2009

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale (Matisse)

présentée par

Xavier Dumas

préparée à l'unité de recherche (n^o + nom abrégé)
(Nom développé de l'unité)
(Composante universitaire)

Application des méthodes par ordres-partiels à la vérification formelle de systèmes asynchrones clos par un contexte : application à SDL

Thèse soutenue à Toulouse
le (date)

devant le jury composé de :

Benoît Baudry
Chargé de recherche, INRIA Rennes / examinateur

Hassan Mountassir
Professeur, Université de Besançon / rapporteur

Ioannis Parissis
Professeur, Université de Valence / rapporteur

Hélène Waeselynck
Chargé de recherche, CNRS Toulouse / examinateur

Virginie Wiels
Maître de recherche, ONERA / examinateur

Eric Bonnafous
Directeur technique, CS-SI Toulouse / examinateur

Frédéric Boniol
Professeur, ONERA / Directeur de thèse

Philippe Dhaussy
Maître de conférence, ENSIETA Brest / Co-directeur de thèse

Table des matières

Première partie

Introduction

Chapitre 1

Introduction

Sommaire

1.1 Contexte : les systèmes réactifs critiques asynchrones

Les systèmes réactifs deviennent extrêmement complexes avec l'essor des hautes technologies. Malgré les progrès techniques, la grande taille de ces systèmes facilite l'introduction d'une plus grande gamme d'erreurs. Parmi ces systèmes, les systèmes asynchrones, composés de sous-systèmes communiquant par échange de messages via des files d'attentes (par exemples des protocoles, des systèmes de communication bord/sol...), apportent une complexité encore supérieure. Actuellement, les industries engagent tous leurs efforts dans le processus de tests et de simulations à des fins de certification. Néanmoins, ces techniques deviennent rapidement inexploitable pour débusquer des erreurs pouvant conduire à des situations catastrophiques. La couverture des jeux de tests s'amincit au fur et à mesure de la complexification des systèmes, et il devient nécessaire d'utiliser de nouvelles méthodes. Parmi celles-ci, les méthodes formelles ont contribué, depuis plusieurs années, à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. Dans ce domaine, les techniques de *model-checking* ont été fortement popularisées grâce à leur faculté d'exécuter, plus ou moins automatiquement, des preuves de propriétés sur des modèles logiciels [CE82, QS82].

Malgré la performance croissante des outils de *model-checking*, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes, dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel. Les techniques de vérification formelle souffrent aussi du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être validé. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités.

Ces dernières années, le combat contre l'explosion combinatoire est devenu le saint graal des chercheurs en méthodes formelles. Des techniques développées autour du *model-checking* consistent en général, à diminuer l'espace d'état du système global, de telle manière à pouvoir effectuer une recherche d'accessibilité d'états. Parmi ces techniques, les méthodes d'abstraction permettent d'éliminer (abstraire) des informations du système inutiles pour les propriétés à vérifier, et ainsi alléger le poids (en terme de mémoire nécessaire) de la vérification. Des méthodes symboliques [McM92] ont pour objectif de factoriser un ensemble d'états en un seul, de manière à les parcourir tous en une seule étape. Ces ensembles d'états sont représentés à la manière d'une "couche d'oignon", et calculés par des méthodes dites d'accélération (*widening*). D'autres méthodes qui ont vu le jour dans les années 90 tirent avantage de la structure même du système. La méthode de réduction par symétrie a été étudiée dans [ESD97], et génère une réduction de l'automate très importante lorsque de nombreux composants identiques sont présents. Les méthodes basées sur les ordres-partiels [God96, HP94, Val91, Ove81, ABH+97], permettent, quant à elles, de réduire l'automate global du système, en éliminant des séquences de transitions équivalentes sans effets sur la propriété observée.

Parmi les nombreux *model-checkers* proposés, le *model-checker* SPIN[1] [Hol97], intégrant le langage formel Promela, a été développé par G.J. Holzmann. L'outil permet la vérification de propriétés LTL [MP92a] qui sont encodées en automates de Büchi. Les techniques de calcul des ordres-partiels ont été implantés dans SPIN [BH05].

1.2 La prise en compte de l'environnement

L'approche dans laquelle s'insère ce travail, prend un point de vue complémentaire en considérant la réduction, non plus du système mais plutôt de son environnement. En effet, l'idée est d'abord de réduire les comportements du système en décrivant un cas d'utilisation particulier de l'environnement avec lequel le système interagit. L'objectif est de guider le *model-checker* à concentrer ses efforts, non plus sur l'exploration de l'automate global, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques.

[1] Simple Promela Interpreter

Un système réactif communique perpétuellement avec son environnement de manière à répondre à ses sollicitations. Pour appliquer la méthode du *model-checking*, il est donc nécessaire de fermer ce système en décrivant son environnement. En général, les environnements ne sont pas quelconques et sont bien déterminés. Ces derniers, sont définis par les connaissances de l'ingénieur sur les modes opératoires du système. De plus, les exigences de la spécification sont dans la majorité des cas écrites selon un environnement bien spécifique tels que les phases d'initialisation, de communication, de changement de modes, de reconfiguration ou encore les modes dégradés. Il devient donc inutile de prendre en compte la totalité de l'environnement pour la vérification de chaque exigence, mais uniquement une partie pertinente et fidèle au comportement du système et à la propriété que l'on souhaite vérifier. Le fait de restreindre les comportements de l'environnement permet de réduire fortement l'automate global obtenu après composition (synchronisation forte) du système avec ce dernier. Cette étape d'identification des comportements spécifiques constitue une première possibilité de réduction.

Sur cette idée, une méthode de description de l'environnement appelé **contexte** a été développée [Rog06, DB07, DRB07, DPC+09], pour permettre à l'utilisateur de spécifier, dans un langage simple, le comportement souhaité de son environnement. Un langage CDL (Context Description Language) [DBL08], présenté en détail dans la section 4.2 est intégré dans une méthodologie de vérification et un outillage nommé OBP (Observer Based Prover) [DPC+09].

1.3 La méthodologie et l'outillage OBP

Méthodologie. La méthodologie qui a été proposée a pour but de définir un cadre structurant, permettant à l'utilisateur de décrire formellement des *unités de preuve* [DB07], à l'aide du langage CDL de description de contextes. Les *unités de preuve* sont identifiées comme des modèles regroupant les données nécessaires à la preuve des propriétés. Elles regroupent essentiellement des modèles CDL décrivant le contexte et les exigences, et une référence au modèle à valider. CDL permet de spécifier des contextes sous la forme de scénarios, et des propriétés temporelles à l'aide de patrons de définition de propriété. De plus, CDL offre la possibilité à l'utilisateur d'associer chaque propriété à une phase du comportement de l'environnement du système, c'est-à-dire un contexte spécifique. Les unités de preuves sont ensuite exploitées pour générer automatiquement des programmes formels assimilables par des outils de vérification.

Lors du processus méthodologique, nous supposons que pour établir une vérification d'un ensemble des exigences, celles-ci peuvent être formalisées sous la forme de propriétés logiques (comportementales de nature fonctionnelle ou non fonctionnelle). Nous supposons également que l'environnement du modèle (le contexte) à valider ainsi que les interactions entre l'environnement et le modèle sont modélisés formellement. Enfin, en vue d'utiliser un vérificateur formel, le modèle de conception doit pouvoir être simulable. Les propriétés de l'environnement et le modèle simulable constituent les données pertinentes et suffisantes pour conduire les vérifications d'exigences sur le modèle. La méthode proposée inclut donc les phases suivantes représentées par la figure 1.1 :

- A partir d'un modèle de conception fourni par l'industriel, un modèle de conception formel est généré (manuellement ou semi-automatiquement) par une extraction des données utiles du modèle de conception. Ces données permettent d'obtenir un modèle comportemental formel simulable.

- A partir d'exigences fournies par le document d'exigences, des propriétés sont formalisées et un ou des modèles du comportement de l'environnement sont construits. Cette formalisation est réalisée à partir d'une interprétation d'exigences textuelles et d'une description de l'environnement du système modélisé. La rédaction d'exigences formelles et la modélisation du contexte sont d'autant plus aisées que, d'une part, la description du comportement de l'environnement est déjà formalisée (cas d'utilisation, scénarios, diagramme de séquence, etc. . .) et que, d'autre part, les exigences sont exprimées sans ambigüité. La formalisation des exigences s'effectue à l'aide de patrons de définition de propriétés [DPC+09].

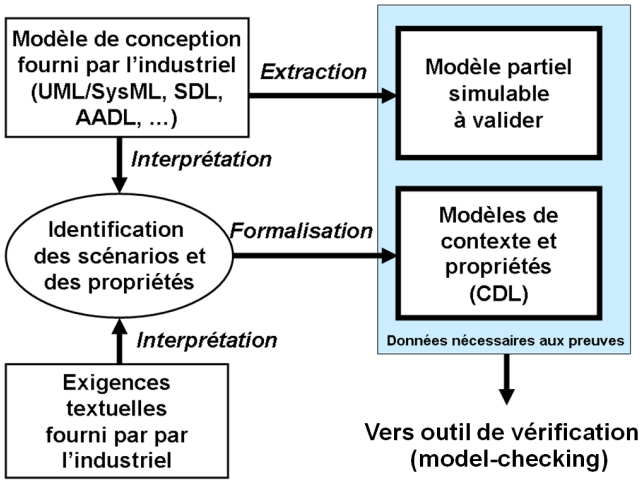


FIG. 1.1: Méthodologie OBP

Outillage. Les modèles CDL sont traduits dans l'outil OBP (figure 1.2), en codes assimilables par un vérificateur. Actuellement, OBP génère des programmes basés sur des automates temporisés au format **IF2** [BFG+99] en vue d'alimenter le simulateur IFx. Des développements sont en cours pour connecter OBP à d'autres *model-checker* tels que **TINA** [BBV04] ou **CADP** [FGK+96]. Nous considérons dans ce mémoire qu'OBP est connecté à SPIN. Les modèles CDL sont importés dans OBP dans un format textuel. OBP les interprète et génère des automates dans un format interne, les α-contexte proposés dans [Rog06], représentant le comportement de chaque entité de l'environnement interagissant avec le modèle soumis à la validation.

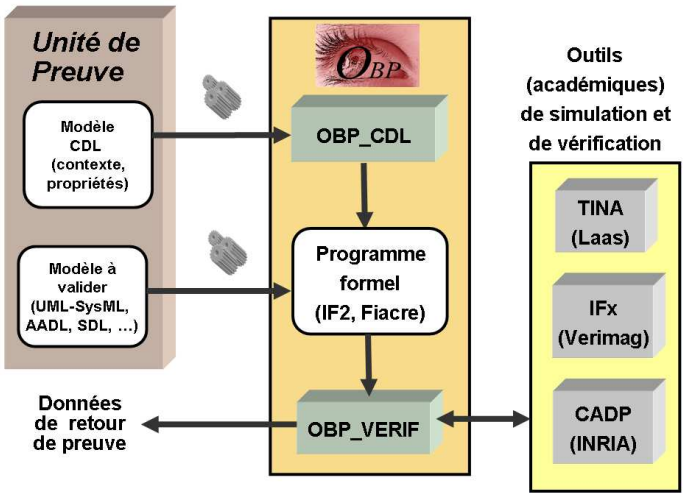


FIG. 1.2: Outil OBP

Diviser pour régner. Ces automates sont ensuite dépliés (passage d'une représentation abstraite à une représentation concrète du contexte sous la forme d'un graphe acyclique) et partitionnés (génération de l'ensemble des scénarios d'un graphe acyclique) pour produire des automates temporisés linéaires IF2 qui représentent l'ensemble des scénarios d'exécution de l'environnement. Ce sont ces scénarios qui sont composés avec les propriétés, elles-même transformées en automates

(observateurs) [DBL08] et le modèle à valider. Cette partition du contexte, en un ensemble de scénarios d'exécution permet d'aboutir, lors de la composition, à des graphes d'états de taille limitée rendant possible l'analyse d'accessibilité pour chaque scénario. Cette étape constitue une deuxième réduction de l'explosion combinatoire en espace.

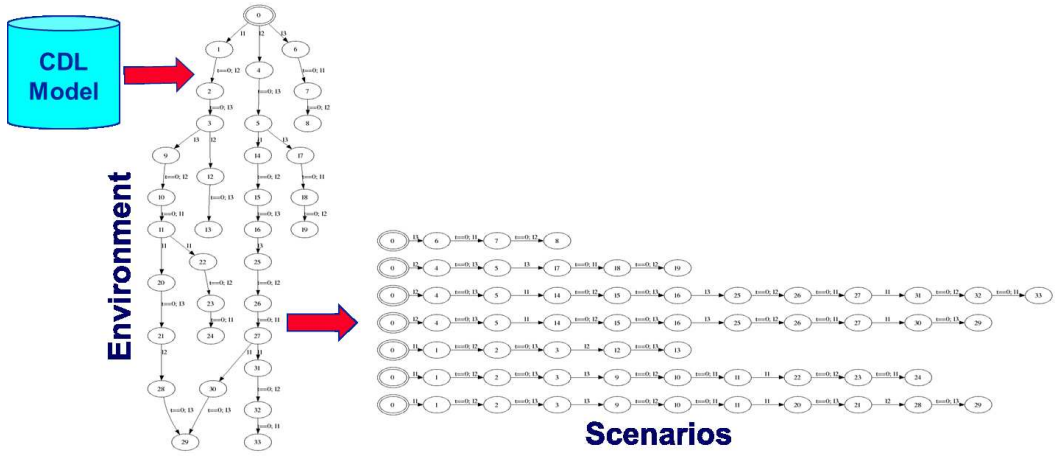


FIG. 1.3: Méthode diviser pour régner

Ainsi, le *model-checking* s'effectue séparément, en composant le système avec la propriété à vérifier ainsi qu'avec chaque scénario du contexte (représenté à droite de la figure 1.3), à la place du graphe global (représenté à gauche de la figure 1.3). Il a été démontré dans la thèse [Rog06] que la vérification d'un système composé avec son environnement global est équivalent à effectuer des vérifications séparées sur chaque scénario du contexte. Par conséquent, la vérification "brute" consistant à composer le système avec l'environnement global peut être décomposée en N *petites* vérifications.

1.4 Inconvénient de la méthode

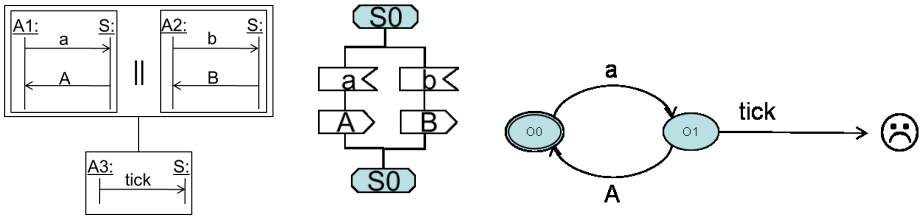
Dualité espace/temps. La description de l'environnement et la décomposition de la vérification, en vérification élémentaire sur chaque scénario, apporte une réduction importante de l'explosion combinatoire en espace. Néanmoins, dès lors que le nombre d'**acteurs** (c'est à dire des **éléments parallèles** du contexte) devient important, le nombre de scénarios dépliés peut devenir prohibitif (par exemple, un environnement composé de P acteurs concurrents chacun émettant un message au système, générera $P!$ scénarios), et par conséquent une explosion combinatoire en temps et non plus en espace. La complexité du processus de vérification n'a pas disparue mais a seulement changé de nature. L'explosion combinatoire qui était de nature spatiale devient, par cette méthode, de nature temporelle.

1.5 Contribution

Objectif. Pour pouvoir contrer l'explosion combinatoire en temps, les travaux de ce mémoire viennent s'interfacer avec la méthodologie OBP de manière à réduire le nombre de scénarios à prendre en compte pour la vérification. En effet, parmi les N scénarios de l'environnement, beaucoup d'entre eux peuvent être équivalents par la nature même d'un système critique qui se doit souvent d'être robuste à la nature asynchrone des acteurs et de son environnement. En effet, de part la complexité intrinsèque des systèmes industriels, et de part les communications le plus souvent asynchrones, les ingénieurs ne possèdent pas une cartographie complète de la dynamique de l'environnement de leur système ; c'est à dire, qu'ils ne peuvent pas déterminer en amont l'ordre d'émission ou de réception de chaque message entre l'environnement et le système. Par conséquent,

le système est modélisé de manière à accepter plusieurs entrelacements possibles des acteurs de son environnement (par exemple par la sauvegarde de messages dans le cas du langage SDL) rendant ainsi équivalents de nombreux stimulis de l'environnement.

Méthode proposée. Pour profiter du mécanisme de robustesse des systèmes critiques, la méthode proposée dans ce mémoire consiste à réduire les scénarios de l'environnement par la détermination d'un entier $N' < N$ *avec* $N' \in \mathbb{N}$ le nombre de scénarios réduits de telle manière que ces N' scénarios suffisent à couvrir la totalité des comportements des N scénarios initiaux. Pour cela, nous nous appuyons sur des méthodes de factorisation des ordres-partiels. L'équivalence appelée relation d'indépendance entre les scénarios de l'environnement est déterminée en "testant" la robustesse du système et de la propriété à vérifier vis-à-vis de ces derniers. Pour illustrer cette approche, considérons l'exemple du système S de la figure 1.4.



(a) Contexte du modèle SDL (b) Exemple de modèle SDL (c) Exemple d'un observateur : Obs_1

FIG. 1.4: Exemple préliminaire

Supposons (figure 1.4a) un système S en interaction avec 3 acteurs A_1, A_2 et A_3. L'acteur A_1 (resp. A_2) peut envoyer a (resp. b) à S et recevoir A (resp. B) de S. Le dernier acteur A_3 envoie le message *tick* au système après les exécutions de A_1 et A_2 (qui sont indépendants).

Sur la figure 1.4b est représentée le modèle SDL [SDL92] décrivant le comportement du système S. Ce modèle est un automate composé d'un seul état S_0 et de deux transitions. Dans l'état S_0, le système peut recevoir les messages a ou b.

- Lors de la réception du message a, le message A est envoyé à l'environnement. Le système retourne alors dans l'état de départ S_0 dans l'attente d'un nouveau message.

- Lors de la réception du message b le message B est envoyé à l'environnement. Le système retourne alors dans l'état de départ S_0 dans l'attente d'un nouveau message.

Supposons que l'on veuille vérifier la propriété encodée par l'observateur figure 1.4c. Cet observateur vérifie que lorsque le message a est envoyé par l'acteur A_1 au système S, alors l'acteur A_3 ne pourra envoyer le message *tick* qu'après la réception du message A envoyé par le système à l'acteur A_1. On observe sur cet exemple que l'ordre d'envoi des messages a et b au système ne modifie pas le comportement de ce dernier. En effet, considérons les scénarios suivants :

- Premier scénario : $a \cdot b \cdot tick$ envoie en séquence le message a suivi du message b suivi du message *tick* au système. Dans ce cas, à partir de l'état initial S_0, le message a est consommé, le message A est envoyé à l'environnement avant de revenir dans l'état S_0. A partir de cet état, le deuxième message b est consommé, et le message B est reçu par l'environnement. Finalement, le système revient dans l'état S_0 dans lequel le message *tick* est "discardé" (perdu car consommé implicitement).

- Deuxième scénario : $b \cdot a \cdot tick$ envoie en séquence le message b suivi du message a et du message *tick* au système. Dans ce cas, à partir de l'état initial S_0, le message b est consommé, le message B est envoyé à l'environnement avant de revenir dans l'état S_0. A partir de cet état, le deuxième message a est consommé, et le message A est reçu par l'environnement. Le

message *tick* est ici également "discardé". Finalement le système revient dans l'état S_0.

A l'issue de la simulation de ces deux scénarios, nous remarquons que le système atteint un même état final S_0. Par conséquent, l'ordre des messages a et b ne modifie pas le comportement du système S qui est donc robuste à l'entrelacement de ces deux messages. De plus, les productions A et B émises par le système n'influent aucunement sur le comportement de l'environnement, car chacun de ces deux messages est reçu par des acteurs concurrents. Enfin, les messages A et B ne modifient pas le comportement de la propriété en cours de vérification représentée par la figure 1.4c.

