a Case Study. In *Proceedings of ICTL'97*Γ1997.

[BLO98] S. BensalemΓY. LakhnechΓand S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In A. Hu and M. VardiΓ editorsΓ*Proceedings of CAV'98 (Vancouver, Canada)*Γvolume 1427 of *LNCS*Γ pages 319–331. SpringerΓJune 1998.

[BLP+99] G. BehrmannΓK.G. LarsenΓJ. PearsonΓC. WeiseΓand W. Yi. Efficient Timed Reachability Analysis using Clock Difference Diagrams. In N. Halbwachs and D. PeledΓeditorsΓ*Proceedings of CAV'99 (Trento, Italy)*Γvolume 1633 of *LNCS*Γ pages 341–353. SpringerΓJuly 1999.

[BM95] J.A. Bergstra and C.A. Middelburg. Process Algebra Semantics of φSDL. In *Proceedings of 2nd Workshop on ACP*Γ1995.

[BM99] M. Bozga and O. Maler. On the Representation of Probabilities over Structured Domains. In N. Halbwachs and D. Peled editors *Proceedings of CAV'99 (Trento, Italy)* volume 1633 of *LNCS* pages 261–273. Springer July 1999.

[BMPY97] M. Bozga O. Maler A. Pnueli and S. Yovine. Some Progress in the Symbolic Verification of Timed Automata. In O. Grumberg editor *Proceedings of CAV'97 (Haifa, Israel)* volume 1254 of *LNCS* pages 179–190. Springer June 1997.

[BMT99] M. Bozga O. Maler and S. Tripakis. Modeling and Verification of the STARI Chip Using Timed Automata. In L. Pierre editor *Proceedings of CHARMÉ99 (Bad Herrenalb, Germany)* volume 1703 of *LNCS* pages 125–141. Springer September 1999.

[BMU98] J.A. Bergstra C.A. Middelburg and Y.S. Usenko. Discrete Time Process Algebra and the Semantics of SDL. Technical Report SEN-R9809 CWI Amsterdam The Netherlands June 1998.

[Bor98] S. Bornot. *De la composition de systèmes temporisés.* PhD thesis Université Joseph Fourier Grenoble France December 1998.

[Bou92] F. Bourdoncle. *Sémantique des langages impératif d'ordre supérieur et interprétation abstraite.* PhD thesis Ecole Polytechnique 1992.

[Boz96] M. Bozga. Vérification formelles de systèmes distribués et diagrammes de décision multivalués. Master's thesis Université Joseph Fourier Grenoble France June 1996.

[BRB90] K.S. Brace R.L. Rudell and R.E. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of 27th ACM/IEEE Design Automation Conference* pages 40–45 1990.

[Bri88] E. Brinksma. A Theory for the Derivation of Tests. In *Proceedings of PSTV'88* 1988.

[Bro91] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects on Computing* 1991.

[Bry86] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computation* 35(8) 1986.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. Technical Report Carnegie Mellon University Pittsburgh PA 15213 1992.

[BST97] S. Bornot J. Sifakis and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)* volume 1536 of *LNCS.* Springer September 1997.

[CBM89] O. CoudertΓC. BerthetΓand J.C. Madre. Verification of Synchronous Sequential Machines based on Symbolic Execution. In *International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*Γvolume 407 of *LNCS*. SpringerΓ1989.

[CES83] E.M. ClarkeΓE.A. EmersonΓand E. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of 10th ACM Symposium on Programming Languages*. ACM PressΓ1983.

[CGJ98] C. ColbyΓP. GodefroidΓand L.J. Jagadeesan. Automatically Closing Open Reactive Systems. In *Proceedings of ACM SIGPLAN on Programming Language Design and Implementation (Montreal, Canada)*Γpages 345–357ΓJune 1998.

[CGL94] E.M. ClarkeΓO. GrumbergΓand D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*Γ16(5)Γ1994.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of ACM Symposium on Principles of Programming Languages*Γ1978.

[Cor94] Digital Equipment Corporation. *TiGeR Library Manual Reference*Γ1994.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximations de points fixes d'opérateurs monotones sur un treillis. Analyse sémantique des programmes séquentiels*. PhD thesisΓUniversité Scientifique et Medicale de GrenobleΓGrenobleΓFranceΓ1978.