- si l'on exécute le scénario $b \cdot a$; dans l'état O_0 si l'on consomme b en premier, le message est perdu car il n'est pas attendu. Par conséquent l'observateur ne change pas d'état et reste dans O_0. Lorsque a est exécuté, l'observateur passe de l'état O_0 à O_1.

- Maintenant avec le scénario $a \cdot b$; dans l'état O_0, l'observateur détecte le message a et passe dans l'état O_1. Dans O_1, b ne peut pas être consommé. Par conséquent, l'état final atteint est O_1.

Par conséquent, les messages a et b sont dits indépendants. L'entrelacement des messages a et b n'a aucune influence sur la propriété à vérifier. Par conséquent, les messages a et b sont dits "indépendants" dans le sens où leur ordre de consommation dans le système est sans importance, et que l'exécution de l'un n'empêche pas l'exécution de l'autre. Il en résulte que les scénarios de départ $a \cdot b \cdot tick$ et $b \cdot a \cdot tick$ sont trace-équivalents par la relation d'indépendance (équivalence) entre a et b. Il en découle qu'un seul des deux scénarios est suffisant pour la vérification de la propriété sur le système.
A présent considérons, l'observateur de la figure 1.5.

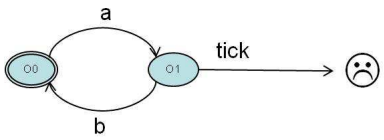


FIG. 1.5: Dépendance au niveau de l'observateur

- si l'on exécute le scénario $b \cdot a$; dans l'état O_0 si l'on consomme b en premier, le message est perdu car il n'est pas attendu. Par conséquent l'observateur ne change pas d'état et reste dans O_0. Lorsque a est exécuté, l'observateur passe de l'état O_0 à O_1.

- Maintenant avec le scénario $a \cdot b$; dans l'état O_0, l'observateur détecte le message a et passe dans l'état O_1. Dans O_1, b peut être consommé. Par conséquent, l'état final atteint est O_0.

Les deux scénarios amènent donc à des états de l'observateur différents et ainsi l'ordre de a et b devient important pour la vérification : a et b sont donc dépendants pour l'observateur.

Dans la suite de ce mémoire, il s'agit de déterminer parmi les événements contenus dans l'environnement, ceux qui sont "indépendants" et construire ainsi la plus grande relation d'indépendance entre les événements. Cette relation permet par la suite de calculer des équivalences entre les scénarios générés par l'outil OBP par l'application de la théorie des traces de Mazurkiewvicz [Maz86] et par la construction des formes normales de Foata [CF69]. L'approche est résumée sur la figure 1.6.

1.6 Plan de la thèse

Le mémoire s'articule de la manière suivante : dans un premier temps un panorama des méthodes de vérification existantes est exposé en commençant par la technique du *model-checking* dans le chapitre 2. Le chapitre 3 est consacré à la présentation des ordres-partiels, ainsi qu'à la description des principaux algorithmes de calcul d'équivalence de séquences d'événements issus de la littérature.

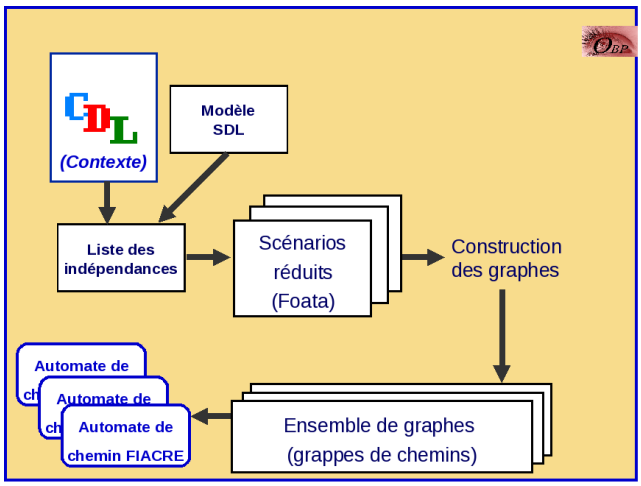


FIG. 1.6: Schéma de réduction

Par la suite, un comparatif entre le *model-checker* SPIN et la méthode présentée dans ce mémoire est effectué. L'objectif de ce mémoire est d'étendre les techniques des ordres-partiels au niveau spécification et d'appliquer ces techniques sur une spécification composée de trois entités : le modèle, l'environnement et la propriété à vérifier sans au préalable avoir besoin de construire l'automate global. Le chapitre 4 introduit les formalisations d'un sous-ensemble du langage SDL pour modéliser les systèmes et CDL pour la modélisation de l'environnement. Le chapitre 5 définit formellement les relations d'indépendance des événements du contexte dans le cas d'un système composé d'un seul processus. Le chapitre 6, étend ces définitions au cas multi-processus. Enfin, la méthode formalisée est appliquée à deux cas d'étude industriels dans le chapitre 7. Dans le chapitre 8, un bilan est dressé sur les performances obtenues par la méthode exposée dans ce mémoire qui est comparée avec les performances du *model-checker* SPIN. Enfin le chapitre 8.2.3 achève le mémoire par les perspectives envisagées et les conclusions des travaux exposés.

1.7 Guide de lecture

Le lecteur ayant des connaissances sur les méthodes formelles pourra directement aborder la section 2.3 qui énumère les principales avancées des techniques des ordres-partiels. Le lecteur habitué aux méthodes des ordres-partiels pourra commencer la lecture du chapitre 4 contenant des notations formelles nécessaires pour la compréhension des chapitres 5 et 6.

Chapitre 2

Les bases théoriques et état de l'art

Sommaire

Résumé. Ce chapitre présente un panorama des méthodes formelles de vérification issues de ces 25 dernières années en partant de la méthode mère : le *model-checking*. Le lecteur intéressé par un historique complet pourra se reporter à l'ouvrage [GV08]. Les principales méthodes qui en découlent ainsi que les enchaînements entre elles sont ensuite brièvement introduites. En particulier, les méthodes de réduction par **ordres-partiels** sont présentées.

2.1 Le *model-checking*

2.1.1 Historique.

Avec l'arrivée des premiers ordinateurs et des premiers programmes séquentiels déterministes, des études sur la vérification ont débuté avec des techniques de preuves manuelles de systèmes séquentiels. La logique de Hoare [Hoa69] pour la correction et la terminaison de programmes a vu le jour, sous la forme de règles d'inférences permettant de vérifier un programme séquentiel. La technique proposée était innovante, élégante et "compositionnelle" (la correction d'un programme est déterminée par la correction de ses composants).

Très vite la question s'est posée de la vérification de systèmes concurrents. Amir Pnuelli en 1977 [Pnu77] a le premier introduit la logique temporelle linéaire permettant de décrire le comportement d'un système concurrent dans le temps de manière "intemporelle" (qualitativement) et surtout de manière formelle. Ses travaux permettent d'unifier la spécification formelle des systèmes séquentiels et concurrents dans un même formalisme.

La vérification de programmes concurrents était néanmoins toujours effectuée manuellement dans les travaux de Pnuelli. En 1978 le langage CSP [Hoa85], algèbre de processus permettant de décrire formellement les systèmes concurrents a été introduite par Hoare. En 1980 et 1981, Emerson et Clarke ont introduit la logique temporelle arborescente CTL[1] (qui pouvait être caractérisée par des points fixes) et un algorithme de vérification automatique basé sur les points fixes qu'ils ont nommé *model-checking* [CE82]. Ils ont ainsi développé le premier *model-checker* baptisé EMC. En 1982, Sifakis et Queille ont aussi introduit la notion de *model-checking* [QS82] indépendamment des travaux effectués par Clarke et Emerson.

Par la suite, en 1986, le *model-checking* a été étudié pour LTL[2] [MP92a], et un algorithme de vérification basé sur la méthode des tableaux a été développé. Il s'agit en effet de transformer une formule LTL en automate de Büchi par décomposition de la formule en sous-formules en forme normale conjonctive. Se sont ensuivis des travaux sur des logiques plus expressives telles que CTL* [EH83] partant du constat que la logique LTL ne pouvait exprimer certaines formules CTL et réciproquement. Enfin, une logique plus expressive : la logique modale μ-calcul [Koz82] s'est vue décrite comme une unification des logiques temporelles, et lui a valu le nom de "théorie du tout".

2.1.2 Principes du *model-checking*

Etant donné un système réel et une exigence de la spécification, le *model-checking* s'intéresse au problème de savoir si le système satisfait bien à l'exigence. La vérification automatique par *model-checking* s'effectue en trois étapes illustrées sur la figure 2.1 :

1. Etape 1 : Modélisation du système réel en modèle formel,
2. Etape 2 : Formalisation de l'exigence (écrite en langage naturel) en logique temporelle,
3. Etape 3 : Vérification par un algorithme.

Modélisation de systèmes complexes. Les systèmes réels étant complexes, une manière d'y faire face consiste à décomposer le système en plusieurs sous-systèmes. C'est à dire représenter le comportement du système global en plusieurs automates concurrents. Ensuite seulement, il s'agit de reconstruire le comportement du système complet en effectuant un produit synchronisé [Niv79, AN82], afin d'obtenir la dynamique du système global. Dans le cas des systèmes complexes, ce produit synchronisé peut générer une explosion combinatoire.

[1] Computational Tree Logic
[2] Linear Temporal Logic

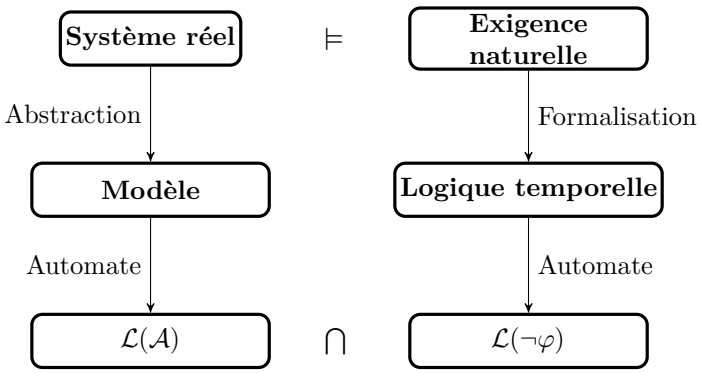


FIG. 2.1: Etapes du *model-checking*

Remarque. Il est à noter que la méthode du *model-checking* commence par effectuer une abstraction du système réel. Par conséquent une erreur du système réel ne peut être détectée dans le système, seulement si elle a été modélisée. Cette méthode à l'avantage d'être **exhaustive**, et permet donc de mettre en évidence la **totalité** des erreurs d'un système. De plus, un contre-exemple peut être généré lorsque la propriété à vérifier est falsifiée.

Etape 1 : modélisation du système par des Structure de Kripke. Les systèmes sont en général modélisés sous forme d'automates. Toutefois, le *model-checking* consiste à vérifier des propriétés sur les états. Pour cela un modèle simplifié basé sur les propositions atomiques est utilisé : la structures de Kripke. Une structure de Kripke est un automate particulier composé par un sextuplet $\mathcal{K} = \langle \Sigma, A, T, I, AP, l \rangle$ où :

- Σ représente l'ensemble des états du système
- A représente l'ensemble des actions
- $Tr \subseteq \Sigma \times A \times \Sigma$ représente l'ensemble des transitions
- $I \subseteq \Sigma$ contient l'ensemble des états initiaux
- AP est l'ensemble des propositions atomiques (c'est à dire des énoncés prenant les valeurs vraies ou fausses)
- $l : S \rightarrow 2^{AP}$ est la fonction d'étiquetage qui associe à un état des propositions atomiques

Cette représentation permet de vérifier simplement des propriétés sur les états du système considéré.

Exemple. L'automate de la figure 2.2 représente un système de digicode qui garantit l'accès uniquement sur la suite d'actions ABA. La structure de Kripke $\mathcal{K}_1$ de ce système est définie par $\mathcal{K}_1 = \langle \Sigma_1, A_1, Tr_1, I_1, AP_1, l_1 \rangle$:

- $\Sigma_1 = \{S_0, S_1, S_2, S_3\}$
- $A_1 = \{A, B, C\}$
- Tr_1 est l'ensemble des 9 transitions
- $I_1 = \{S_0\}$
- $AP_1 = \{OPEN\}$
- $l_1(S_0) = \{OPEN\}$

La structure de Kripke est bien adaptée pour la vérification de propriétés d'états. Sur cet exemple, le nombre d'états et de transitions sont finis mais le système peut potentiellement avoir un comportement infini.

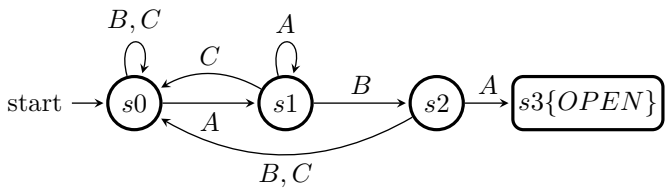


FIG. 2.2: Structure de Kripke du digicode

Etape 2 : formalisation d'exigences avec les logiques temporelles. Même si nous travaillons sur des systèmes à états finis, nous cherchons souvent à vérifier des propriétés sur des exécutions potentiellement infinis, puisque l'on souhaite que la propriété reste vraie quel que soit le comportement du système.

De nombreuses logiques temporelles ont été développées comme citées précédemment. Parmi ces dernières trois fondamentales se distinguent : la logique linéaire (LTL) décrivant des propriétés sur des scénarios d'exécution, la logique arborescente (CTL) et la logique modale (μ-calcul) qui est la logique la plus expressive. Bien entendu certaines propriétés peuvent s'exprimer à l'aide de quantificateurs avec la logique de premier ordre qui est très expressive. Le développement de divers langages de logique temporelle permet toutefois d'utiliser une syntaxe simple et l'expressivité adéquate aux propriétés que l'on souhaite vérifier. Par exemple, la logique LTL permet d'exprimer des propriétés sur des traces d'exécution. La logique CTL permet de formuler des propriétés sur un ensemble d'embranchement de traces. Par contre, la logique LTL ne peut exprimer certaines propriétés telle que : il y aura un nombre pair d'événement a. La logique CTL ne peut exprimer l'équité. Ainsi les logiques temporelles permettent d'exprimer un large panorama de propriétés parmi lesquelles : l'accessibilité, l'invariance, la sûreté, la vivacité, l'équité.

Remarque : dualité fini/infini. Pour pouvoir vérifier des comportements infinis (les propriétés) sur des systèmes finis (structure de Kripke) la méthode consiste alors à calculer les composantes fortement connexes maximales. En effet lorsqu'une composante fortement connexe est atteinte, alors, une infinité de fois, un même comportement peut s'effectuer. Dans la suite, les principaux algorithmes existant pour les logiques temporelles LTL, CTL et le μ-calcul sont brièvement présentés. Le lecteur intéressé par les algorithmes pour d'autres logiques temporelles telles que CTL^* pourra consulter l'ouvrage [CGP00].

Etape 3 : algorithme de vérification. Il existe deux principales questions concernant la vérification :

- Le *model-checking* qui cherche à démontrer si un système satisfait la propriété donnée
- La satisfiabilité qui cherche à démontrer s'il existe un modèle permettant de satisfaire la propriété

Ces deux problèmes sont complémentaires. La satisfiabilité est très importante pour déterminer la sémantique des langages temporels et leur domaine d'expressivité et de validité. A partir de là, étant donné un automate et une propriété, il suffit de savoir si le modèle en question fait parti des modèles permettant de satisfaire la propriété, pour obtenir le diagnostic final.

On présente maintenant les principes utilisés pour la vérification de la logique linéaire et la logique arborescente . Comme énoncé précédemment, l'objectif est de vérifier des propriétés sur des **exécutions infinies** du **système fini**. Pour cela, la méthode classique consiste à associer le calcul des composantes fortement connexes avec une analyse d'accessibilité.

Pour le cas de la logique LTL, il s'agit de récrire la formule à l'aide de la méthode dite "du tableau" de manière à obtenir un automate de **Büchi**, en décomposant successivement la formule

LTL, en sous-formules.

Pour le cas de la logique CTL, la méthode dite du marquage est utilisée : elle consiste à décomposer la formule générale en sous-formules. Pour chaque sous-formule, il s'agit de marquer les états qui la vérifie. Ce procédé est réitéré sur un ensemble de sous-formules. Au final les sous-formules représentent la formule globale et la méthode du marquage se termine. Cette méthode permet de vérifier des propriétés écrites sous forme de point fixe [Tar55].

Obstacle au *model-checking*. Bien qu'élégante, la méthode du *model-checking* se confronte à un problème fondamental : le problème de l'arrêt qui découle de la théorie du calcul et de la complexité [Sip96], qui s'adonne à la question suivante : "qu'est ce qui est calculable efficacement et qu'est ce qui ne l'est pas ?" **indépendamment de toute technologie**. Pour cela if faut remonter dans les années 40 et les travaux innovants de **Alan Turing** (avec la célèbre machine à penser), **Alonzo Church** et **Kurt Gödel**. Le problème peut être défini de la manière suivante : étant donnée une machine de Turing [Tur37] (qui est un modèle de calcul[3] permettant de définir la sémantique d'exécution d'un programme) avec une bande **infinie**, et la propriété : est ce que la machine termine un jour (sur un espace blanc par exemple représenté par le caractère 'b') , il a été démontré que ce problème connu sous le nom de problème de l'arrêt ou encore "the halting problem" est indécidable ; c'est à dire qu'il n'existe pas de fonction caractéristique répondant systématiquement par "vrai" ou "faux" excepté pour les cas triviaux. Cela est le fruit du théorème de Rice. De tels théorèmes mettent à néant les espoirs de vérification automatique. De manière générale, vérifier si un programme fait exactement ce que l'on souhaite est un problème indécidable. Néanmoins, ce problème concerne les machines possiblement infinies. Le fait de se restreindre à des machines moins expressives avec les structures de Kripke permettent d'introduire la vérification automatique, en ne prenant en compte, que des machines à états finis.

En résumé, étant donné un programme, il n'existe aucun programme permettant de décider systématiquement dans des délais raisonnables si un programme boucle ou non, quelque soit la puissance de calcul de l'ordinateur. Même l'accessibilité d'un état de contrôle n'est pas décidable. La machine ne pourra donc pas remplacer totalement l'activité humaine. Toutefois, dans le cas où un problème reste décidable, le phénomène de l'explosion combinatoire empêche une vérification exhaustive du programme.

Pour pouvoir dépasser ces limites, il faut faire une abstraction du modèle de conception en prenant des modèles moins réalistes (d'où le nom *model-checking*) par exemples en éliminant des informations inutiles : les modèles finis, dans lesquels les états et les transitions du système sont énumérables.

L'expression des propriétés. Un autre point délicat lors de l'application du *model-checking*, est la nécessité d'exprimer la propriété à vérifier dans une logique temporelle telle que LTL ou CTL, qui reste une étape délicate et demandant beaucoup d'expertise. C'est pourquoi la méthodologie proposée par l'ENSIETA consiste à se restreindre à des classes d'accessibilité plus faciles à démontrer, et qui peuvent facilement être modélisées sous formes d'observateurs. La vérification par les observateurs se ramène ainsi à vérifier, si un état correct est atteint un jour (état de succès) : $\diamond happy$, ou bien si un état non désiré n'est jamais atteint (état de rejet) : $\Box(\neg unhappy)$. Dans la suite on se restreint à cette classe de propriétés.

2.1.3 Discussion sur le *model-checking*

La méthode générale du *model-checking* qui consiste à explorer l'ensemble des états du système global après composition de ses sous-composants, présente l'avantage d'être exhaustive ; en contrepartie, elle n'est vraiment efficace que pour certains systèmes dont la complexité est raisonnable. Dans le cas de systèmes complexes, une explosion combinatoire est obtenue sur le nombre d'états.