[CR79] E. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM*Γ 22(5)Γ1979.

[CS93] R. Cleaveland and B. Steffen. A Linear Time Model Checking Algorithm for the Alternation Free Modal Mu-Calculus. *Formal Methods in System Design*Γ1993.

[CT96] C. Courcoubetis and S. Tripakis. Extending Promela and Spin for Real Time. In T. Margaria and B. SteffenΓeditorsΓ*Proceedings of TACAS'96 (Passau, Germany)*Γvolume 1055 of *LNCS*Γpages 329–348. SpringerΓMarch 1996.

[CVWY90] C. CourcoubetisΓM. VardiΓP. WolperΓand M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. In E. Clarke and R. KurshanΓeditorsΓ*Proceedings of CAV'90 (New Brunswick, USA)*Γvolume 3 of *DIMACS*Γpages 207–218. AMS/ACMΓJune 1990.

[Daw98] C. Daws. *Méthodes d'analyse de systèmes temporisés : de la théorie à la pratique*. PhD thesisΓInstitut National Polytechnique de GrenobleΓGrenobleΓFranceΓOctober 1998.

[DH99] M.B. Dwyer and J. Hatcliff. Slicing Software for Model Construction. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*ΓJanuary 1999.

[Dil89] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*Γvolume 407 of *LNCS*. SpringerΓ1989.

[EL86] E.A. Emerson and C-L. Lei. Efficient Model Checking in Fragments of the Propositional Mu-Calculus. In *Proceedings of LICS'86*Γ1986.

[EM85] E. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications*. Springer-VerlagΓ1985.

[Fer88] J.Cl. Fernandez. *ALDEBARAN: un système de vérification par réduction de processus communicants*. PhD thesisΓ Université Joseph FourierΓ GrenobleΓ FranceΓMay 1988.

[FG98] H. Fleischhack and B. Grahlmann. A Compositional Petri Net Semantics for SDL. In *Proceedings of ATPN'98 (Application and Theory of Petri Nets) (Lisboa, Portugal)*Γvolume 1420 of *LNCS*Γpages 144–164. SpringerΓJune 1998.

[FGK+96] J.Cl. FernandezΓH. GaravelΓA. KerbratΓR. MateescuΓL. MounierΓand M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T.A. HenzingerΓeditorsΓ*Proceedings of CAV'96 (New Brunswick, USA)*Γ volume 1102 of *LNCS*Γpages 437–440. SpringerΓAugust 1996.

[FJJM92] J.Cl. FernandezΓC. JardΓT. JéronΓand L. Mounier. "On the Fly" Verification of Finite Transition Systems. *Formal Methods in System Design*Γ1992.

[FJJV97] J.Cl. FernandezΓC. JardΓT. JéronΓand C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*Γ29Γ1997.

[FKM93] J.Cl. FernandezΓA. KerbratΓand L. Mounier. Symbolic Equivalence Checking. In C. CourcoubetisΓeditorΓ*Proceedings of CAV'93 (Heraklion, Greece)*Γvolume 697 of *LNCS*Γpages 85–96. SpringerΓJune 1993.

[FM90] J.Cl. Fernandez and L. Mounier. Verifying Bisimulations "On the Fly". In J. QuemadaΓ J. ManasΓ and E. VázquezΓ editorsΓ *Proceedings of FORTE/PSTV'90 (Madrid, Spain)*. North HollandΓNovember 1990.

[FM91] J.Cl. Fernandez and L. Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K.G. Larsen and A. SkouΓeditorsΓ*Proceedings of CAV'91 (Aalborg, Denmark)*Γvolume 575 of *LNCS*Γpages 181–191. SpringerΓ July 1991.

[FS90] S.J. Friedman and K.J. Supowit. Finding the Optimal Variable Ordering for Binary Decision Diagrams. *IEEE Transactions on Computers*Γ39(5):710–713Γ May 1990.

[Gar89a] H. Garavel. *Compilation et vérification de programmes LOTOS*. PhD thesisΓ Université Joseph FourierΓGrenobleΓNovember 1989.

[Gar89b] H. Garavel. Compilation of LOTOS Abstract Data Types. In Son T. VuongΓ editorΓ*Proceedings of FORTE/PSTV'89 (Vancouver, Canada)*Γpages 147–162. North HollandΓDecember 1989.

[Gar98] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for VerificationΓ SimulationΓand Testing. In B. SteffenΓeditorΓ*Proceedings of TACAS'98 (Lisbon, Portugal)*Γvolume 1384 of *LNCS*Γpages 68–84. SpringerΓMarch 1998.

[GL93] S. Graf and C. Loiseaux. A Tool for Symbolic Program Verification and Abstraction. In C. CourcoubetisΓeditorΓ*Proceedings of CAV'93 (Heraklion, Greece)*Γ volume 697 of *LNCS*Γpages 71–84. SpringerΓJune 1993.

[GM96] H. Garavel and L. Mounier. Specification and Verification of Distributed Leader Election Algorithms for Unidirectional Ring Networks. *Science of Computer Programming*Γ29(1–2):171–197ΓJuly 1996.

[God91] J.C. Godskesen. An Operational Semantic Model for Basic SDL. Technical Report TFL RR 1991-2ΓTele Danmark ResearchΓ1991.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State Explosion Problem*Γvolume 1032 of *LNCS*. SpringerΓ January 1996. ISBN 3-540-60761-7.

[Gre] M.R. Greenstreet. STARI: Skew Tolerant Communication. 1997.

[Gre93] M.R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. PhD thesisΓPrinceton UniversityΓPrinceton CAΓ1993.

[GS90a] H. Garavel and J. Sifakis. Compilation and Verification of LOTOS Specifications. In L. LogrippoΓ R.L. ProbertΓ and H. UralΓ editorsΓ *Proceedings of PSTV'90 (Ottawa, Canada)*Γpages 379–394. North HollandΓJune 1990.

[GS90b] S. Graf and B. Steffen. Compositional Minimisation of Finite State Processes. In E. Clarke and R. KurshanΓeditorsΓ*Proceedings of CAV'90 (Rutgers, USA)*Γ volume 3 of *DIMACS*Γpages 57–74. AMS/ACMΓ1990.

[GV90] J.F. Groote and F. Vaandrager. An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. Technical Report CS-R9001ΓCWIΓAmsterdamΓ The NetherlandsΓJanuary 1990.

[GW91] P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K.G. Larsen and A. SkouΓeditorsΓ *Proceedings of CAV'91*Γ volume 575 of *LNCS*Γ pages 332–342. SpringerΓ July 1991.

[Hal79] N. Halbwachs. *Détermination automatique de rélations linéaires vérifiées par les variables d'un programme.* PhD thesisΓUniversité Scientifique et Medicale de GrenobleΓGrenobleΓFranceΓ1979.

[HBAB93] H. HulgaardΓS. BurnsΓT. AmonΓand G. Boriello. Practical Application of an Efficient Time Separation Algorithm. In *Proceedings of ICCAD'93*Γ1993.

[HD93] A.J. Hu and D. Dill. Efficient Verification wiyh BDDs using Implicitly Conjoined Invariants. In C. CourcoubetisΓeditorΓ*Proceedings of CAV'93 (Heraklion, Greece)*Γvolume 697 of *LNCS*Γpages 3–14. SpringerΓJune 1993.

[HMP92] T. HenzingerΓZ. MannaΓand A. Pnuelli. What Good Are Digital Clocks? In *Proceedings of ICALP'92*Γvolume 623 of *LNCS*. SpringerΓ1992.

[HNSY94] T.A. HenzingerΓX. NicollinΓJ. SifakisΓand S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*Γ111(2)Γ1994.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*Γ12Γ1969.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*Γ21(8)ΓAugust 1978.

[Hoa84] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall InternationalΓ 1984.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software SeriesΓ1991.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Technical Report 8807ΓInternational Organization for Standardization — Information Processing Systems — Open Systems InterconnectionΓGenèveΓ1988.

[ISO89] ISO/IEC. Estelle — A Formal Description Technique based on an Extended State Transition Model. Technical Report 9074ΓInternational Organization for Standardization — Information Processing Systems — Open Systems InterconnectionΓGenèveΓ1989.

[ISO92] ISO/IEC. 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). Technical Report 9646ΓInternational Organization for Standardization — Information Processing Systems — Open Systems InterconnectionΓGenèveΓ1992.