[3] La RAM (*Random Access Memory*) est un autre exemple de modèle de calcul

Or, les systèmes embarqués mettent en œuvre l'asynchronisme, la concurrence et le non déterminisme (résultant de l'asynchronisme de cette concurrence). Ces critères augmentent d'autant plus les possibilités combinatoires, par les nombreux entrelacements engendrés par les événements des processus concurrents.

Le *model-checking* dans le milieu industriel. Dans le cas du génie logiciel, le *model-checking* est appliqué sur des "modèles" de conception (par exemple une spécification SDL) à partir desquels le code est généré par des outils certifiés, préservant les propriétés à vérifier. Ce mémoire se positionne au niveau de la conception détaillée (figure 2.3) et prend comme point de départ, les modèles de conception réalisés dans le langage SDL par l'équipe de spécification formelle de l'entreprise CS-SI.

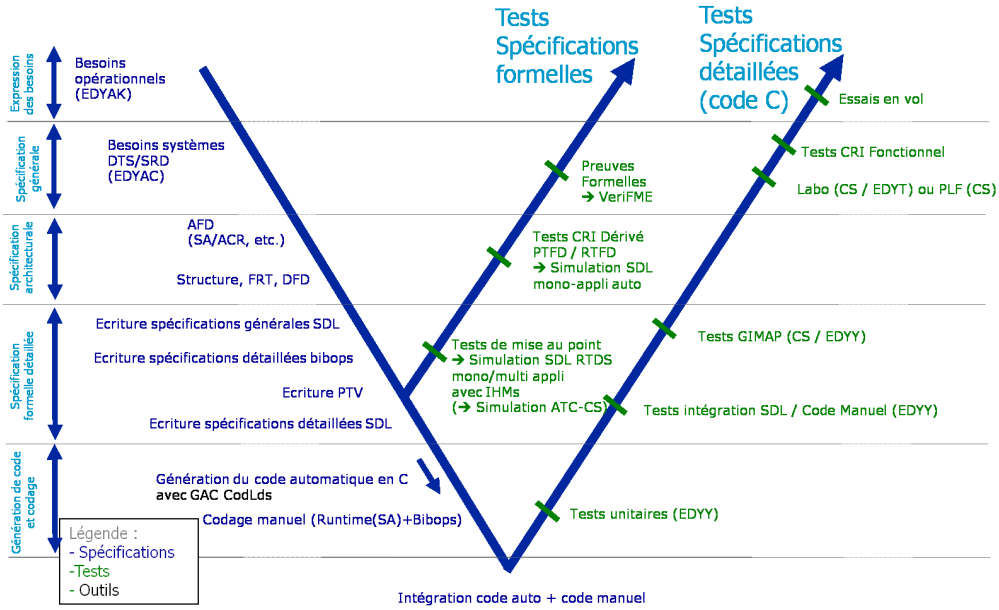


FIG. 2.3: Positionnement du mémoire dans le cycle en V

Néanmoins, le *model-checking* atteint rapidement ses limites, lorsque l'on veut valider des systèmes de taille industrielle. Pour pallier à cette contrainte, de nouvelles techniques basées sur les avancées du *model-checking* ont été réalisées, en particulier le *model-checking* "à la volée", les méthodes d'abstraction et les Diagrammes de Décisions Binaires : les BDDs [Bry92].

2.2 Au-delà du *model-checking*.

Partant du constat que les méthodes classiques sont insuffisantes pour s'opposer à l'explosion combinatoire sur des systèmes volumineux, de nombreuses recherches ont été menées sur le moyen de réduire le système, afin de n'en garder que l'essence même de son comportement, suffisante pour la vérification de propriétés.

2.2.1 Réduction d'automate à l'aide d'interface de processus

Dans [GS90], une méthode de composition "générationnelle" est proposée, pour réduire de manière incrémentale le système d'automate global, sur des systèmes communiquant par rendez-vous. Le principe consiste à réduire chaque automate (un processus) modulo une relation d'équivalence, et ensuite effectuer la composition en prenant en compte son environnement (les processus communicant avec ce dernier), pour ainsi déterminer les états inaccessibles et aberrants à éliminer. L'environnement du processus est décrit par des processus appelés **interfaces**. La difficulté de cette

approche est l'obligation à l'utilisateur de spécifier les interfaces à contraintes manuellement. La génération des environnements a été rendue automatique dans les travaux de [CK96]. Cependant, l'algorithme de création de l'environnement présente le défaut de rester souvent sur-approximative, et de ne pas permettre d'éliminer certains états aberrants ou non atteignables.

2.2.2 Abstraction de modèles

Le *model-checking* nécessite une première abstraction du modèle de conception en éliminant les informations inutiles pour la vérification. Ce processus peut être raffiné, pour obtenir des modèles beaucoup plus simples. Les techniques d'abstractions sont nombreuses et variées, et leurs méthodes sont dépendantes du langage de spécification, de modélisation, du type du système à vérifier, ainsi que des propriétés que l'on veut étudier (sûreté, vivacité).

L'idée consiste à s'abstraire et donc de supprimer tous les détails (généralement les données et leur domaine de valeur) inutiles ou non pertinentes pour la vérification du système [CGL94, Yus04, Das03, BGP+02, GK97]. La difficulté consiste alors à trouver l'abstraction la plus juste possible possédant juste assez d'informations pour la vérification du système. Néanmoins, le modèle abstrait possède plus de comportements que le modèle concret (modèle initial). En effet, il existe une relation de simulation entre le modèle abstrait et le modèle concret mais pas l'inverse. Ainsi, si une propriété est vérifiée sur le modèle abstrait alors forcément elle sera vérifiée sur le modèle concret. En revanche, une propriété falsifiée dans le modèle abstrait, ne le sera pas automatiquement dans le modèle concret, ce qui peut engendrer des faux contre-exemples, qui doivent être vérifiés et testés sur le modèle concret. Dans [CFH+03, KGC04] la méthode d'abstraction proposée utilise ces contres-exemples pour raffiner l'abstraction. Ainsi, trouver une bonne abstraction est un travail délicat et demande beaucoup de créativité de la part de l'ingénieur, car non générique.

2.2.3 *Model-checking* Symbolique

Dualité représentation implicite/explicite d'états. Pour faire face à la complexité de systèmes industriels, des méthodes de compressions et de factorisations des états ont été étudiées. Il s'agit de méthodes symboliques. Elles ont vu le jour avec les travaux de [Bry92, McM92]. La technique manipulant les BDDs (Diagrammes de Décisions Binaires) fait partie de la grande famille du *model-checking* symbolique. L'idée sous-jacente part du constat que tout programme peut être manipulé en tant que suite de booléens. La méthode des BDDs consiste à ne pas représenter en extension les ensembles d'états (représentation **explicite**), mais en faire une représentation symbolique (représentation **implicite**). De manière générale, cette méthode est utilisée sur les systèmes composés de circuits logiques. Il s'agit en effet de représenter les contraintes du système sous la forme d'un ensemble de formules booléennes, et appliquer des méthodes de simplification permettant d'arriver dans une solution unique (forme normale conjonctive), où toutes les contraintes booléennes sont satisfaites. Du fait de la combinatoire possible sur les différentes variables booléennes, des règles de simplification des formules ont été développées. En contre-partie, sur des systèmes volumineux de très nombreuses variables booléennes doivent être manipulées. De plus l'ordonnancement de ces variables pour le calcul de la Forme Normale Conjonctive (CNF), peut totalement modifier l'efficacité de la technique, rendant nécessaire l'utilisation d'heuristiques.

2.3 Les ordres-partiels

Dans la continuité des travaux sur les méthodes de réduction de l'explosion combinatoire, une vague de recherche a vu le jour dans les années 90. A la suite des travaux de la théorie des traces de Mazurkiewicz [Maz86], basés sur la notion des ordres-partiels. Contrairement aux méthodes symboliques, il s'agit ici de représenter explicitement les états de l'automate global au fur et à mesure de sa construction (à la volée) ; mais, à l'opposé du *model-checking* où tous les états sont explicitement construits, les méthodes par ordres-partiels réduisent le nombre d'états et de transitions à explorer en profitant des symétries présents dans le système.

Souvent, les entrelacements des transitions du système génèrent des scénarios redondants symétriques. Dans le cas où certaines actions (transitions) sont indépendantes les unes aux autres, leur

ordre d'apparition ne modifie pas le comportement du système. Si deux séquences distinctes ne sont pas différenciées par une formule LTL, alors elles peuvent être considérées comme équivalentes d'un point de vue comportemental. C'est pourquoi, les méthodes par ordres-partiels permettent de supprimer ces redondances en ne conservant qu'un seul ordonnancement représentatif.

Dualité entrelacement/concurrence vraie. On distingue deux principales sémantiques de la concurrence : la sémantique par entrelacement et la sémantique de la concurrence vraie.

- La sémantique par entrelacement considère que les événements en parallèles s'exécutent toujours les uns à la suite des autres ; c'est à dire que la simultanéité est impossible. C'est par exemple ce procédé qui est utilisé par les "ordonnanceurs" des systèmes d'exploitation mono-processeur. Le parallélisme est simulé par l'ordonnanceur. Dans les méthodes par ordres-partiels c'est cette sémantique qui est utilisée, puisque l'on souhaite tirer avantage des séquences de transitions équivalentes ; c'est à dire dont l'ordonnancement des transitions amènera le système dans le même état global.

- La deuxième sémantique est celle de la concurrence vraie. Dans ce cas, les événements peuvent réellement arriver simultanément. Cette fois si ce comportement pourrait être simulé par des machines multi-processeurs.

Dans l'article [TA04], il est toutefois remarqué que l'on peut toujours se ramener à une sémantique par entrelacement en raffinant la granularité du système. Dans la suite, le mémoire se situe dans le cadre des systèmes asynchrones, et considère la sémantique par entrelacements, dans l'objectif d'appliquer des méthodes par ordres-partiels (qui suivent la sémantique d'entrelacement) et ainsi déterminer une relation d'équivalence entre les scénarios.

Discrimination de transitions. Lorsque l'on souhaite générer l'automate global par composition de processus concurrents, la sémantique de l'entrelacement n'impose aucun ordre sur les événements, de telle manière à être le moins discriminant possible. Ces entrelacements génèrent ainsi une explosion combinatoire sur le nombre d'états et transitions, qui sont maintenus pour la vérification. Or, dans la grande majorité des cas, les entrelacements de ces événements sont insensibles à la propriété en cours de vérification, c'est à dire que la valeur de vérité de la propriété ne sera pas modifiée uniquement par une très faible minorité d'événements. Cela a pour conséquence que de très nombreuses séquences seront équivalentes pour la vérification de la propriété en cours. Pour tirer avantage de cette caractéristique et ainsi réduire l'explosion combinatoire, les méthodes par ordres-partiels se sont développées profitant de la propriété dite du "diamant".

2.3.1 Propriété du diamant

Il s'agit d'améliorer la technique du *model-checking* en construisant l'automate global à la volée et en éliminant les symétries ; c'est à dire des transitions qui peuvent être commutées et qui atteignent un même état global sans changer le comportement du système et la véracité de la propriété. C'est ce que l'on appelle la propriété du "losange" ou propriété du "diamant".

Exemple. Considérons l'automate représenté sur la figure 2.4. L'entrelacement des messages A et C mènent dans le même état s_4. Les deux séquences de messages $A.C$ et $C.A$ sont donc symétriques et une seule est nécessaire pour la vérification. Sur la figure 2.5 ; la séquence $A.C$ est suffisante pour représenter l'automate (losange) initial. Les transitions en pointillés sont éliminés ainsi que l'état s_2. Cette propriété permet ainsi d'alléger la méthode du *model-checking* en allégeant le nombre d'états stockés en mémoire. Ce mémoire exploite ces symétries pour la vérification de systèmes SDL contraints par un environnement CDL.

Les ordres-partiels appliqués aux systèmes SDL. Depuis l'émergence du *model-checking* en 1981, de nombreux langages et des outils de vérification se sont développés. Les méthodes par ordres-partiels sur des modèles SDL ont été étudiés dans [KLM+98, TGH95] .En particulier le

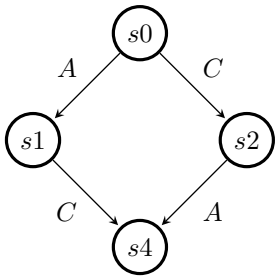


FIG. 2.4: Automate initial

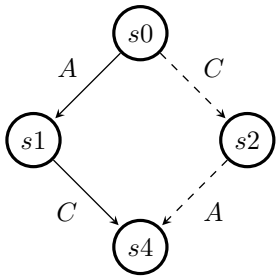


FIG. 2.5: Automate réduit

model-checker SPIN [Hol97] basé sur le langage Promela a été créé par G. J. Holzmann. L'outil permet la vérification de propriétés LTL encodées sous forme de "never claim" qui est ensuite transformée en automate de Büchi. L'outil a connu de très nombreuses optimisations prenant en compte l'ensemble des méthodes présentées ci-dessus (BDD, abstractions, ordres-partiels, etc...). Dans [BH05] par exemple, l'algorithme de parcours en profondeur généralement utilisé pour l'application des ordres-partiels, est converti en algorithme de parcours en largeur de manière à pourvoir appliquer des méthodes symboliques. En réalité le parcours en largeur est très efficace pour générer le plus petit contre exemple pour la vérification en cours. Néanmoins, les réductions obtenues restent sensiblement les mêmes que pour l'algorithme de recherche classique.

Avec l'arrivée des langages de spécification pour les systèmes embarques tel que langage SDL, des travaux sur la vérification de modèles SDL avec le *model-checker* SPIN ont été développés. SPIN ne supportant que le langage Promela, une transformation de langage a du être implanté. La méthode consiste à transformer le langage SDL vers un langage formel pivot : IF [BFG+99] (*Interchange Format*) avec l'outil sdl2if, puis à nouveau de transformer le système IF en système Promela avec l'outil if2pml. Néanmoins Promela ne supporte pas tous les concepts de SDL tels que les sauvegardes. La transformation des sauvegardes SDL en Promela a été étudiée dans [PCDR02]. Dans [Tuo99] les timers SDL sont approximés et représentés par des clauses particulières SET et RESET des timers SDL. Dans cet article seul l'aspect qualitatif des timers est prise en compte (les valeurs des horloges ne sont pas prises en comptes). Dans [BDHS00] Promela est étendu en Promela DT (*Discrete Time*) et SPIN par SPIN DT pour pouvoir prendre en compte cette fois-ci l'aspect quantitatif du temps malgré les différentes optimisations apportées au *model-checker* SPIN(compression des état, *bit-state-hashing* [Hol98] (supertrace)) ce dernier ne permet pas d'appliquer le *model-checking* sur des systèmes de taille industrielle.

2.3.2 Au-delà des ordres-partiels

Cette partie présente les principaux travaux effectués dans le domaine des ordres-partiels. L'idée développée dans cette théorie est basée sur la sémantique de la concurrence par entrelacement. En effet deux événements concurrents exécutés par deux processus différents sont entrelacés de manière à obtenir l'ensemble de leurs traces d'exécutions. Les méthodes par ordres-partiels consistent ainsi à ne parcourir que partiellement les différents jeu d'entrelacement possible par le biais d'équivalence entre les permutations d'événements qui sont dits indépendants.

De très nombreuses recherches ont été effectuées dans ce domaine [Pel98, GPS96, Ove81, Var98, KM00, CGMP99, BB01, MP92b, ERV96]. La première grande approche des entrelacements dans les ordres-partiels est mise en exergue dans la théorie des ***stubborn set*** développée par Antti Valmari dans [Val91] dans lequel il expose la notion de commutation et indépendance dans des systèmes concurrents. Ces travaux ont été initialement appliqués sur des réseaux de pétri. Comme décrit par la suite, il met aussi en relief le problème d'ignorance (***ignoring problem***) pouvant empêcher l'exécution équitable de transitions dans un état donné. Dans [GPS96] Godefroid unifie les relations d'indépendances sous le nom de ***persistent set***(ensemble persistant), et développe la méthode ***sleep set***(ensemble endormis) permettant de raffiner les critères de réduction des transitions d'un automate. Pour obtenir davantage de réductions, des méthodes par ordres-partiels dynamiques sont développées sous le nom d'indépendances conditionnelles. Néanmoins cette méthode nécessite un parcours non plus statique mais dynamique du système [FG05, GP93, KP92].

2.3.3 Obstacle aux ordres-partiels

Il existe une difficulté particulière pour appliquer les méthodes des ordres-partiels. Il s'agit du problème de l'équité entre les transitions exécutables d'un système. En effet, les algorithmes classiques des ordres-partiels étant non déterministes, des transitions (pertinentes pour la vérification) peuvent potentiellement ne jamais être visitées. C'est le problème de "**l'ignorance**" [Val91].

Exemple : événements ignorés. Considérons la figure 2.6. Supposons que tous les événements a, b, c soient indépendants. Dans l'état global initial $s0r0$, nous pouvons choisir entre l'exécution de a et de c puisque ces événements sont indépendants. Si a est choisi, l'état global $s1r0$ est atteint. b et c étant indépendants, si b est choisi arbitrairement, alors c peut ne jamais être exécuté. Cela peut être gênant si par exemple la propriété à vérifier concerne un état contenant $r1$.

Ce problème est résolu dans les travaux de Godefroid [God96] et de Peled [Pel93] en ajoutant des conditions supplémentaires dans la relation d'indépendance appelés ***proviso***. En effet les conditions stipulent que lorsqu'un cycle peut potentiellement empêcher d'exécuter une transition de manière équitable, on explore la ou les transitions qui pourraient potentiellement ne jamais être exécutées pour assurer l'équité. Il est toutefois à noter que ces conditions supplémentaires ne sont nécessaires que lorsque l'on souhaite vérifier des propriétés plus évoluées que des propriétés d'accessibilité.

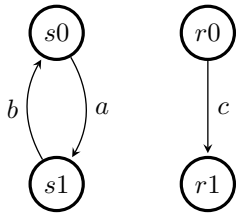


FIG. 2.6: Problème de l'ignorance

2.3.4 Comparaisons avec la méthode proposée

Dualité : systèmes ouverts/systèmes fermés. Premièrement, ce mémoire considère trois entités distinctes.

- le système,
- l'environnement (le contexte),
- la propriété.

Un contexte représente une configuration souhaitée de l'environnement du système de telle manière à l'amener dans des situations pertinentes pour la vérification. Un contexte est composé d'acteurs s'exécutant en parallèles et interagissant de manière bidirectionnelle uniquement avec le système. Chaque acteur peut alors exécuter des messages en séquence ou en alternative comme expliqué dans la partie consacrée au langage CDL (voir section 4.2).

Dans l'approche classique, ces entités sont composés à la volée pour construire au fur et à mesure les relations de commutation et d'indépendance pour chaque état global. Dans notre cas toutefois, ces relations de commutation sont définies en amont de la phase de composition sans la construction explicite des états. En effet, les méthodes précédentes appliquent les algorithmes de réduction directement lors de la génération de l'automate global ce que l'on veut clairement éviter. Ce mémoire propose de s'interfacer sur la méthodologie OBP dans l'objectif de réduire la complexité de la vérification d'un système industriel en décomposant l'environnement du système en scénarios équivalents.

Comme décrit précédemment l'équité dans la sélection des ensembles persistants est un problème récurrent. En effet, lors du calcul d'un ensemble persistant, la première transition est sélectionnée

dans un ordre arbitraire de manière non déterministe. Cela a pour conséquence que lorsqu'un système contient une boucle dans un certain état alors le problème de l'équité survient ; c'est à dire que cette sélection arbitraire peut entraîner une "ignorance" des autres transitions exécutables dans le même état et ainsi éliminer tout un sous ensemble de transitions qui pourraient être explorées par la suite (par exemple en sélectionnant à chaque fois la transition de la boucle). La relation d'indépendance est ainsi falsifiée. Ce problème est connu sous le nom de "ignoring problem" qui a été introduit par Antti Valmari. Toutefois, ce problème n'existe plus dans la méthodologie OBP puisque le contexte (l'environnement) représenté par un graphe fini **acyclique** est déplié. Par conséquent, tous les scénarios sont parcourus et le problème de l'équité est ainsi résolu.