[IT94a] ITU-T. Annex F.2 to Recommendation Z.100. Specification and Description Language (SDL) - SDL Formal Definition: Static Semantics. Technical Report Z-100ΓInternational Telecommunication Union – Standardization SectorΓGenèveΓ 1994.

[IT94b] ITU-T. Annex F.3 to Recommendation Z.100. Specification and Description Language (SDL) - SDL Formal Definition: Dynamic Semantics. Technical Report Z-100ΓInternational Telecommunication Union – Standardization SectorΓ GenèveΓ1994.

[IT94c] ITU-T. Recommendation Q.2110. ATM Adaptation Layer - Service Specific Connection Oriented Protocol (SSCOP). Technical Report Q-2110ΓInternational Telecommunication Union – Standardization SectorΓGenèveΓ1994.

[IT94d] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100ΓInternational Telecommunication Union – Standardization SectorΓGenèveΓ1994.

[IT94e] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120ΓInternational Telecommunication Union – Standardization SectorΓGenèveΓ 1994.

[JJ89] T. Jeron and C. Jard. On-line Model-Checking for Finite Linear Temporal Logic. In *International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*Γvolume 407Γpages 275–285ΓJune 1989.

[JJ91] C. Jard and T. Jéron. Bounded Memory Algorithms for Verification On-the-fly. In K.G. Larsen and A. SkouΓeditorsΓ *Proceedings of CAV'91 (Aalbord, Denmark)*Γvolume 575 of *LNCS*Γpages 192–202. SpringerΓJuly 1991.

[JM99] T. Jéron and P. Morel. Test Generation Derived from Model Checking. In N. Halbwachs and D. PeledΓeditorsΓ *Proceedings of CAV'99 (Trento, Italy)*Γ volume 1633 of *LNCS*Γpages 108–122. SpringerΓJuly 1999.

[Kam90] T.Y. Kam. *Multi-valued Decision Diagrams.* PhD thesisΓUniversity of CaliforniaΓDepartment of Electrical Engineering and Computer ScienceΓBerkeley CA 94720Γ1990.

[Kil73] G. Kildall. An Unified Approach to Global Program Optimization. In *ACM Symposium on Principles of Programming Languages*Γ1973.

[KJG99] A. KerbratΓT. JéronΓand R. Groz. Methods and Methodology for Incremental Test Generation from SDL. In R. DssouliΓG. BochmannΓand Y. LahavΓeditorsΓ *Proceedings of SDL FORUM'99 (Montreal, Canada)*Γpages 135–152. ElsevierΓ June 1999.

[KLM+98] R. KurshanΓV. LevinΓM. MineaΓD. PeledΓand H. Yenigün. Static Partial Order Reduction. In B. SteffenΓeditorΓ *Proceedings of TACAS'98 (Lisbon, Portugal)*Γ volume 1384 of *LNCS*Γpages 345–357. SpringerΓMarch 1998.

[KM97] J.P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. In E. Brinksma, editor, *Proceedings of TACAS'97 (Enschede, The Netherlands)*, volume 1217 of *LNCS*, pages 239–258. Springer, April 1997.

[Koz83] D. Kozen. Results on the Propositional μ-Calculus. *Theoretical Computer Science*, 1983.

[KRL97] A. Kerbrat, C. Rodriguez, and Y. Lejeune. Interconnecting the ObjectGEODE and CADP Toolsets. In A. Cavalli and A. Sarma, editors, *Proceedings of SDL FORUM'97 (Evry, France)*, pages 475–490. Elsevier, September 1997.

[KS90] P. Kanellakis and S. Smolka. CCS Expressions, Finite State Processes and Three Problems of Equivalence. *Information and Computation*, 86(1), May 1990.

[Lam80] L. Lamport. "Sometimë is Sometimes "Not Never". On the Temporal Logic of Programs. In *Proceedings of ACM Symposium on Principles of Programming Languages*, January 1980.

[Lan77] G. Le Lann. Distributed Systems – Towards a Formal Approach. In *Information Processing*. North Holland, 1977.

[LL97] L. Leonard and G. Leduc. An Introduction to ET-LOTOS for the Description of Time-Sensitive Systems. *Computer Networks and ISDN Systems*, (29), 1997.

[Lon93] D.E. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, July 1993.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL: Status & Developments. In O. Grumberg, editor, *Proceedings of CAV'97 (Haifa, Israel)*, volume 1254 of *LNCS*, pages 456–459. Springer, June 1997.

[Mat98] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1998.

[McM93] K.L. McMillan. *Symbolic Model Checking: an Approach to the State Explosion Problem*. Kluwer Academic Publisher, 1993.

[Mil80] R. Milner. *A Calculus of Communication Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mou92] L. Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. PhD thesis, Université Joseph Fourier, Grenoble, France, January 1992.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MP95] O. Maler and A. Pnueli. Timing Analysis of Asynchronous Circuits using Timed Automata. In H. Eveking P.E. Camurati editor *Proceedings of CHARMÉ95* volume 987 of *LNCS* pages 189–205. Springer 1995.

[Muc97] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers 1997.

[NMV90] R. De Nicola U. Montanari and F. Vaandrager. Back and Forth Bisimulations. Technical Report CWI Amsterdam The Netherlands May 1990.

[NRSV89] X. Nicollin J-L. Richier J. Sifakis and J. Voiron. ATP: An Algebra for Timed Processes. In *Proceedings of the IFIP Working Conference on Programming Concepts and Methods* 1989.

[NV90] R. De Nicola and F. Vaandrager. Action versus State based Logics for Transition Systems. In *Proceedings Ecole de Printemps on Semantics of Concurrency* number 469 in LNCS 1990.

[ORSvH95] S. Owre J. Rushby N. Shankar and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering* 1995.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. *Theoretical Computer Science* 104:167–183 March 1981.

[Pel94] D. Peled. Combining Partial Order Reductions with On-The-Fly Model-Checking. In D. Dill editor *Proceedings of CAV'94* volume 818 of *LNCS* pages 377–390. Springer June 1994.

[Pha94] M. Phalippou. Test Sequence Generation using Estelle or SDL Structure Information. In D. Hogrefe and S. Leue editors *Proceedings of FORTE/PSTV'94 (Berne ,Switzerland)* pages 405–420 October 1994.

[Pnu85] A. Pnueli. In Transition From Global To Modular Temporal Reasoning About Programs. *Logics and Models for Concurrent Systems* 1985.

[PT87] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal of Computing* 16(6):973–989 December 1987.

[QS82] J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Programs in CESAR. In *International Symposium on Programming* volume 137 of *LNCS* pages 337–351 1982.

[Que98] J. Quemada. Final Comitee Draft on Enhancements to LOTOS. Technical Report ISO/IEC JTC1/SC33/WG9 April 1998.

[RBP+91] J. Rumbaugh M. Blaha W. Premerlani F. Edyy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall Inc. Englewood Cliffs 1991.

[Rod88] C. Rodriguez. *Spécification et validation de systémes en XESAR*. PhD thesisΓ Institut National Polytechnique de GrenobleΓGrenobleΓFranceΓ1988.

[Saa99] A. Saadani. Validation of SDL Specifications via Abstractions. Master's thesisΓ Ecole Politechnique de TunisieΓ1999.

[Sig99] M. Sighireanu. *Contribution at the Definition and Implementation of E-LOTOS*. PhD thesisΓUniversité Joseph FourierΓGrenobleΓFranceΓ1999.

[Som] F. Somenzi. *CUDD Decision Diagram Package*. Colorado University.

[ST87] R. Saracco and P.A.J. Tilanus. CCITT SDL: Overview of the Language and its Applications. *Computer Networks and ISDN Systems*Γ1987.

[TB97] S. Tasiran and R. Brayton. STARI: A Case Study in Compositional and Hierarchical Timing Verification. In O. GrumbergΓeditorΓ*Proceedings of CAV'97 (Haifa, Israel)*Γvolume 1254 of *LNCS*Γpages 191–201. SpringerΓJune 1997.

[Tip94] F. Tip. A Survey of Program Slicing Techniques. Technical Report CS-R9438Γ CWIΓAmsterdamΓThe NetherlandsΓ1994.

[Tre92] J. Tretmans. *A Formal Approcah to Conformance Testing*. PhD thesisΓUniversity of TwenteΓTwenteΓThe NetherlandsΓ1992.