Dans la thèse de Godefroid, la procédure de calcul des transitions indépendantes dans un état global peut être assujetti à une forte dégradation de ses résultats, par la méthode sur-approximative des persistent sets. En effet, il peut arriver que des transitions indépendantes ne puissent être éliminées dans un état dans le cas où chacune de ces transitions peut mener à une transition dépendante. Pour contrer ce phénomène, la méthode ***sleep set*** (ensemble endormis) a été introduite. Cette méthode est présentée dans le chapitre suivant.

Ici encore ce problème n'existe plus lorsque l'environnement est partitionné en scénarios complets couvrant entièrement le graphe de départ. Le chapitre suivant présente un exemple de calcul d'**ensemble endormis**.

Applications des algorithmes par ordres-partiels existant. Il a été démontré dans [God96] que les méthodes ***stubborn set***(développée par Antti Valmari [Val91]), et ***ample set*** (développée par Doron Peled [Pel93]), sont aussi des ensembles persistants. Par conséquent l'algorithme *persistent set* peut être appliqué à la place des deux autres algorithmes.

En revanche, dans ce mémoire, l'algorithme des persistent set n'est plus adaptée pour le calcul des relations d'indépendances. En effet, deux événements indépendants dans l'environnement n'implique pas obligatoirement leur indépendance dans le système.

Cette différence dans le calcul des relations d'indépendance est d'autant plus marquée par le fait de la présence de la sauvegarde de messages dans les modèles SDL qui ne sont pas traitées par les algorithmes précédents. Une indépendance sur les événements sauvegardés a toutefois été développée dans [HCR01] pour le langage Electre basé sur la sémantique des structure d'événements [Win86]. En effet les messages sauvegardés permettent d'introduire une plus grande quantité d'indépendance entre les événements car le système devient ainsi plus robuste aux différents stimulis de son environnement, pouvant ainsi prendre en compte des messages non attendus sans les perdre et sans changer le comportement du système.

2.4 Discussion générale

Comparaison des méthodes de réduction. Avant de poursuivre, une comparaison simple entre les 3 méthodes introduites précédemment est proposée :

- Méthode du *model-checking*
- Méthode classique des ordres-partiels
- Méthode proposée dans le mémoire

La figure 2.7 représente l'automate global d'un système. La méthode du *model-checking* parcourt exhaustivement l'ensemble des états et exécuter l'ensemble des transitions.

Sur la figure 2.8 l'exécution des algorithmes des ordres-partiels permet de réduire fortement les états inutiles et les transitions redondantes issues de l'automate précédent. Un sous automate réduit de l'automate initial est ainsi obtenu.

Sur la figure 2.10 une réduction plus importante que précédemment est obtenue car cette fois-ci une configuration bien particulière dans laquelle le système doit être vérifié est définie. Dans la méthode classique des ordres-partiels, l'algorithme va chercher à réduire à la volée l'ensemble

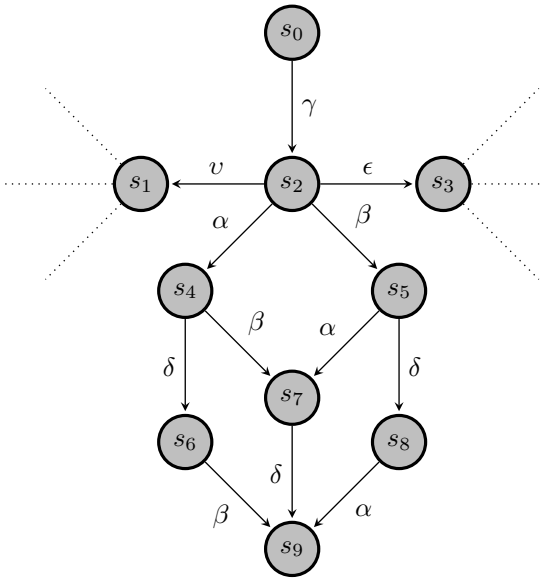


FIG. 2.7: *model-checking*

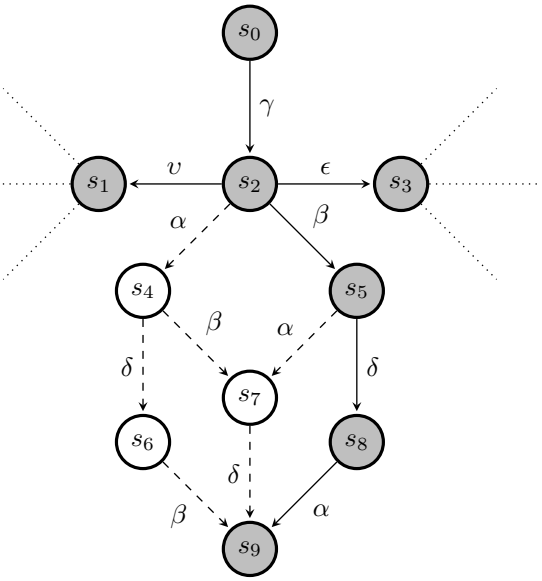


FIG. 2.8: Ordres-partiels

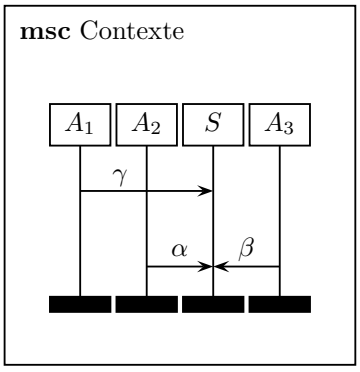


FIG. 2.9: Contexte

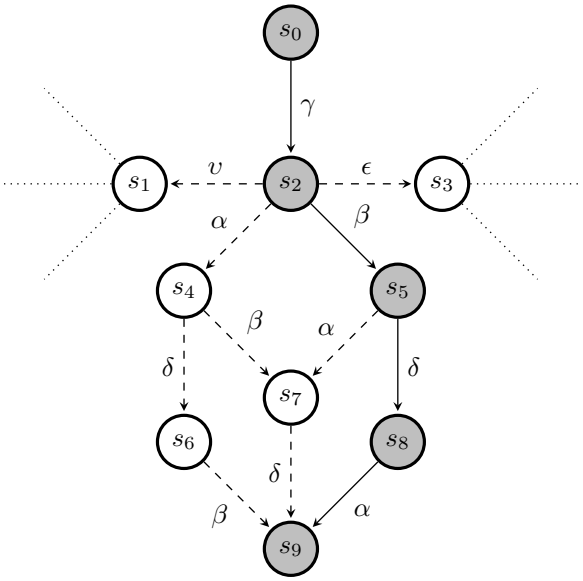


FIG. 2.10: Restriction par le contexte

de l'automate global sans chercher à déterminer si la partie parcourue est bien pertinente pour la vérification. Au contraire dans la méthode basée sur la notion de contexte, une incision est pratiquée sur les parties du système qui ne se trouvent pas dans la configuration prédéfinie. Cela est effectué en définissant un contexte qui va isoler la partie du système pertinente pour la vérification en cours. Nous voyons donc ici l'intérêt d'une telle approche. Cette méthode de réduction est détaillée dans le chapitre suivant.

Bilan. Ce chapitre a introduit un panorama des techniques existantes pour la vérification de systèmes de taille volumineux, ainsi que des méthodes de réduction permettant de dépasser l'explosion combinatoire des systèmes complexes. La méthode du *model-checking* reste limitée à des systèmes de petite taille. Les méthodes d'abstractions nécessitent beaucoup de créativité et peuvent engendrer de faux contre-exemples. La méthode qui nous intéresse particulièrement dans la suite de ce mémoire concerne les méthodes de réduction par ordres-partiels, qui permettent de factoriser

un ensemble de traces du système en classes d'équivalences. Ces méthodes sont présentées dans le chapitre suivant.

Chapitre 3

Principe des ordres-partiels

Sommaire

Résumé. Le chapitre précédent a exposé les méthodes de vérification formelle existantes pour la vérification de systèmes finis sur la base de la méthode du *model-checking*. La méthode par ordres-partiels se différencie du *model-checking* en effectuant un parcours non exhaustif du système à vérifier, en éliminant des séquences de transitions équivalentes (dites indépendantes) menant le système dans un même état. Dans ce chapitre, la théorie des ordres-partiels et les principaux algorithmes sont introduits. Un comparatif avec la méthode proposée est ensuite effectué. Tout d'abord, la théorie des traces de Mazurkiewicz est rappelée. Elle constitue la sémantique de base des méthodes par ordres-partiels basées sur des monoïdes libres, introduits par Foata et Cartier dans leurs travaux de combinatoire et de réarrangement [CF69].

3.1 Définition formelle : ordres-partiels

Définition 1 *Un ordre-partiel est une relation par ordre binaire $\mathcal{R}$ sur un ensemble E $(\leq, E)$ qui est :*

$$\textit{Réflexive } \forall x \in E, x\mathcal{R}x \tag{3.1}$$
$$\textit{Anti-symétrique } \forall (x,y) \in E^2, x\mathcal{R}y \wedge y\mathcal{R}x \Rightarrow x = y \tag{3.2}$$
$$\textit{Transitive } \forall (x,y,z) \in E^3, x\mathcal{R}y \wedge y\mathcal{R}z \Rightarrow x\mathcal{R}z \tag{3.3}$$

Intuitivement, un ordre partiel définit une relation binaire entre certains éléments de l'ensemble E. Un ordre total définit un ordre binaire entre tous les éléments de l'ensemble E.

On dira que deux événements sont concurrents si leur ordre d'occurrence est indéterminé : $evt1 \parallel evt2 \Leftrightarrow evt1 \not\leq evt2 \wedge evt2 \not\leq evt1$

3.2 Les relations d'équivalences

Les modèles de concurrence. Ce mémoire a pour but de déterminer des comportements équivalents de l'environnement d'un système donné. Pour cela, de nombreuses relations d'équivalence ont été énoncées. En effet, il existe de très nombreuses manière de représenter un système, de nombreux langages ayant différentes sémantiques. Un moyen d'unifier les représentations d'un système concurrent consiste à le représenter dans une structure mathématique telle que les automates, de telle manière à pouvoir raisonner de manière structurelle et comportementale.

Une autre manière de représenter un système consiste à déterminer l'ensemble de ses exécutions en partant d'un état initial à un état final. Il s'agit en fait d'un **dépliage** de l'automate global et **partitionnement** de l'ensemble de traces.

Une des principales question lorsque l'on manipule des automates volumineux est aussi d'apporter des solutions de réduction de son espace d'état. Pour cela il s'agit de déterminer très souvent des relations d'équivalences de telle manière à éliminer des parties du systèmes qui possèdent le même comportement modulo une relation d'équivalence. Sur la figure 3.1 est illustré un panorama des relations d'équivalences existantes. La partie supérieure regroupe les relations d'équivalences les plus fortes (celles qui différencient le plus d'automates et par conséquent permettant de considérer de très nombreuses équivalences de systèmes) et tout en bas les relations les plus faibles différenciant le moins d'automates. Il existe deux principaux critères de classifications pour distinguer deux systèmes de transitions :

- Des relations d'équivalences basées sur le temps linéaire,
- Des relations d'équivalences basées sur le temps arborescent.

Dans le premier cas, l'équivalence sur les traces d'exécution du système est considérée. Deux systèmes sont équivalents dès lors qu'ils possèdent les mêmes traces d'exécution. il s'agit d'équivalence de trace forte ou d'équivalence de trace complète.

Dans le deuxième cas, nous nous intéressons aux arbres d'exécution dans lequel un choix est possible parmi plusieurs embranchement. Parmi les relations d'équivalences les plus connues, nous pouvons citer les équivalences de simulation, de *bisimulation* [Mou92, Pin93].

Exemple. Par exemple, sur la figure 3.2, les deux automates possèdent les mêmes traces : $a \cdot b$ et $a \cdot c$. Par conséquent ils sont trace-équivalents puisqu'ils acceptent tous les deux les mêmes mots.

Sur la figure 3.3, les deux automates sont équivalents par la *bisimulation* forte puisque dans l'automate de gauche, à partir de p_0, on peut exécuter a une infinité de fois puis exécuter b ou bien exécuter directement b. Dans l'automate de droite, il existe les mêmes comportements : exécuter a une infinité de fois puis exécuter b. En résumé, les états q_0, q_2, q_3, q_4 sont "bisimilaires" avec l'état

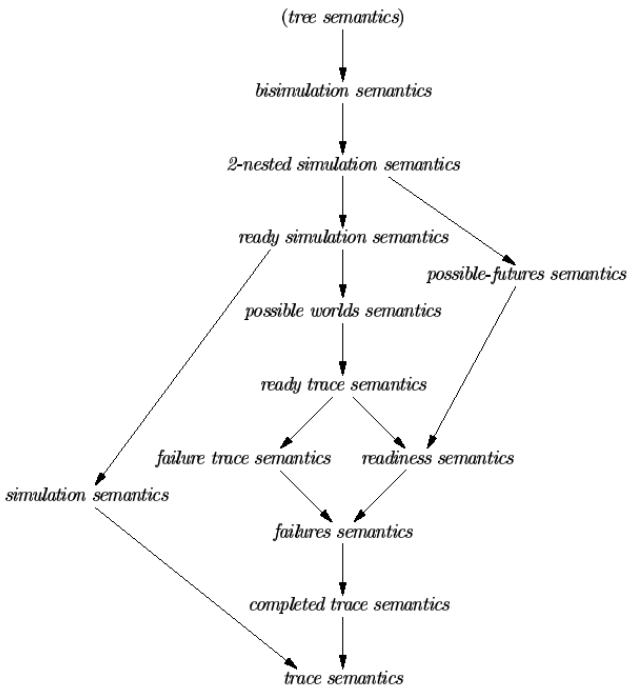


FIG. 3.1: Treillis des relations d'équivalences [Gla99]

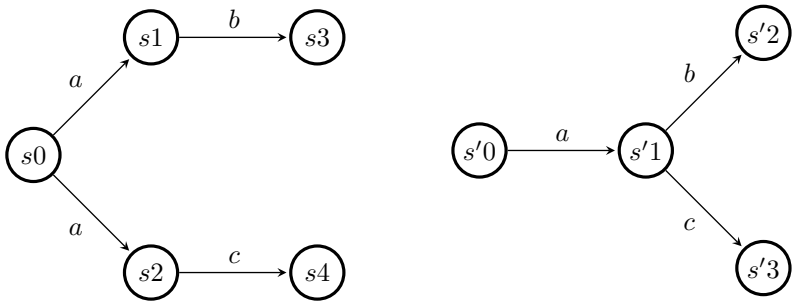


FIG. 3.2: Deux automates trace équivalents

p_0 puisque tous ces états ont exactement le même comportement. Enfin p_1 et q_1 sont "bisimilaires". Au final les deux automates sont "bisimilaires".

Pour faire le parallèle avec le cas du *model-checking*, il s'agit de déterminer si le modèle du système est inclus dans le modèle de la propriété à satisfaire, à savoir une relation de simulation : est ce que le modèle de la propriété simule le modèle du système.

Dans le cadre de ce travail consistant à identifier des relations d'équivalences entre les scénarios d'un environnement donné, la relation d'équivalence de trace est déterminée par une relation d'indépendance. Pour cela nous nous appuyons sur les équivalences des traces de Mazurkiewicz qui s'apparentent à l'équivalence des traces usuelles mis à part que cette fois ci les traces de Mazurkiewicz permettent de confondre beaucoup plus de traces d'exécution : non seulement celles possédant les même mots, mais aussi les mots équivalents modulo certaines permutations de certaines lettres dites indépendantes.

Remarque. Il existe une relation d'équivalence complémentaire aux deux précédentes qui est la relation d'équivalence de tests. Il s'agit cette fois-ci de considérer deux systèmes comme étant équivalents, si les mêmes tests effectués génèrent les mêmes résultats [CH90, NH83]. Ici encore le problème d'ignorance peut subvenir, empêchant ainsi le test de s'effectuer de manière équitable.

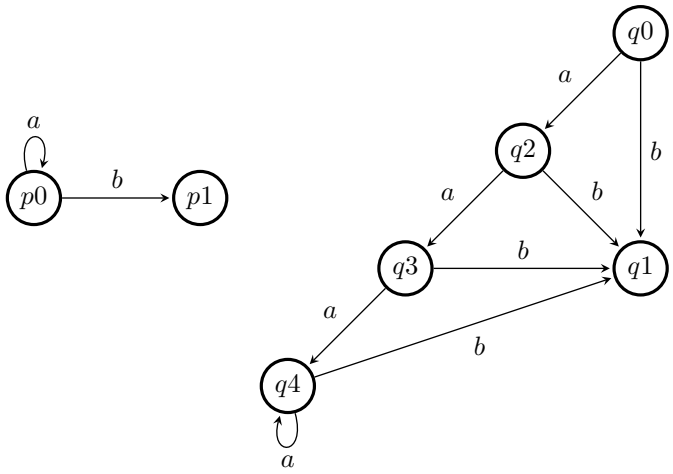


FIG. 3.3: Exemple de *bisimulation* forte

Pour cela la notion de test équitable a été introduite dans [RV07].

3.3 Théorie des traces

Cette section cherche à déterminer sous quelles conditions deux scénarios sont équivalents. Pour cela les traces de Mazurkiewicz sont considérées. En effet, étant donnée une relation d'indépendance I, il s'agit d'étendre cette relation entre deux événements à une relation entre deux scénarios. L'idée des traces de Mazurkiewicz est de considérer deux scénarios comme équivalents si l'on peut obtenir l'une par permutation d'événements indépendants de l'autre. Par la suite une procédure de décision évitant de calculer toutes les permutations entre scénarios est présentée : la méthode des formes normales de Foata [CF69].

3.3.1 Trace de Mazurkiewicz

Définition des traces de Mazurkiewicz. L'idée derrière les traces de Mazurkiewicz [Maz86] est qu'à partir d'une relation d'indépendance, deux traces peuvent être considérées comme étant équivalentes si la première peut être obtenue à partir de la seconde par des permutations successives d'événements indépendants. Soit $I \subseteq Sig \times Sig$ la relation d'indépendance entre deux événements (deux événements en relation par I commutent). $\mathbb{M}(Sig, I)$ est un monoïde libre partiellement commutatif représentant l'ensemble des traces quotientées par la plus petite congruence $\equiv_I$ sur Sig^* telle que $ab \equiv_I ba\ \forall (a,b) \in I^2$. Les traces de $\mathbb{M}(Sig, I)$ quotientées par $\equiv_I$ sont appelées traces de Mazurkiewicz. Soit w un mot de Sig^*, $[w]$ dénote la trace de Mazurkiewicz de w (pour une relation d'indépendance donnée).

Monoïde. Il s'agit d'une structure algébrique très utilisée en informatique et dans la théorie des langages. Elle a l'avantage de pouvoir décrire des systèmes évoluant dans le temps sans avoir la possibilité d'annuler ou de revenir en arrière. C'est exactement le cas en informatique. Cela est dû à leur propriété irréflexive. Un monoïde possédant l'opérateur interne de concaténation et contenant le mot vide $'1'$ est caractérisé de monoïde libre. Ce dernier permet de décrire les systèmes à l'aide d'un ensemble de traces.

Définition 2 $\mathbb{M} = (\mathbb{M}, ., 1)$ *est un monoïde si il satisfait aux axiomes suivants :*

$\forall x, y \in \mathbb{M}, x.y \in \mathbb{M}$ $\qquad$ $\forall x \in \mathbb{M},\ x.1 = 1.x = x$

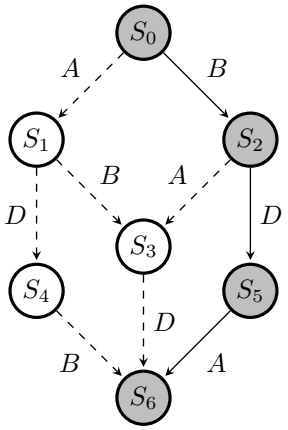
$\forall x, y, z \in \mathbb{M}, (x.y).z = x.(y.z)$

Définition 3 *Soit Sig un alphabet, ε le mot vide et l'opération de concaténation défini tel que $u.v = uv, \forall u, v \in Sig^* \times Sig^*$. Alors $Sig^* = (Sig^*, ., \varepsilon)$ est un* ***monoïde libre****.*

Définition 4 *Soit Σ un alphabet et Σ^* les mots finis sur Σ. Soit I une relation d'indépendance entre les lettres issues de Σ. Deux mots $(u, v) \in \Sigma^*$ sont équivalents : $u \sim_I v$ (avec $\sim_I$ la plus petite congruence sur Σ^*) ssi il existe un préfixe et un suffixe $x, y \in \Sigma^*$ et $(a, b) \in I$ tel que $u = xaby$ et $v = xbay$. $\mathbb{M}(\Sigma, I)$ est appelé* ***monoïde de trace partiellement commutatif****. Il représente l'ensemble de toutes les traces.*

Application au cadre de travail. Dans la suite de ce mémoire, les lettres de l'alphabet Σ symbolisent les événements envoyés par le contexte au système ; les mots finis de Σ^* représentent les scénarios du contexte et chaque classe de Mazurkiewicz contient des scénarios tous équivalents.

Exemple. Considérons trois événements A, B et D. Supposons qu'ils soient indépendants deux à deux (ils commutent et leur ordre est sans importance). Etant donnée la relation d'indépendance $I = \{(A, B), (B, D), (A, D)\}$, la classe d'équivalence de ABD est définie par : $[ABD] = \{ABD, ADB, BAD, BDA, DAB, DBA\}$ puisque tous les événements sont indépendants. Tous les scénarios contenus dans la classe $[ABD]$ sont équivalents et un seul sera nécessaire lors de la vérification avec le système.



- Soit $\Sigma = \{A, B, D\}$
- Soit $I = \{(A, B); (A, D), (B, D)\}$
- $\mathbb{M}(\Sigma, I) = \{ABD, ACB, BAD, BDA, DAB, DBA\}$
- $[ABD] = \{ABD, ADB, BAD, BDA, DAB, DBA\}$

3.3.2 Forme Normale de Foata.

La question est alors, à partir d'une relation d'indépendance, comment calculer la liste des $[w]$ distinctes, sans avoir à prendre en compte toutes les permutations d'événements adjacents. Pour ce faire la méthode des formes normales de Foata est utilisée. L'idée consiste à récrire un mot à partir d'un ensemble de règles de réécriture de manière à aboutir à une unique forme irréductible : la **forme normale**. La forme obtenue étant **unique**, toutes les traces ayant la même forme normale appartiendront à une même classe de Mazurkiewicz. Dans [DM97] une procédure de transformation d'un mot en sa forme normale de Foata modulo une relation d'indépendance I est présentée. C'est cette méthode qui est utilisée dans ce mémoire pour calculer les traces équivalentes.

Définition 5 *Un mot de Sig^* est sous la **forme normale de Foata** si il correspond au mot vide ou si il existe un nombre $n > 0$ de mots non vide $x_i (1 \leq i \leq n)$ tel que*

1. $x = x_1 ... x_n$,
2. $\forall i$, *le mot* x_i *est un produit de lettres adjacentes indépendantes et* x_i *est écrit en respectant l'ordre lexicographique.*
3. $\forall i,\ 1 \leq i \leq n\ \forall a \in x_{i+1}\ \exists b \in x_i, (a, b)$ *soient dépendants*

Procédure de décisions pour le calcul des formes normales. Considérons le monoïde de traces $\mathbb{M}(Sig, I)$ et un ensemble de piles utilisé pour représenter chaque lettre. La trace est lue de la droite vers la gauche. Chaque lettre est ajoutée dans la pile correspondante. Un marqueur (astérisque) est ajouté dans chaque pile non indépendante avec la lettre courante. Pour obtenir la forme normale de Foata, les lettres situées à la tête de chaque pile sont dépilées, réarrangées dans l'ordre lexicographique. Un marqueur est dépilée dans chaque pile correspondant à une lettre qui est dépendante de l'ensemble sélectionné.

Exemple. Considérons l'alphabet suivant $Sig = \{a, b, c, d\}$ et les relations d'indépendances $I = \{(a, d), (b, c)\}$. Considérons que les piles de la figure 3.4 sont organisées dans l'ordre lexicographique : la première colonne de gauche est la colonne de a, la deuxième celle de b, la troisième celle de c et enfin la dernière celle de d. Calculons la forme normale de Foata pour la trace *badacb*. La dernière lettre de la trace est tout d'abord sélectionnée. Il s'agit de la lettre b qui est ajoutée dans la colonne correspondante. Cette lettre est indépendante avec c et donc dépendante avec a et d. Par conséquent un astérisque est ajouté dans les colonnes a et d. La procédure est réitérée sur la lettre suivante en partant de la fin : la lettre c. La lettre est ajoutée dans sa colonne et un astérisque est marqué dans les colonnes correspondant à a et d. En effectuant les mêmes opérations sur les lettres restantes, les piles de la figure 3.4 sont obtenues.

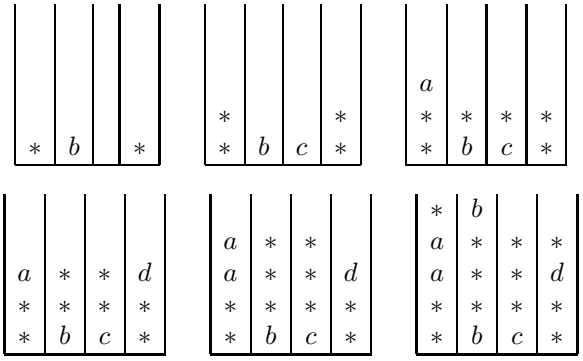


FIG. 3.4: Initialisation de la pile

Maintenant calculons la forme normale à partir de l'état des piles obtenues. Toutes les lettres au *top* de la pile (*Last In First Out*) sont tout d'abord sélectionnées. Pour la figure 3.3.2 les lettres entre crochets sont obtenues. Parmi les éléments sélectionnés, les lettres sont organisées sous forme lexicographique. Ici, il n'y a que b. b est sélectionnée et dépilée. Il faut encore enlever les astérisques présents dans les colonnes dépendantes des lettres sélectionnées ; ici dépendantes de b. Il s'agit des colonnes a et d. Un astérisque présent dans ces deux colonnes (symbolisé par la barre) est dépilé. Sur la figure 3.3.2, l'état des piles est représenté, après avoir effectué les opérations décrites précédemment. Les *top* de chaque pile sont sélectionnés et organisés sous forme lexicographique. les lettres a et d ainsi obtenues sont organisées sous leur forme ad (ordre lexicographique). Comme précédemment un astérisque est dépilé dans les colonnes dépendantes de a et de d. Ici les colonnes b et c. Au final obtient la forme normale de Foata de la trace *badacb* est (b)(ad)(a)(bc).

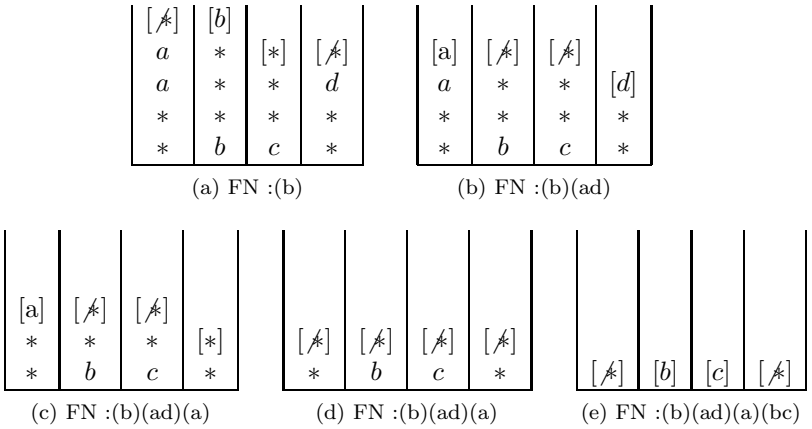


FIG. 3.5: Forme normale de Foata (FN)

3.4 Algorithmes de réduction des méthodes par ordres-partiels

Objectif. Le but des algorithmes qui sont présentés dans cette section est d'éliminer à la volée, des transitions qui peuvent être permutées sans modifier le comportement du système. Ici, il s'agit de conserver les propriétés d'accessibilité et donc en particulier les *deadlocks*[1].

Algorithme général. La méthode de réduction des ordres-partiels consiste à sélectionner l'état global initial et de calculer les transitions qui seront suffisantes pour la vérification ; l'objectif étant de sélectionner un minimum de transitions. Pour chaque transition retenue, il s'agit ensuite de parcourir chaque état voisin atteint (par l'exécution des transitions sélectionnées dans l'état parent), et réitérer le calcul précédent jusqu'à ce qu'aucune transition ne puisse être sélectionnée. L'ensemble des transitions sélectionné pour chaque état est calculé par l'algorithme de réduction (**AlgorithmeReduction**). L'algorithme du parcours en profondeur d'abord itératif est présenté ci-dessous. *Stack* contient les états restant à visiter et H les états déjà visités. Dans la suite de ce chapitre, les principaux algorithmes de réduction existant dans la littérature sont décrits.

```
selectiveSearch :
Stack ← ∅;
H ← ∅;
Stack.push(s0);
while Stack ≠ ∅ do
    if (s = Stack.pop ∉ H) then
        H.push(s);
        T = AlgorithmeReduction(s);
        foreach t ∈ T do
            s' = t.succ(s);
            Stack.push(s')
        end
    end
end
```

Algorithme 1 : Parcours en profondeur d'abord itératif

[1] Etat dans lequel aucun processus ne peut progresser.

3.4.1 Approche classique

De manière générale la méthode de réduction par ordres-partiels s'effectue de la manière suivante. Pour chaque état d'un système global, les transitions (le plus petit possible) que l'on doit parcourir par la suite sont calculées, en supprimant toutes les transitions qui mèneraient à des entrelacements équivalents. Ainsi pour chaque état un ensemble dit persistant est calculé représentant les seules transitions à exécuter pour la vérification.

Pour déterminer les transitions à sélectionner parmi celles qui peuvent être exécutées (transitions dites ***enabled*** pour signifier les transitions exécutables dans l'**état global** courant) dans l'état courant, une relation d'indépendance doit être fournie. Une analyse statique du modèle en cours de validation permet d'extraire cette relation.

La différence entre les différentes méthodes présentes dans la littérature concerne la définition de cette relation d'indépendance. En effet selon les informations utilisées pour obtenir cette relation, les coûts de calculs seront modifiés ainsi que le nombre de transitions à prendre en compte pour chaque état. Ainsi, plus le nombre d'informations utilisé pour déterminer l'indépendance de deux événements est important, plus le temps de calcul de l'indépendance sera long mais l'on peut espérer une réduction du nombre de transitions plus importante. A l'inverse moins le nombre d'informations utilisé est important et plus les calculs seront rapides avec une espérance d'un gain de réduction moindre.

En réalité le fait de prendre en compte plus d'informations ne mène pas toujours à un meilleur résultat et différents algorithmes ont été étudiés dans [GPS96] pour évaluer les performances.

Réduction par ordres-partiels. Dans les systèmes concurrents, les processus s'exécutent en parallèles, les scénarios d'exécutions sont définis par l'ensemble des entrelacements des événements des différents processus. Pour n événements concurrents, il existe $n!$ possibilités d'exécutions. Si tous ces événements sont indépendants les uns des autres, alors tous les entrelacements sont équivalents puisqu'ils mènent à un même état. Dans ce cas là un seul des entrelacement est nécessaire pour simuler le comportement du système.

Ainsi, les méthodes par ordres-partiels permettent de déterminer un sous-ensemble de scénarios suffisant pour vérifier le système exhaustivement, sans avoir à considérer l'automate global. Pour cela, deux méthodes vont être présentées : la méthode dite *persistent sets* et la méthode *stubborn sets*.

Exemple général : le dîner des philosophes. Pour appliquer les algorithmes des ordres-partiels issus de la littérature présentés dans ce chapitre, un exemple simple et bien connu est introduit : le dîner des philosophes qui est un problème classique de partage de ressources sur les systèmes d'exploitation. Pour simplifier la taille de l'automate et la compréhension des algorithmes, le système est restreint à deux philosophes[2]. Pour N philosophes, il y a donc N baguettes disponibles. Le schéma de ces deux philosophes est représenté figure 3.6. Ainsi, les deux philosophes partagent les mêmes baguettes ; la baguette de droite R_1 du philosophe P_1 correspond à la baguette gauche L_2 du philosophe P_2. Les comportements des philosophes sont représentés par deux structures de Kripke. Chaque philosophe peut se trouver dans 3 états :

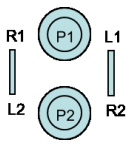


FIG. 3.6: Dîner de deux philosophes avec deux baguettes

[2]Dans la littérature, le problème est généralement composé de 5 philosophes

- Le philosophe Pense,
- le philosophe à Faim,
- le philosophe Mange.

Pour obtenir l'automate global, le produit synchronisé est effectué sur les deux automates définis par :

$\Phi_1 = \langle \Sigma_1, A_1, Tr_1, I_1, AP_1, l_1 \rangle$:
- $\Sigma_1 = \{0, 1, 2, 3, 4\}$
- $A_1 = \{L_1, R_1, \overline{L_1}, \overline{R_1}, T_1\}$
- Tr_1 est l'ensemble des 5 transitions
- $I_1 = \{0\}$
- $AP_1 = \{PENSE, AFAIM, MANGE\}$

$\Phi_2 = \langle \Sigma_2, A_2, Tr_2, I_2, AP_2, l_2 \rangle$:
- $\Sigma_2 = \{0, 1, 2, 3, 4\}$
- $A_2 = \{L_2, R_2, \overline{L_2}, \overline{R_2}, T_2\}$
- Tr_2 est l'ensemble des 5 transitions
- $I_2 = \{0\}$
- $AP_2 = \{PENSE, AFAIM, MANGE\}$

La notation barrée représente l'action du philosophe P_i reposant la baguette correspondante. T_1 (respectivement T_2 symbolisent l'action de penser du philosophe P_i.

Dans cet exemple, le comportement des deux philosophes est décrit de la même manière, à savoir :

1. Le philosophe pense (T_1 respectivement T_2),
2. Le philosophe commence par prendre la baguette de gauche si elle est disponible (L_1 respectivement L_2),
3. Le philosophe continue avec la baguette de droite si elle est disponible (R_1 respectivement R_2),
4. Le philosophe mange dès qu'il possède les deux baguettes,
5. Le philosophe commence par déposer la baguette de gauche une fois son repas terminé ($\overline{L_1}$ respectivement $\overline{L_2}$),
6. Il relâche ensuite la baguette de droite ($\overline{R_1}$ respectivement $\overline{R_2}$),
7. Le philosophe se remet à penser (T_1 respectivement T_2).

Nous obtenons l'automate global de la figure 3.7. Nous effectuons toutefois le produit synchronisé avec des restrictions sur certains états. En effet, les états grisés ne peuvent être atteints ; les transitions en pointillés ne peuvent être exécutées. Cela est dû au fait que les deux philosophes ne peuvent simultanément accéder aux mêmes baguettes. Par conséquent les états grisés ne sont pas atteignables. Notons qu'il existe un *deadlock* dans cet automate dans l'état 11.

Intuitivement, sur la figure 3.6 les deux philosophes partagent les mêmes baguettes et par conséquent toutes les transitions R_1 sont dépendantes de L_2 et de même pour R_2 et L_2. Par conséquent sur ce premier exemple aucune réduction ne peut être obtenu.

Modifions la disposition initiale des philosophes et introduisons une troisième baguette comme décrit sur la figure 3.8.

Cette fois-ci, seules les transitions R_1 et L_2 ne peuvent être effectuées simultanément puisque cela impliquerait que les deux philosophes accèdent à une même baguette. En effectuant le produit synchronisé, l'automate de la figure 3.9 est obtenu, avec quatre états inaccessibles.

Algorithmes classiques. Nous appliquons à présent les algorithmes des *stubborn set* et *persistent set* sur l'exemple 3.9 pour comparer les réductions obtenues.

3.4.2 Algorithme *Stubborn set*

Intuitivement, l'algorithme se traduit par :

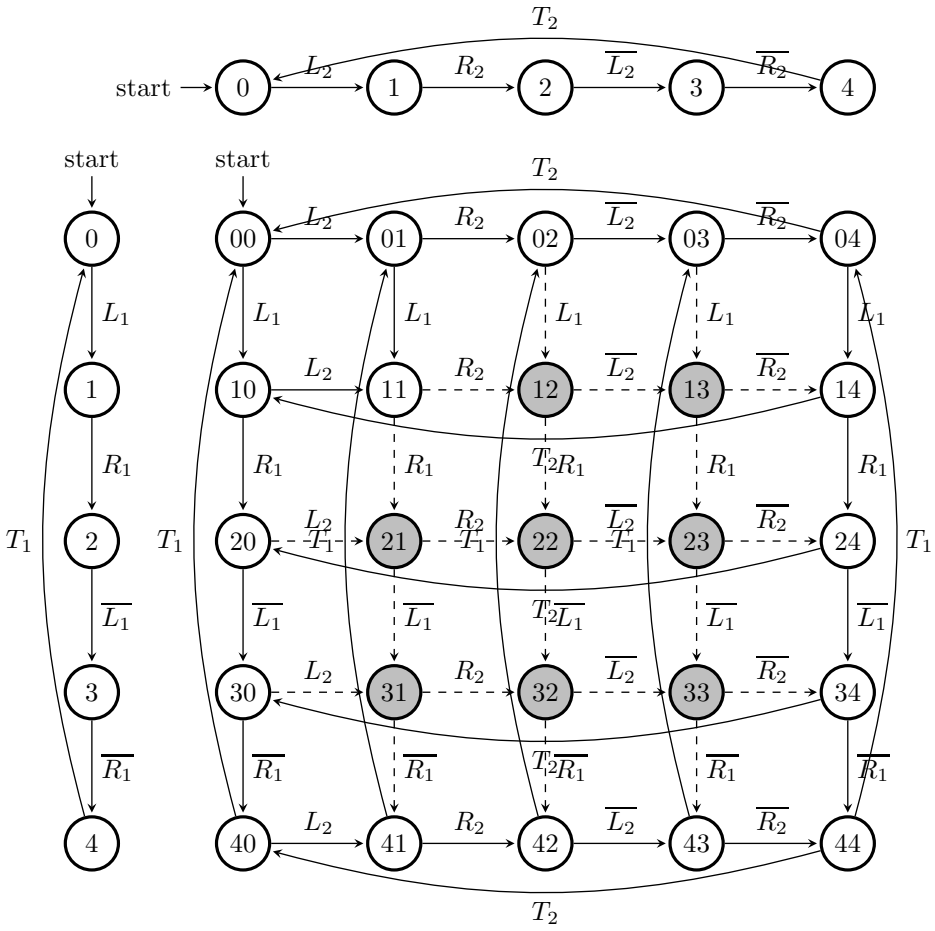


FIG. 3.7: Le dîner des deux philosophes

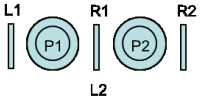


FIG. 3.8: Dîner de deux philosophes et trois baguettes

- Pour chaque état s choisir une transition $t0$ exécutable que l'on ajoute dans SS (*Stubborn Set*).
- Pour chaque transition t de SS, toutes les transitions dépendantes de t à partir de l'état s sont sélectionnées.
- Parmi ces transitions dépendantes qui ne sont pas exécutables à partir de s, les transitions exécutables sont sélectionnées dans l'état s. Elles permettront d'atteindre la ou les transitions dépendantes
- Les transitions exécutables à partir de s sont renvoyées

Exemple. Reprenons l'exemple de la figure 3.7. Les transitions dépendantes sont définies par le couple : $D = \{(R_1, L_2)\}$. Appliquons l'algorithme *stubborn set* sur l'état 00 de l'automate global.