[Tri98] S. Tripakis. *L'Analyse Formelle de Systèmes Temporisés en Pratique*. PhD thesisΓUniversité Joseph FourierΓGrenobleΓFranceΓDecember 1998.

[Val92] A. Valmari. A Stubborn Attack on State Explosion. *Formal Methods in System Design*Γ1992.

[Ver] Verilog. *ObjectGEODE Reference Manual*. `http://www.verilogusa.com/`.

[vG90] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum. Technical Report CS-R9029ΓCWIΓAmsterdamΓThe NetherlandsΓ1990.

[vGW89] R.J. van Glabbeek and W.P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics. Technical Report CS-R8911ΓCWIΓAmsterdamΓThe NetherlandsΓ1989.

[VL92] B. Vergauwen and J. Levi. A Linear Algorithm for Solving Fixed Point Equations on Transitions Systems. In *Proceedings of 17th Colloquium on Trees in Algebra and Programming (Rennes, France)*Γvolume 581 of *LNCS*. SpringerΓ1992.

[VL93] B. Vergauwen and J. Levi. A Linear Local Model Checking Algorithm for CTL. In *Proceedings of CONCUR'93 (Hildesheim, Germany)*Γvolume 715 of *LNCS*. SpringerΓ1993.

[Win90] G. Winskel. Compositional Checking of Validity on Finite State Processes. In *Workshop on Theories of Communication*Γvolume 458 of *LNCS*Γ1990.

[WZ91] M.N. Wegman and F.K. Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*Γ13(2)Γ April 1991.

[Yov97] S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*Γ1(1+2):123–133ΓDecember 1997.

Annexe A

Le μ-calcul modal

Syntaxe

Si A dénote un ensemble d'actions et $a \in A\Gamma$ la syntaxe des formules de μ-calcul est la suivante :

$$\Phi ::= tt \mid X \mid \neg\Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \nu X.\Phi$$

Une restriction syntaxique supplémentaire est faite sur les formules de la forme $\nu X.\Phi$: chaque occurrence de X dans Φ doit être sous la portée d'un nombre paire de négations.

Sémantique

Une formule est interprétée sur une système de transitions étiquetées $P = (A, Q, T)$ par l'intermédiaire d'une fonction $\xi\Gamma$qui associe un sous-ensemble d'états Q à chaque variable X de la formule. La sémantique d'une formule $\Phi\Gamma$notée $[[\Phi]]_\xi$ représente l'ensemble d'états de Q qui satisfont Φ :

$$\begin{aligned}
[[\ tt\ ]]_\xi &= Q \\
[[\ X\ ]]_\xi &= \xi(X) \\
[[\ \neg\Phi\ ]]_\xi &= Q \setminus [[\ \Phi\ ]]_\xi \\
[[\ \Phi_1 \wedge \Phi_2\ ]]_\xi &= [[\ \Phi_1\ ]]_\xi \cap [[\ \Phi_2\ ]] \\
[[\ [a]\Phi\ ]]_\xi &= \{\ q \in Q \mid \forall q'\ q \xrightarrow{a} q' \ \Rightarrow\ q' \in [[\ \Phi\ ]]_\xi\ \} \\
[[\ \nu X.\Phi\ ]]_\xi &= \bigcup\{\ S \mid [[\ \Phi\ ]]_{\xi[S/X]} \subseteq S\ \}
\end{aligned}$$

Macros

Afin de faciliter la description de propriétés nous utilisons les abréviations suivantes :

$$
\begin{aligned}
ff &= \neg tt \\
\Phi_1 \vee \Phi_2 &= \neg(\neg\Phi_1 \wedge \neg\Phi_2) \\
\langle a \rangle \Phi &= \neg[a]\neg\Phi \\
\mu X.\Phi &= \neg\nu X.\neg(\Phi[\neg X/X]) \\
\textbf{all } \Phi &= \nu X.\Phi \wedge [*]X \\
\textbf{pot } \Phi &= \mu X.\Phi \vee \langle * \rangle X \\
\textbf{inev } \Phi &= \mu X.\Phi \vee \langle * \rangle X \wedge [*]tt
\end{aligned}
$$

Alternation

La complexité des algorithmes d'évaluation des formules de μ-calcul dépend de la profondeur d'alternation de points fixes. Intuitivement il s'agit de la longueur maximale des chaînes de sous-formules mutuellement récursives avec des point fixes différents. Par exemple les formules $(\mu X.X) \wedge (\nu Y.\langle a \rangle Y)$ et $\mu X.\mu Y.(X \wedge Y)$ ont la profondeur d'alternation 1 et la formule $\mu X.\nu Y.(X \wedge Y)$ a la profondeur d'alternation 2.