- La transition L_1 est sélectionnée : $SS(00) = \{L_1\}$ est sélectionnée. Parmi toutes les transi-

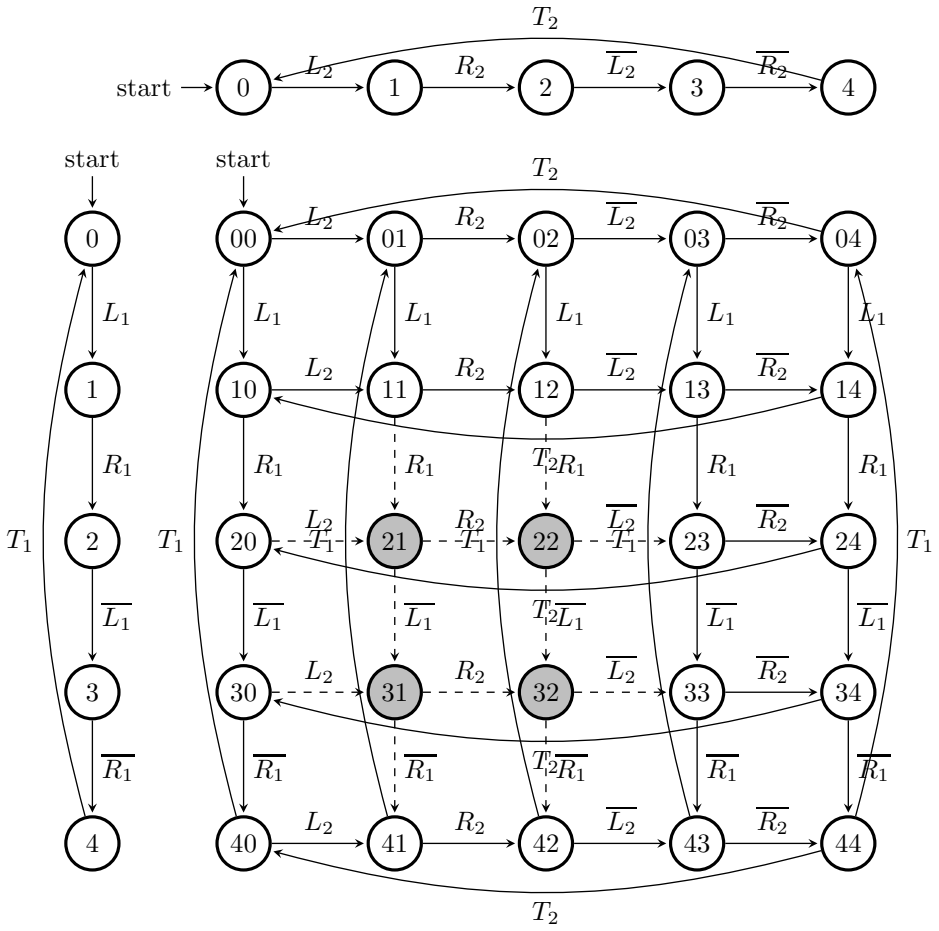


FIG. 3.9: Le dîner des deux philosophes

tions de l'automate global, celles qui sont dépendantes avec L_1 sont sélectionnées. Ici il n'y en a pas. Il est important de remarquer que le choix non déterministe de transition de départ peut modifier la réduction obtenue. En effet si L_2 avait été sélectionnée à la place de L_1, la dépendance entre R_1 et L_2 aurait été prise en compte. Cela aurait impliqué la sélection de toutes les transitions de 00 pour prendre en compte, le cas où R_1 est exécuté avant L_2 et réciproquement. Ici il n'y a aucune dépendance avec L_1. Par conséquent, $SS(00) = \{L_1\}$.

- Cette étape est inutile puisque qu'il n'existe aucune dépendance avec les transition L_2.
- Identique à l'étape précédente.
- L'ensemble *stubborn set* est l'ensemble constitué de $SS(00)$ privées des transitions non exécutables à partir de 00 ; c'est à dire $SS(Q0) = \{L_1\}$.

Calculons à présent l'ensemble *stubborn set* dans l'état 10.

- La transition R_1 est sélectionnée. $SS(10) = \{R_1\}$.
- Parmi toutes les transitions de l'automate global, les transitions dépendantes avec R_1 sont sélectionnées. Ici il y en a une : L_2. $SS(10) = \{R_1, L_2\}$.
- Dans l'état 10 la transition L_2 est exécutable et par conséquent, $SS(10) = \{R_1, L_2\}$.
- L'ensemble *stubborn set* est l'ensemble constitué de $SS(10)$ privées des transitions non exé-

cutables à partir de 10 ; c'est à dire $SS(10) = \{R_1, L_2\}$.

On doit à présent effectuer les mêmes opérations pour les nouveau états atteints : 20 et 11. Pour ces deux états le calcul est direct puisqu'une seule transition est exécutable. On obtient donc : $SS(20) = \{\overline{L_1}\}$ et $SS(11) = \{R_2\}$ et ainsi de suite jusqu'à ce que qu'un état ait déjà été visité. Le calcul des ensembles endormis est résumé dans le tableau de la figure 3.10.

3.4.3 Algorithme *Persistent set*

De la même manière que pour l'algorithme des stubborn sets, nous sélectionnons les mêmes transitions initiales.

Algorithme informel : *persistent set*. Intuitivement, l'algorithme est définit de la manière suivante :

- Pour chaque état s choisir une transition $t0$ parmi celles qui sont exécutables et l'ajouter par défaut dans l'ensemble persistant : $PS(s) = \{t0\}$
- Ajouter dans PS toutes les transitions t exécutables à partir de s dépendantes avec $PS(s)$
- Si il existe une transition dépendante avec PS mais non exécutable dans l'état s, alors toutes les transitions exécutables de l'état s sont ajoutées dans PS.

Appliquons l'algorithme *persistent set* sur l'état 00 de l'automate global.

- L'état initial global est 00. Une transition exécutable à partir de cet état est sélectionnée et ajoutée dans l'ensemble persistant noté $PS(s)$ pour l'état s. $PS(00) = \{L1\}$.
- Dans cet état il n'y a pas de transitions dépendantes avec L_1
- De la même manière il n'existe aucune transition dépendante avec L_1. ,Nous obtenons $PS(00) = \{L_1\}$

Effectuons la même opération pour l'état suivant 10.

- Une transition exécutable est sélectionnée à partir de l'état 10. Elle est ajoutée dans l'ensemble persistant PS. $PS(10) = \{R_1\}$.
- Dans cet état il n'y a qu'une transition dépendante avec R_1 ; il s'agit de L_2 qui est exécutable à partir de l'état 10. $PS(10) = \{R_1, L_2\}$.
- Toutes les transitions dépendantes étant exécutables dans l'état courant, finalement, $PS(10) = \{R_1, L_2\}$

En appliquant l'algorithme sur le système complet, nous obtenons les résultats de la figure 3.10.

Bilan. Pour le système des philosophes, les mêmes résultats sont obtenus si l'on sélectionne les mêmes transitions de départ dans chaque état. La principale différence entre les deux algorithmes provient du fait que lorsque deux transitions peuvent être dépendantes (pas forcément dans l'état courant), la méthode *stubborn set* va essayer de découvrir par quel scénario il va pouvoir accéder à cette transition dépendante, et ainsi sélectionner la transition exécutable dans l'état courant, menant à cette transition. La méthode *persistent set* ne fait pas autant d'efforts ; si elle sait qu'une transition dépendante existe mais n'est pas exécutable dans l'état courant, elle va sélectionner tout simplement toutes les transitions possibles de l'état courant, pour n'oublier aucune possibilité d'atteindre un jour la transition dépendante. La méthode *persistent set* effectue donc une sur-approximation dans le calcul des transitions. Bien qu'elle soit de complexité inférieure à la méthode *stubborn set*, les mêmes résultats sont obtenus. Dans sa thèse, Godefroid montre que les deux algorithmes sont efficaces pour certains systèmes spécifiques, et peu efficaces pour d'autres et

Etat	t_0	$SS(t_0)$	$PS(t_0)$
00	L_1	$\{L_1\}$	$\{L_1\}$
10	R_1	$\{R_1, L_2\}$	$\{R_1, L_2\}$
20	$\overline{L_1}$	$\{\overline{L_1}\}$	$\{\overline{L_1}\}$
11	R_2	$\{R_2\}$	$\{R_2\}$
30	$\overline{R_1}$	$\{\overline{R_1}\}$	$\{\overline{R_1}\}$
12	$\overline{L_2}$	$\{\overline{L_2}\}$	$\{\overline{L_2}\}$
40	L_2	$\{L_2\}$	$\{L_2\}$
13	$\overline{R_2}$	$\{\overline{R_2}\}$	$\{\overline{R_2}\}$
41	R_2	$\{R_2\}$	$\{R_2\}$
14	R_1	$\{R_1\}$	$\{R_1\}$
42	$\overline{L_2}$	$\{\overline{L_2}\}$	$\{\overline{L_2}\}$
24	$\overline{L_1}$	$\{\overline{L_1}\}$	$\{\overline{L_1}\}$
43	$\overline{R_2}$	$\{\overline{R_2}\}$	$\{\overline{R_2}\}$
34	$\overline{R_1}$	$\{\overline{R_1}\}$	$\{\overline{R_1}\}$
44	T_2	$\{T_2\}$	$\{T_2\}$

FIG. 3.10: Calcul des ensembles têtus et persistants

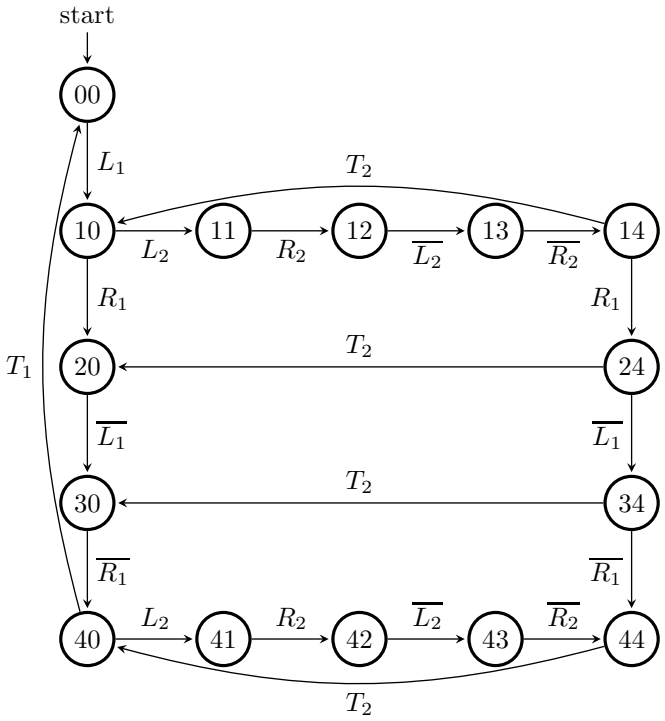


FIG. 3.11: Le dîner des deux philosophes

réciproquement. Il n'y a pas à proprement parler de meilleur algorithme pour chaque situation, et des heuristiques doivent permettre de déterminer le meilleur choix d'algorithme.

3.4.4 Algorithme Sleep set

Limitation des méthodes précédentes. Il peut arriver que des transitions indépendantes fassent partie d'un même ensemble persistant dans un état de l'automate global. Dans ce cas-là tous les entrelacements entre deux transitions indépendantes seront quand même calculées. Sur la figure 3.12, supposons que A et C sont indépendantes, et A et B dépendantes et appliquons

l'algorithme des *persistent sets* :

- Une transition est tout d'abord sélectionnée arbitrairement A. $PS(s0) = \{A\}$
- Les transitions de $s0$ exécutables et dépendantes avec A sont ajoutées dans l'ensemble persistant. Ici il n'y en a aucune.
- Une cherche des transitions dépendantes avec A dans le système global est ensuite effectuée. B est dépendant avec A. Par conséquent, toutes les transitions exécutables dans $s0$ sont ajoutées dans l'ensemble persistant.
- $PS(s0) = \{A, C\}$

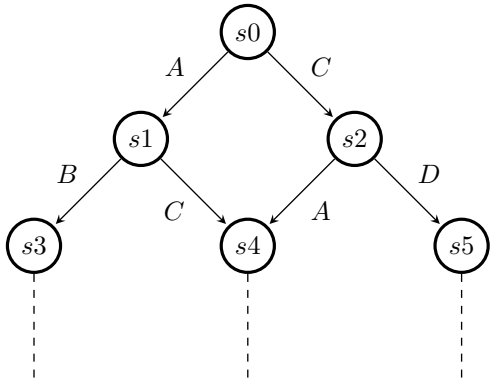


FIG. 3.12: Limitation de la méthode *persistent sets*

Nous voyons ici la limitation des *persistent set* qui n'éliminent pas toujours les séquences de transitions équivalentes. Même si les transitions C et A sont indépendantes, nous ne pouvons pas éliminer une des deux séquences du "losange". En effet, cela pourrait provoquer l'impossibilité d'exécuter B ou bien D. Pour pallier à ce problème, la méthode *Sleep set* a été introduite dans [GPS96]. Elle consiste à supprimer lors du calcul d'un ensemble persistant, les transitions indépendantes qui ont été exécutées par d'autres séquences de transitions. Pour cela un historique des transitions explorées est mémorisé sur chaque état. Cette fois-ci, seules les transitions sont supprimées contrairement aux algorithmes précédent qui suppriment les états globaux.

Algorithme informel : *sleep set.* L'algorithme consiste à identifier l'ensemble des transitions qui ont déjà été exécutées dans d'autres états selon l'historique des états déjà explorés et en éliminant ainsi des redondances de transitions. A noter que dans l'état initial, l'ensemble endormi correspond à l'ensemble vide. Les ensembles endormis sont construits dynamiquement en fonction des explorations précédentes. Ainsi lorsque l'on atteint un état q' à partir d'un état q en exécutant une transition t, alors l'algorithme de calcul des ensembles endormis effectue deux mises à jour sur les ensembles endormis des états q et q' de la manière suivante :

- L'ensemble endormi de q' hérite des transitions t' de l'ensemble endormi de q tel que t et t' sont indépendantes dans l'état q.
- L'ensemble endormi associé à q est augmenté de t, ce qui permet de mémoriser les transitions explorées

Exemple d'application *persistent sets + sleep sets.* Nous allons réduire l'automate partiel de la figure 3.12 à l'aide de la combinaison des algorithmes *persistent set* et *sleep set.*

- L'ensemble endormi de $s0$ est initialisée avec l'ensemble vide ; $sleep(s0) = \emptyset$
- Commençons par calculer l'ensemble persistant de $S0$. $PS(S0) = \{A, C\}$
- A cet ensemble les transitions qui ont déjà été exécutées dans d'autres séquences sont supprimées : $T = PS(S0) \backslash sleep(S0)$. ici $sleep(S0) = \emptyset$ donc $PS(S0)$ reste le même.
- Exécutons maintenant chacune des transitions a et c pour trouver les ensembles endormis des états successeurs par a et c : $S1$ et $S2$.
- Commençons par a. L'état atteint est donc $S1$. L'ensemble endormi de $S1$ est calculé en cherchant les transitions de l'ensemble endormi de l'état prédécesseur (ici $S0$) qui sont indépendantes avec a. Ici, $sleep(S1) = \emptyset$ puisque $sleep(S0) = \emptyset$
- L'ensemble endormi de $S0$ devient : $sleep(S0) = sleep(S0) \cup \{a\}$. La transition exécutée dans l'état $S0$ est ajoutée pour pouvoir ensuite signaler à $S2$ qu'il n'aura plus besoin d'exécuter a. Au terme de cette étape, $sleep(S0) = \{a\}$.

3.5 Discussion

Jusqu'à présent, l'ensemble des travaux issus de la littérature portant sur les ordres-partiels sont appliqués à la volée sur l'automate du système global, afin de réduire la taille du système, et d'appliquer par la suite la technique classique du *model-checking*.

Dans ce mémoire cependant, nous adoptons une approche orthogonale en appliquant la méthode de réduction non plus sur le système global, mais séparément sur la spécification du système, sur l'environnement et sur la propriété à valider. Cette approche est justifiée par plusieurs raisons.

La première est que les systèmes deviennent de plus en plus complexes de part l'essor de nouvelles technologies toujours plus sophistiquées, et de la criticité de ces applications. Les fonctionnalités devenant de plus en plus nombreuses ne peuvent empêcher une explosion combinatoire du système. Par conséquent, la méthode brute consistant à réduire le système d'automate global ne peut perdurer dans le temps.

Une deuxième raison est que chaque exigence du système n'est pertinente que pour certaines configurations du système et par conséquent, de nombreux états et transitions n'auront aucune utilité pour la vérification et la validation du système concernant l'exigence en cours. C'est pourquoi, le fait d'éliminer les parties du système sans répercussion sur la propriété à vérifier (à l'aide de la description d'un environnement précis du système), permet de réduire le système en amont du *model-checking*.

Nous pouvons appliquer la méthode des traces de Mazurkiewicz pour réduire les scénarios de l'environnement. Par contre, la méthode usuelle de réduction à la volée des ordres-partiels est incompatible car nous souhaitons réduire le nombre de scénarios avant même la composition avec le système. De plus, des événements indépendants dans l'environnement n'implique pas forcément leur indépendance dans le système.

Reprenons l'exemple des deux philosophes 3.8 et ajoutons un serveur distribuant les baguettes. Supposons que les philosophes puissent à présent saisir la baguette du milieu uniquement lorsqu'ils ont saisi leur baguette respective. Pour le serveur il est évident que la pose des trois baguettes est indépendante puisqu'au final il aura dressé la table. En revanche, cela n'est plus vrai pour les philosophes. Imaginons que la première baguette posée soit celle de gauche ; alors le philosophe P_2 pourra manger en premier. Par contre si c'est d'abord la baguette R_2 qui est déposée en premier, le philosophe P_1 mangera le premier. C'est pourquoi les relations d'indépendances que l'on définit par la suite doivent non seulement prendre en compte le comportement de l'**environnement : le serveur**, mais aussi du **système : les philosophes** et des **propriétés : manger le premier** à vérifier.

Par la suite, nous proposons d'adapter les algorithmes de Godefroid et Valmari au cas où l'automate global est séparé en deux entités : l'environnement et le système lui-même. Dans ce cas,

les relations d'indépendances doivent être modifiées.

Deuxième partie

Contribution de la thèse

Introduction

La partie précédente a introduit les méthodes par ordres-partiels, pour la réduction à la volée de l'automate qui serait issu du produit synchronisé du système, de l'environnement et de l'observateur. Or, dans le cas de systèmes complexes, ce produit synchronisé explose.

Dans cette partie, nous proposons d'appliquer les méthodes de réduction par ordres-partiels sur le contexte, en déterminant une relation d'indépendance sur les événements de ce dernier, par une analyse statique du système, de l'environnement mais aussi de l'observateur. Cette relation d'indépendance permettra ainsi de calculer les classes de Mazurkiewicz pour les scénarios du contexte. Cette réduction est appliquée en **amont** de la composition entre le système et son environnement rendant ainsi inutile la construction du produit synchronisé complet.

Cette partie s'articule de la manière suivante : Dans le chapitre 4, le cadre de modélisation est tout d'abord formalisé, en commençant par la formalisation d'un fragment du langage SDL pour la modélisation du système (section 4.1), puis par la formalisation d'un sous-ensemble du langage CDL pour la modélisation de l'environnement (section 4.2), puis enfin la formalisation des observateurs pour l'expression des propriétés (section 4.4).

Le chapitre 5 formalise les relations d'indépendance entre événements du contexte CDL dans le cas d'un système SDL mono-processus ainsi que la méthode de calcul des classes d'équivalence de scénarios du contexte, par la méthode des formes normales de Foata. Les algorithmes du calcul des relations d'indépendances sont présentées dans la section 5.4.2. La fin de ce chapitre est consacré à une comparaison entre la méthode proposée et le vérifieur SPIN. Pour comparer les performances, les expérimentations sont également effectuées sur le *model-checker* SPIN (implantant la méthode par ordres-partiels décrite dans [HP94] sur l'automate global). Cette comparaison est effectuée sur le cas d'étude présenté dans la section 5.1.

Le chapitre 6 étend la relation d'indépendance mono-processus au cas multi-processus. En effet, les communications inter-processus peuvent interférer dans le calcul des relations d'indépendance. Une nouvelle procédure de décision est décrite pour les calculer. L'algorithme de la procédure de décision est décrit dans la section 5. Les expérimentations sont effectuées sur la nouvelle version multi-processus du contrôleur du cas d'étude introduit dans la section 5.1.

Chapitre 4

Cadre formel

Sommaire

Résumé. Ce mémoire s'intéresse à définir une relation d'équivalence entre les scénarios d'un environnement (contexte formalisé) décrit dans le langage CDL [DADB+08], à partir des informations issues non seulement du système modélisé dans le langage SDL [SDL92], des propriétés (observateurs), mais aussi du contexte lui-même. Le modèle vérifié est donc composé de trois parties :

- le modèle du système lui même supposé décrit dans un sous-ensemble du langage SDL,
- le modèle de l'environnement, décrit dans le langage CDL,
- et enfin, l'expression des propriétés à vérifier, que l'on supposera décrite sous la forme d'observateurs, c'est-à-dire de machine à états observant le comportement du système et de son contexte, et produisant un événement *reject* lorsque la propriété est falsifiée.

Afin d'extraire une nouvelle relation d'indépendance entre les scénarios du contexte, nous introduisons dans ce chapitre le cadre formel nécessaire, en se restreignant à une sous-partie de l'expressivité du langage SDL et du langage CDL.

Tout d'abord, la syntaxe et la sémantique d'un système SDL sont décrites. Ensuite, la syntaxe et la sémantique d'un contexte CDL sont définies. La composition entre un système et son contexte est ensuite présentée, avant de terminer avec la formalisation des observateurs à vérifier.

4.1 SDL pour la modélisation de systèmes

4.1.1 Présentation du langage

SDL (*Specification and Description Language*) est un langage formel hiérarchique standardisé par l'organisme ITU-T Z.100. Il a d'abord été dédié à la spécification de protocoles de communication et étendu par la suite pour la modélisation de systèmes embarqués à événements discrets. SDL est un langage formel graphique décomposé en 4 vues : la vue hiérarchique, la vue communication, la vue comportementale et la vue donnée. Dans la suite, seule la vue comportementale est considérée, et un système SDL est restreint à un processus séquentiel. La sémantique de SDL repose sur des machines à états communiquant par échanges asynchrones d'événements. Le lecteur non familier avec le langage SDL peut se référer à [SDL92] pour une présentation détaillée du langage. Le mémoire se focalise sur un sous ensemble du langage SDL (mono processus, sans timers, et sans données).

4.1.2 Syntaxe et sémantique

Formalisation. Un modèle SDL est un système de transitions $\mathcal{S} = \langle \Sigma, s_0, T, Sig_{in}, Sig_{out}, Sv \rangle$ où :

- Σ est un ensemble fini d'états, et $s_0 \in \Sigma$ l'état initial de $\mathcal{S}$;
- Sig_{in} et Sig_{out} sont respectivement l'ensemble des événements d'entrée et de sortie de $\mathcal{S}$; Sig_{in} et Sig_{out} sont supposés disjoints ;
- $T \subseteq \Sigma \times Sig_{in} \times Sig_o^* \times \Sigma$ est l'ensemble des transitions ; une transition est un tuple (s_1, a, σ, s_2) où s_1 est l'état source de la transition, s_2 est l'état d'arrivée, a est l'événement de Sig_{in} déclenchant la transition (la transition est tirée si et seulement si le système est dans l'état s_1 et si a est reçu), et σ est la séquence d'événements de sortie émise lors du franchissement de la transition. (Sig_o^* représente l'ensemble des mots finis, c'est-à-dire des séquences, d'événements de sortie) ;
- $Sv : \Sigma \rightarrow 2^{Sig_{in}}$ est la fonction qui associe à chaque état $s \in \Sigma$ un ensemble (possiblement vide) d'événements d'entrée sauvegardés (i.e., mémorisés) s'ils sont reçus lorsque le système est dans l'état s.

Notons que si un événement e est reçu lorsque le système se trouve dans un état s alors que cet événement n'est déclencheur d'aucune transition quittant s, alors si $e \notin Sv(s)$, e est perdu (i.e., il n'est pas gardé dans la file des événements d'entrée). En revanche, si $e \in Sv(s)$, il est mémorisé dans la file d'entrée pour exploitation dans le prochain état.

Exemple. Considérons l'exemple de la figure 4.1 représentant un processus SDL gérant la communication entre un capteur et un serveur d'affichage. Le processus interagit avec 4 acteurs issus de son environnement. Les acteurs A_1 et A_2 permettent de changer le mode d'exécution du gestionnaire. L'acteur A_3 représente le serveur d'affichage et l'acteur A_4 représente un opérateur humain pouvant désactiver (respectivement activer) le gestionnaire. Les interactions entre le système et son environnement sont représentés sur la figure 4.2

Le gestionnaire peut se trouver dans deux modes différents (événements $mode_1$ et $mode_2$) ; dans le mode 1, à la suite d'une requête du gestionnaire de capteur (événement *req*), lorsque le système reçoit une donnée du capteur (événement *dcapt*), il l'envoie au serveur d'affichage (événement *data*). Dans le mode 2, deux requêtes successives sont effectuées auprès du capteur pour obtenir une nouvelle donnée et la renvoyer ensuite au serveur d'affichage. En reprenant la formalisation introduite ci-dessus, ce processus SDL est défini par $\mathcal{S} = \langle \Sigma, \mathit{Wait}, T, Sig_{in}, Sig_{out}, Sv \rangle$ avec :

- $\Sigma = \{ \mathit{Wait}, \mathit{Sleep}, S_1, S_2 \}$,

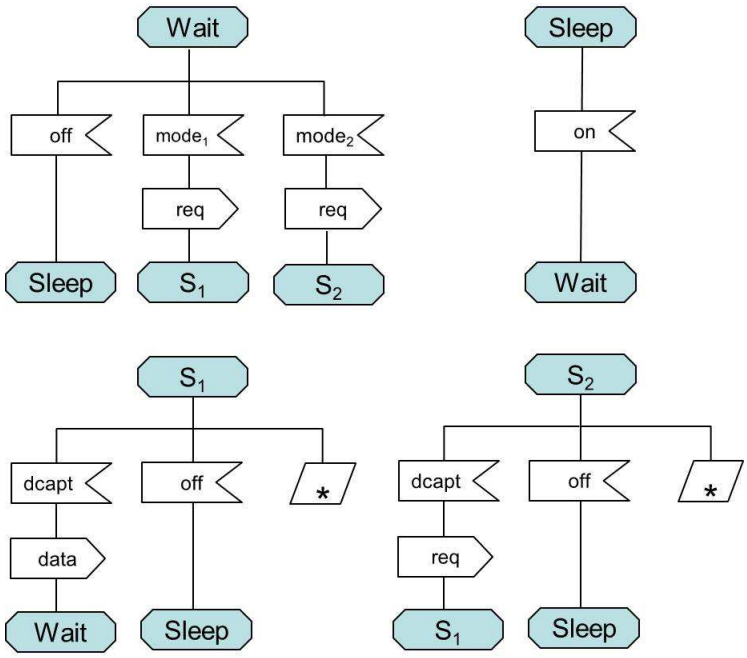


FIG. 4.1: Exemple de processus SDL : gestionnaire de capteurs

- $T = \{$ $(Wait,\ off,\ Sleep)$; $(Wait,\ mode_1,\ (req),\ S_1)$;
  $(Wait,\ mode_2,\ (req),\ S_2)$; $(Sleep,\ on,\ Wait)$;
  $(S_1,\ dcapt,\ (data),\ Wait)$; $(S_1,\ off,\ Sleep)$;
  $(S_2,\ dcapt,\ (req),\ S_1)$; $(S_2,\ off,\ Sleep)\}$
  est l'ensemble des 8 transitions représentées figure 4.1

- $Sig_{in} = \{mode_1, mode_2, dcapt,\ off,\ on\}$ les événements d'entrée du modèle : $mode_1$ (resp. $mode_2$) sont les deux modes de fonctionnement du gestionnaire de capteur correspondant respectivement à la récupération d'une donnée du capteur après une seule requête (respectivement deux requêtes)

- $Sig_{out} = \{req, data\}$ les événements envoyés par le gestionnaire au capteur pour la récupération d'une donnée (respectivement pour l'envoi de la donnée au serveur d'affichage)

- Sv la fonction de sauvegarde (mémorisation) qui associe l'ensemble vide à *Wait* (car aucun message n'est sauvegardé dans cet état), et $\{mode_1, mode_2, on\}$ à S_1 et S_2. En effet l'astérisque est une simplification signifiant tous les messages exceptés ceux qui peuvent être consommés dans l'état courant. Ici, tous les messages reçus à l'exception de *dcapt* et *off* (qui peuvent être consommés dans S_1 et S_2 sont mémorisés. Graphiquement, la fonction Sv est définie par un parallélogramme oblique associé à chaque état s (l'absence de parallélogramme signifie que $Sv(s) = \emptyset$.).

Exemple. La transition issue de l'état source *Wait* en direction de l'état destinataire S_1 est gardée par l'événement $mode_1$ et produit lorsqu'elle est franchie l'événement *req* (qui est envoyé ici à l'environnement). Notons que si *dcapt* est reçu lorsque le système est dans l'état *Wait*, il est perdu (car spécifié comme non mémorisé). En revanche, si $mode_1$ est reçu dans S_1, il est mémorisé pour être consommé dans l'état suivant (si cela est possible, sinon il est soit de nouveau sauvegardé si l'état le spécifie, soit perdu si il n'est pas mémorisé et non consommable).

Enfin, pour illustrer le point technique traité dans ce mémoire, considérons les scénarios $off \cdot on$

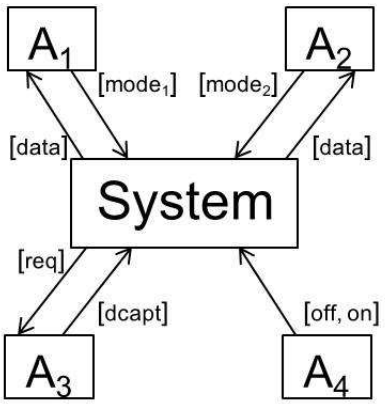


FIG. 4.2: Communication entre le système et son environnement

et $on \cdot off$ dans l'état S_1. D'un point de vue observationnel, ces deux scénarios ont le même effet sur le système à partir de l'état S_1. Ils conduisent dans le même état *Wait* sans produire aucun message de sortie. Nous pouvons alors considérer que les deux scénarios $off \cdot on$ et $on \cdot off$ sont équivalents.

Sémantique. Soit un automate SDL $\mathcal{S} = \langle \Sigma, init, T, Sig_{in}, Sig_{out}, Sv \rangle$. Par définition du langage SDL, $\mathcal{S}$ est un automate réagissant de façon asynchrone par rapport à son environnement, recevant ses événements d'entrée dans une file d'attente (appelé *buffer* ci-après), et consommant ces événements dans leur ordre d'arrivée. La sémantique, c'est-à-dire le comportement de $\mathcal{S}$, sera donc exprimée par une relation d'évolution sur le couple (s, B) où s est l'état courant de $\mathcal{S}$ et B son buffer d'entrée à l'instant courant. La sémantique de $\mathcal{S}$ est définie par une relation

$$(s, B) \ \overrightarrow{\sigma}_{\mathcal{S}} \ (s', B')$$

pour signifier que si $\mathcal{S}$ est dans l'état s et a pour buffer d'entrée B, alors il évolue en produisant le mot de sortie $\sigma \in Sig^*_{out}$ et en atteignant le nouvel état s' associé à un nouveau buffer B'.

Dans la suite, un buffer est un mot fini et ordonné de Sig^*_{in}. Par souci de simplicité, $null_\sigma$ désigne le buffer vide (i.e., le mot vide). $a \cdot B$ dénote le buffer (non vide) composé en tête de l'événement a puis du contenu du buffer B, et par abus de notation, Enfin $B_1 \cdot B_2$ représente le buffer composé de la concaténation des contenus de B_1 et B_2 (B_1 avant B_2). Cette relation sémantique de SDL est définie par induction par les règles suivantes :

- La règle *trans* exprime que si l'événement a est en tête du buffer courant, et si à partir de s (l'état courant) il existe une transition de $\mathcal{S}$ (la notation "pointée" $\mathcal{S}.T$ désigne le champ T (les transitions) de $\mathcal{S}$) franchissable par a, alors cette transition peut être franchie, la séquence σ d'événements de sortie de la transition est produite, l'état destination de la transition est atteint, et l'événement a est consommé.

$$\frac{(s, a, \sigma, s') \in \mathcal{S}.T}{(s, a \cdot B) \ \overrightarrow{\sigma}_{\mathcal{S}} \ (s', B)} \quad [\text{trans}]$$

- La règle $discard_S$ exprime que si l'événement a est en tête du buffer courant, et si à partir de s (l'état courant) il n'existe aucune transition franchissable par a, et si enfin a n'est pas spécifié comme mémorisé dans l'état s, alors a est perdu (i.e., effacé du buffer). L'état courant reste inchangé. Aucun événement de sortie n'est produit.

$$\frac{a \notin Sv(s) \quad \forall s', \forall \sigma, (s, a, \sigma, s') \notin \mathcal{S}.T}{(s, a \cdot B) \xrightarrow[null_\sigma]{} \mathcal{S} \quad (s, B)} \text{ [discard}_S\text{]}$$

- Enfin, la règle *save* exprime que si l'événement a est en tête du buffer courant, et si à partir de s (l'état courant) il n'existe aucune transition franchissable par a, et si enfin a est spécifié comme mémorisé dans l'état s, alors a reste en tête du buffer, mais le système évolue (s'il le peut) en exploitant les événements suivants du buffer.

$$\frac{a \in Sv(s) \quad (s, B) \xrightarrow[w]{} \mathcal{S} \quad (s', B') \quad \forall s', \forall \sigma, (s, a, \sigma, s') \notin \mathcal{S}.T}{(s, a \cdot B) \xrightarrow[w]{} \mathcal{S} \quad (s', a \cdot B')} \text{ [save]}$$

Dans la suite, $(s, B) \not\rightarrow_{\mathcal{S}}$ exprime le fait qu'aucune des règles ci-dessus ne peut s'appliquer sur $\mathcal{S}$ dans l'état s avec le buffer B, et par conséquent que le système n'évolue pas.

Exemple. Reprenons le système de la figure 4.1 (page 53). Supposons que le système se trouve dans l'état *Wait* et que le buffer contient les messages $mode_1$; $mode_2$; *off* qui ont été préalablement envoyés par le contexte. Si l'on applique la règle *trans*, alors l'événement $mode_1$ est consommé dans l'état $Wait$, le message de sortie *req* est produit pour être envoyé à l'environnement, l'état de destination S_1 est atteint, et le buffer contient à présent les messages $mode_2$; *off*.

Pour illustrer la règle *save*, continuons de consommer les messages restant du buffer (contenant les messages $mode_2$; *off*) du processus SDL. Dans l'état S_1, $mode_2$ ne peut pas être consommé, mais il est sauvegardé (l'astérisque sauvegarde tous les messages non consommables dans l'état courant). Par conséquent, $mode_2$ est mis à la tête du buffer, et les messages restant dans le buffer sont traités (ici *off*). L'état d'arrivée n'a pas bougé car aucun message n'a pu être consommé. Le système reste donc dans l'état S_1, et le buffer contient désormais le message $mode_2$ qui a été sauvegardé et le message *off* qui reste à traiter. Le buffer contient donc les messages $mode_2$; *off*.

Illustrons finalement la règle *discard*. Dans l'état S_1, le buffer contient des messages qui n'ont pas encore été traités hormis ceux qui sont sauvegardés. C'est le cas du message *off*. Il est consommable dans l'état S_1, amenant ainsi le système dans l'état *Sleep* avec le buffer contenant à présent : $mode_2$. Ce dernier est proposé pour être consommé dans l'état *Sleep*. Or, il n'est pas consommable (il n'est présent dans aucune transition de *Sleep*). De plus aucune mémorisation n'est effectuée dans l'état *Sleep*. Par conséquent, le message est perdu et le buffer ne contient plus aucun élément. L'état atteint reste l'état *Sleep* car aucune transition n'a permis au système d'évoluer.

4.2 CDL pour la modélisation de contextes

Le système global considéré dans ce mémoire est formé du système lui-même, de son environnement et des propriétés à vérifier. Pour modéliser l'environnement, le langage CDL est utilisé.

4.2.1 Présentation du langage

Le langage CDL [DADB+08, DBL08, DRB07, DPC+09] dérive du formalisme des cas d'utilisation (Uses Cases) et des travaux de [Whi05]. Il permet de décrire formellement un environnement d'un système en structurant et reliant un ensemble de diagrammes de séquences (MSCs) à l'aide de trois opérateurs : la séquence, le parallélisme et l'alternative. Une fois le contexte décrit par un ensemble de MSCs, il est déplié et partitionné de telle manière à générer un ensemble de scénarios à

composer avec le système et les propriétés à vérifier : les observateurs. Ces observateurs sont décrits dans une syntaxe particulière et dérivent des travaux de [DAC98]. Ce langage a des similitudes avec les **HMSC** [HMS96] et les étend dans plusieurs directions : d'abord il permet l'ajout de compteur sur le MSC permettant ainsi un partitionnement de scénarios finis. Il permet aussi de rattacher des propriétés sur des MSCs spécifiques de manière à optimiser les coûts de vérification. Dans ce mémoire les contextes sont modélisés à l'aide d'une sous-partie du langage CDL suffisante pour la description de l'environnement des systèmes étudiés. En particulier, ni les compteurs, ni les gardes CDL ne seront manipulés.

4.2.2 Syntaxe et sémantique

Le langage CDL possède trois types d'opérateurs simples à manipuler et permettant de structurer l'expression d'un environnement par combinaison de ces derniers.

- L'opérateur de **séquence** permet l'exécution successive de messages d'envoi ou de réception par un même acteur.

  Exemple. Sur la figure 4.3a, les envois de $mode_1$ et l'attente de *data* sont exécutés par l'acteur A_1. A noter que ces messages ne sont pas exécutées atomiquement pour permettre l'entrelacement exhaustif de scénarios. Sur ce contexte, l'acteur A_3 exécute en séquence deux MSC contenant chacun deux messages en séquence.

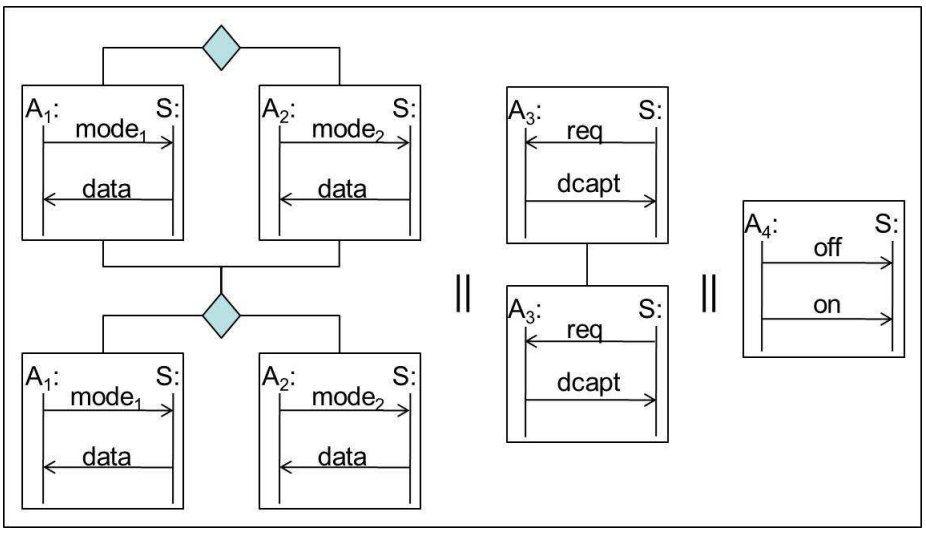
- L'opérateur de **parallélisme** symbolisé par $\|$ implante la sémantique de l'entrelacement dans le contexte. C'est lui qui permet de générer exhaustivement l'ensemble des scénarios d'un contexte. Sur la figure 4.3a, les acteurs A_1 et A_3 ou encore A_2 et A_3 peuvent s'exécuter en parallèle. Ainsi par exemple le message $mode_1$ pourra être envoyé au système avant ou après la réception et l'envoi des messages *req* et *dcapt* par A_3. Ainsi $mode_1$ pourra être envoyé après la réception du deuxième message *req* reçu par A_3.