Annexe B

Syntaxe graphique de SDL

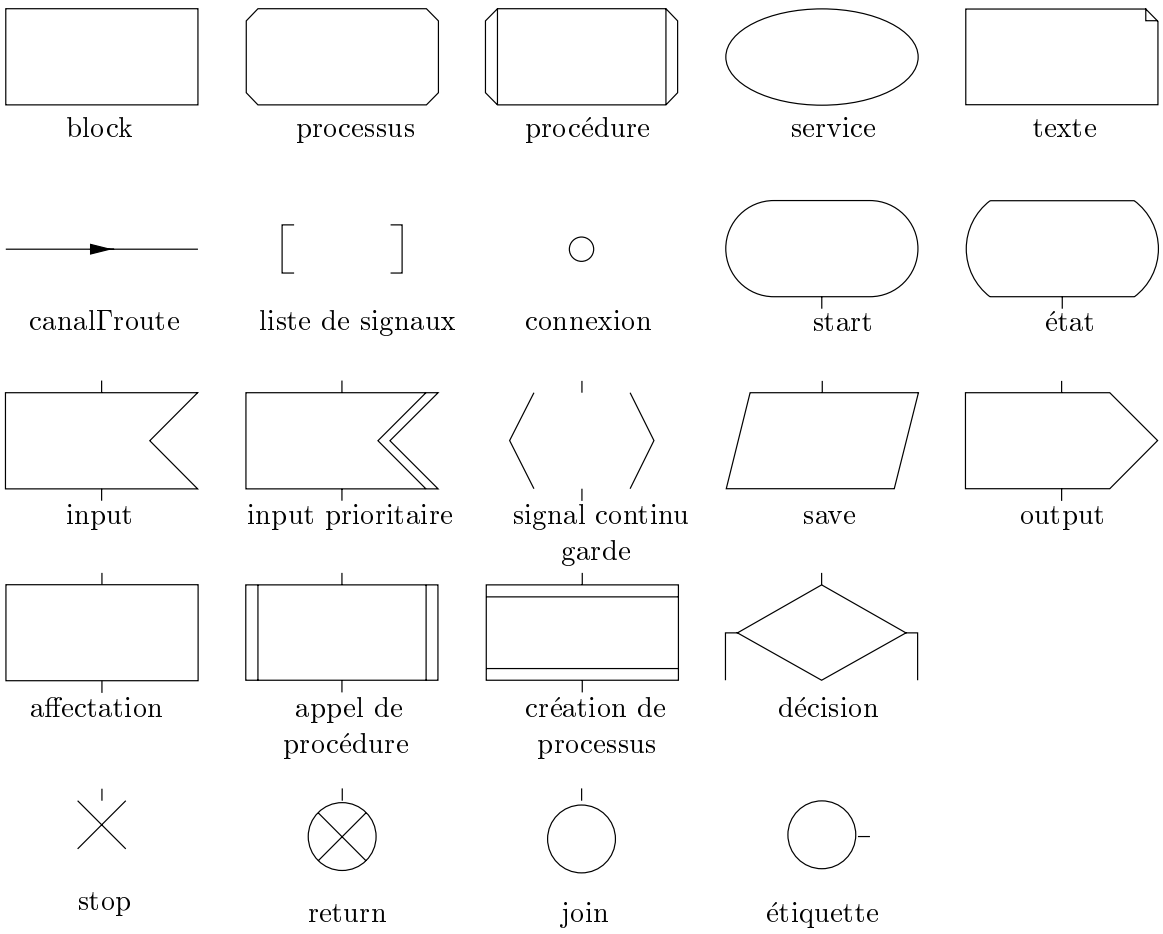

FIG. B.1 – *Symboles graphiques du SDL.*