- L'opérateur **alternative** est une clause CDL permettant de parcourir toutes les branches d'un point de décision. La différence majeure entre la décision (choix) et l'alternative, est que dans le cas d'un choix, une branche seulement est parcourue tandis que dans le cas de l'alternative, toutes les branches seront parcourues. En effet, le contexte a pour objectif d'être déplié en un ensemble de scénarios. Par conséquent, l'alternative générera autant de scénarios que de branches présentes. Il s'agit d'une facilité d'écriture. Sur la figure 4.3a, considérons uniquement la partie gauche du contexte constitué des acteurs A_1 et A_2 qui sont en "alternative". Cette alternative est effectuée en séquence une deuxième fois. Par conséquent l'ensemble des scénarios est :

  $mode_1 \cdot data \cdot mode_1 \cdot data$
  $mode_1 \cdot data \cdot mode_2 \cdot data$
  $mode_2 \cdot data \cdot mode_1 \cdot data$
  $mode_2 \cdot data \cdot mode_2 \cdot data$

  Notons que cette construction n'est pas exprimable en particulier dans le langage Promela du *model-checker* SPIN car les conditions et les décisions sont toujours déterministes.

Considérons maintenant la fermeture du système par un environnement. Ce mémoire reprend (en la restreignant) la notion de contexte introduite par [DPC+09, Rog06]. Un contexte définit l'ensemble des comportements valides (ou un sous-ensemble si les propriétés à vérifier ne concernent qu'une partie de l'espace de ses comportements) de l'environnement du système. Un contexte est décrit sous la forme d'une composition séquentielle, parallèle ou alternative, de diagrammes de séquences (appelés ci-après MSC pour Message Sequence Chart).



(a) Contexte du gestionnaire

$$A_1 = mode_1!; data?; \mathbf{0}$$
$$A_2 = mode_2!; data?; \mathbf{0}$$
$$A_3 = req?; dcapt!; \mathbf{0}$$
$$A_4 = off!\,;\, on!; \mathbf{0}$$

(b) Description formelle du contexte

FIG. 4.3: Exemple d'un contexte CDL : gestionnaire de capteur

Formalisation. Formellement, un contexte est un processus fini produisant et recevant des événements décrits par la grammaire suivante :

$$\begin{array}{lcll} C & ::= & M \mid C_1.C_2 \mid C_1 + C_2 \mid C_1 \| C_2 & \\ M & ::= & \mathbf{0} \mid a!; M \mid a?; M & \text{avec } a \in \text{un alphabet d'événements} \end{array}$$

Un contexte est soit un MSC M, soit une composition séquentielle de deux contextes $(C_1.C_2)$, soit une alternative entre deux contexte $(C_1 + C_2)$, soit enfin la composition parallèle de deux contextes $(C_1 \| C_2)$. Un MSC est soit le MSC terminal qui ne fait plus rien $(\mathbf{0})$, ou la succession d'émissions $a!$ et d'attentes $a?$ d'événements. Considérons la fonction $Voc(C)$ retournant le vocabulaire du contexte C (les événements émis et reçus par C) :

$$Voc(\mathbf{0}) \stackrel{\text{def}}{=} \emptyset \qquad Voc(C_1.C_2) \stackrel{\text{def}}{=} Voc(C_1) \cup Voc(C_2)$$
$$Voc(a!; M) \stackrel{\text{def}}{=} \{a\} \cup Voc(M) \qquad Voc(C_1||C_2) \stackrel{\text{def}}{=} Voc(C_1) \cup Voc(C_2)$$
$$Voc(a?; M) \stackrel{\text{def}}{=} \{a\} \cup Voc(M) \qquad Voc(C_1 + C_2) \stackrel{\text{def}}{=} Voc(C_1) \cup Voc(C_2)$$

Exemple. Considérons le contexte C de la figure 4.3a. Ce contexte décrit un environnement du système présenté figure 4.1 (page 53). Ce contexte est composé de deux acteurs en parallèle (à noter que A_1 et A_2 sont en fait un seul et même acteur. Un serveur d'affichage A_1 (A_2) et un capteur A_3. L'acteur A_1 (A_2) envoie différents modes de fonctionnement au système de gestion de capteur et attend en retour une donnée. L'alternative ici ne correspond pas à la sémantique classique d'un choix. Ici cette alternative signifie que l'on souhaite tester les deux éventualités à chaque fois. L'acteur A_3 quand à lui attend une requête du système avant de lui retourner une donnée. En reprenant la formalisation du contexte de la figure 4.3b, ce contexte s'écrit :

$$C = [(A_1 + A_2).(A_1 + A_2)] \parallel (A_3.A_3) \parallel A_4$$

Sémantique. Soit une fonction préliminaire $Wait(C)$ associant à tout contexte l'ensemble des événements attendus dans l'état courant du contexte, définie par induction par :

$$Wait(\mathbf{0}) \stackrel{\text{def}}{=} \emptyset \qquad Wait(a!;M) \stackrel{\text{def}}{=} \emptyset \qquad Wait(C_1 + C_2) \stackrel{\text{def}}{=} Wait(C_1) \cup Wait(C_2)$$
$$Wait(a?;M) \stackrel{\text{def}}{=} \{a\} \qquad Wait(C_1.C_2) \stackrel{\text{def}}{=} Wait(C_1) \qquad Wait(C_1 \| C_2) \stackrel{\text{def}}{=} Wait(C_1) \cup Wait(C_2)$$

Un contexte peut être assimilé à un processus communicant de façon asynchrone avec le système mémorisant ses événements d'entrée (reçus du système) dans un buffer. La sémantique du langage CDL est définie par une relation

$$(C, B) \xrightarrow{a} (C', B')$$

pour exprimer que le contexte C associé à un buffer B (une file d'attente d'événements émis par le système à destination de son environnement) "fait l'action" a (qui peut être une émission ou une réception, voire l'événement $null_\sigma$ si C n'évolue pas) avant de devenir le nouveau contexte C' soumis à un nouveau buffer B'. Cette relation est définie par les règles suivantes (dans l'ensemble de ces règles, a représente un événement différent de $null_\sigma$) :

- règle *pref1* (sans pré-condition) : un MSC commençant par une émission $a!$ émet cet événement puis poursuit le reste du MSC.

$$\frac{}{(a!;M, B) \xrightarrow{a!} (M, B)} \quad [\text{pref1}]$$

- règle *pref2* (sans pré-condition) : un MSC commençant par une réception $a?$ et confronté à un buffer d'entrée contenant en tête cet événement a consomme cet événement puis poursuit le reste du MSC.

$$\frac{}{(a?;M, a.B) \xrightarrow{a?} (M, B)} \quad [\text{pref2}]$$

- règle *seq1* : une séquence $C_1.C_2$ se comporte comme C_1 tant que celui-ci n'est pas terminé.

$$\frac{C_1' \neq \mathbf{0} \quad (C_1, B) \xrightarrow{a} (C_1', B')}{(C_1.C_2, B) \xrightarrow{a} (C_1'.C_2, B')} \quad [\text{seq1}]$$

- règle *seq2* : en revanche, dès que C_1 s'achève (devient $\mathbf{0}$), la séquence devient C_2.

$$\frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{(C_1.C_2, B) \xrightarrow{a} (C_2, B')} \quad [\text{seq2}]$$

- règle *alt* : le choix entre deux contextes $C_1 + C_2$ se comporte soit comme C_1, soit comme C_2. Mais une fois le choix fait, celui-ci est définitif (notons que cette règle a deux conclusions).

$$\frac{(C_1, B) \xrightarrow{a} (C_1', B')}{\begin{array}{l}(C_1 + C_2, B) \xrightarrow{a} (C_1', B') \\ (C_2 + C_1, B) \xrightarrow{a} (C_1', B')\end{array}} \quad [\text{alt}]$$

- règle *par1* : le comportement de $C_1 \| C_2$ est l'entrelacement non déterministe du comportement des deux contextes tant que ceux-ci ne sont pas terminés. Notons que cette composition parallèle est asynchrone au sens où les deux contextes n'évoluent pas simultanément mais l'un après l'autre.

$$\frac{\begin{array}{c} C_1' \neq \mathbf{0} \\ (C_1, B) \xrightarrow{a} (C_1', B') \end{array}}{\begin{array}{l} (C_1 \| C_2, B) \xrightarrow{a} (C_1' \| C_2, B') \\ (C_2 \| C_1, B) \xrightarrow{a} (C_2 \| C_1', B') \end{array}} \text{[par1]}$$

- règle *par2* : si l'un des deux contextes termine (devient $\mathbf{0}$), alors la composition parallèle devient le second contexte (celui qui n'est pas terminé).

$$\frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{\begin{array}{l} (C_1 \| C_2, B) \xrightarrow{a} (C_2, B') \\ (C_2 \| C_1, B) \xrightarrow{a} (C_2, B') \end{array}} \text{[par2]}$$

- règle $discard_C$: enfin si un événement a en tête du buffer d'entrée du contexte n'est pas attendu par celui-ci, alors il est perdu (supprimé de la tête du buffer).

$$\frac{a \notin Wait(C)}{(C, a.B) \xrightarrow{null_\sigma} (C, B)} \text{[discard}_C\text{]}$$

Notons que le caractère asynchrone de la composition parallèle (qui correspond à l'asynchronisme de l'environnement réel entourant le système) induit en pratique une explosion du nombre des traces possibles d'un contexte (ses scénarios). Par exemple, le contexte figure 4.3a (page 57) décrit 18900 scénarios différents.

4.3 Composition du contexte avec le système

On définit maintenant la fermeture d'un système par un contexte. Soit un système $\mathcal{S}$ et un contexte C. L'évolution de $\mathcal{S}$ fermé par C est donnée par la relation :

$$< (C, B_1) | (s, B_2) > \xrightarrow[\sigma]{a}_{\mathcal{S}} < (C', B_1') | (s', B_2') >$$

pour exprimer que $\mathcal{S}$ dans l'état s associé à un buffer d'entrée B_2 et fermé par le contexte (C, B_1) évolue vers l'état s' en recevant l'événement a (produit par le contexte) et en produisant la séquence d'événement σ (vers le contexte). La fermeture du système par son contexte est définie par les règles suivantes :

- règle *cp1* : si $\mathcal{S}$ peut produire σ, alors $\mathcal{S}$ évolue et σ est placé en fin du buffer de C.

$$\frac{(s, B_2) \xrightarrow[\sigma]{}_{\mathcal{S}} (s', B_2')}{< (C, B_1) | (s, B_2) > \xrightarrow[\sigma]{null_e}_{\mathcal{S}} < (C, B_1.\sigma) | (s', B_2') >} \text{[cp1]}$$

- règle *cp2* : si C peut émettre a, C évolue et a est placé en fin du buffer de $\mathcal{S}$.

$$\frac{(C, B_1) \xrightarrow{a!} (C', B_1')}{< (C, B_1) | (s, B_2) > \xrightarrow[null_\sigma]{a}_{\mathcal{S}} < (C', B_1') | (s, B_2.a) >} \text{[cp2]}$$

- règle *cp3* : enfin, si C peut consommer a, alors il évolue tandis que $\mathcal{S}$ reste inchangé.

$$\frac{(C, B_1) \xrightarrow{a?} (C', B_1')}{< (C, B_1)|(s, B_2) > \xrightarrow[null_\sigma]{null_e} \mathcal{S} \quad < (C', B_1')|(s, B_2) >} \text{[cp3]}$$

Notons que la composition de fermeture entre un système et son contexte est assimilable à une composition parallèle asynchrone : les comportements de C et de $\mathcal{S}$ sont entrelacés, et la communication est réalisée par échanges asynchrones via des buffers. On notera

$$< (C, B)|(s, B') > \not\rightarrow_{\mathcal{S}}$$

pour exprimer que le système et son contexte, associés à leur buffer respectif, ne peuvent plus évoluer selon les règles ci-dessus (le système fermé est bloqué ou le contexte est terminé). On définit alors l'ensemble des traces (appelées des *runs*) du système fermé par son contexte et à partir d'un état s, par la fonction :

$$\begin{aligned}[\![C \mid (s, \mathcal{S})]\!] \stackrel{\text{def}}{=} \{a_1 \cdot \sigma_1 \cdot \ldots a_n \cdot \sigma_n \cdot end_C \mid \quad & < (C, null_\sigma) \mid (s, null_\sigma) > \quad \xrightarrow[\sigma_1]{a_1} \mathcal{S} \\ & < (C_1, B_1) \mid (s_1, B_1') > \quad \xrightarrow[\sigma_2]{a_2} \mathcal{S} \quad \ldots \quad \xrightarrow[\sigma_n]{a_n} \mathcal{S} \\ & < (C_n, B_n) \mid (s_n, B_n') > \quad \not\rightarrow_{\mathcal{S}} \quad \}\end{aligned}$$

$[\![C|(s, \mathcal{S})]\!]$ est l'ensemble des exécutions du système $\mathcal{S}$ fermé par le contexte C et en considérant s comme l'état de départ de $\mathcal{S}$.

Notons que, les contextes étant formés de compositions séquentielles ou parallèles de MSC finis, les traces des contextes sont donc nécessairement finies. En conséquence les *runs* du système fermé par son contexte sont nécessairement finis. Chaque *run* de $C|(s, \mathcal{S})$ est étendu par l'événement terminal end_C permettant à l'observateur d'observer la fin d'un scénario et de "tester" d'éventuelles propriétés de vivacités (une propriété du type "un jour a" se traduira sous cette restriction de finitude des contextes par "un jour a avant end_C").

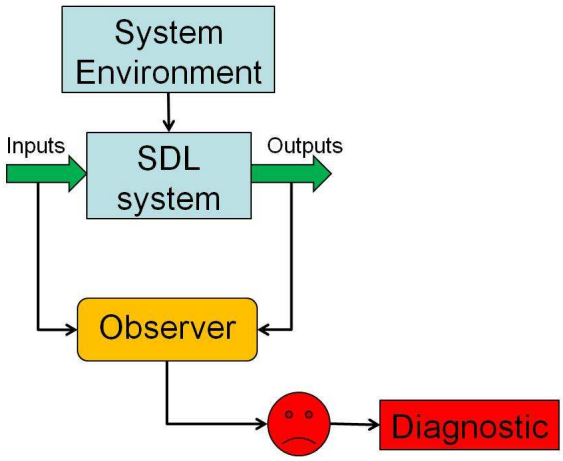
4.4 Les observateurs

Dans le langage CDL, les observateurs permettent d'encoder des propriétés formelles dans le but de valider le système en cours de vérification. L'approche par rapport aux logiques temporelles est différente dans le sens où l'observateur observe les entrées, les sorties du système et retourne une erreur dès lors qu'un état spécial dit de rejet est atteint (figure 4.4a).

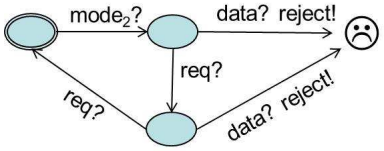
Parmi les différents types d'observateurs présents dans la littérature, Nous pouvons citer par exemple les automates de Büchi utilisés pour la vérification de logiques temporelles linéaires notamment pour la vérification de propriétés de vivacité, ou encore les automates de tests [ABL98] utilisés pour la vérification de propriétés de sûreté. On peut aussi citer un exemple d'utilisation d'observateurs intégrés dans un outil industriel Object Géode [Dol03] pour la modélisation et simulation de modèles SDL. Ces observateurs sont modélisés en SDL écrits avec le langage spécifique GOAL [Obe99].

Par rapport au logiques temporelles, les observateurs ont un pouvoir d'expression plus restreint. Ils sont plus simples d'utilisation. Les observateurs peuvent exprimer les propriétés d'accessibilité qui englobent les propriétés de sûreté et de vivacité bornée. Un point important sur les observateurs est leur caractéristique non intrusive ; c'est à dire qu'ils n'ont pas la possibilité de modifier le comportement du système en cours d'observation mais uniquement observer les dysfonctionnements pour générer un retour de diagnostic.

Les lecteurs intéressés par un état de l'art sur les observateurs pourront s'orienter sur la thèse de Jean Charles Roger [Rog06].



(a) Principe d'un observateur



(b) Observateur Obs_2

FIG. 4.4: Les observateurs

Exemple. Reprenons le système figure 4.1 (page 53), fermé par le contexte figure 4.3a (page 57). Considérons la propriété P_2 :

• P_2 : si je reçois *mode2* je ne pourrai envoyer une donnée *data* uniquement si deux requêtes *req* ont été reçues par le serveur d'affichage.

Cette propriété est modélisée par l'observateur de la figure 4.4b. En effet, si le message *data* est reçu avant les deux requêtes *req*, alors le message de falsification *reject !* est envoyé par l'observateur pour signifier l'échec de la vérification.

Formalisation.

D'un point de vue formel, un observateur est un automate SDL $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Sig, \{reject\}, Sv_o \rangle$

- n'émettant qu'un seul événement de sortie : *reject*,
- où Sig est l'ensemble des événements observés par l'observateur, c'est-à-dire produits et reçus par le système et son contexte
- et tel que toutes les transitions étiquetées par la production de *reject* arrivent dans un unique état puit (i.e., sans successeurs) appelé **"unhappy"**.

Sémantique.

Soit un système $\mathcal{S}$ et un contexte C. Soit un observateur $\mathcal{O}$. $\mathcal{S}$ dans l'état $s \in \Sigma$ fermé par C satisfait $\mathcal{O}$, noté $C|(s,\mathcal{S}) \models \mathcal{O}$, si et seulement si aucune exécution de $\mathcal{O}$ confrontée aux runs r de $[\![C|(s,\mathcal{S})]\!]$ ne produit l'événement *reject*. C'est-à-dire :

$$C|(s,\mathcal{S}) \models \mathcal{O} \Leftrightarrow \begin{cases} \forall r \in [\![C|(s,\mathcal{S})]\!], \\ (init_o, r) \xrightarrow{null_\sigma}_{\mathcal{O}} (s_1, r_1) \xrightarrow{null_\sigma}_{\mathcal{O}} \cdots \xrightarrow{null_\sigma}_{\mathcal{O}} (s_n, r_n) \not\rightarrow_{\mathcal{O}} \end{cases}$$

Remarque. exécuter $\mathcal{O}$ sur un run r de $C|(s,\mathcal{S})$ revient simplement à remplir le buffer de $\mathcal{O}$ par r, et dérouler son exécution à partir de son état initial. Si la propriété est satisfaite, alors cette exécution n'émettra que des événements vides ($null_\sigma$) (et donc jamais *reject*).

Chapitre 5

Application de la méthode de réduction par ordres-partiels aux systèmes SDL mono-processus

Sommaire

Résumé. Ce chapitre introduit une nouvelle relation d'indépendance entre les événements du contexte, à partir des informations du système, du contexte et des observateurs. Cette relation d'indépendance se distingue des méthodes proposées dans la littérature sur le fait que les relations d'indépendances sont calculées sans la génération explicite de l'automate global. Le mémoire se concentre tout d'abord à déterminer une relation d'indépendance dans le cas de système SDL mono-processus. Cette relation sera étendue dans le chapitre suivant.

Dans un souci de faciliter la compréhension du chapitre et afin d'illustrer les définitions formelles introduites, nous proposons tout d'abord d'étudier un système mono-processus d'un contrôleur d'un passage à niveau, en reprenant les notations formelles du langage SDL, CDL et des observateurs pour décrire son fonctionnement.

Ensuite nous présentons formellement la nouvelle relation d'indépendance (section 5.3). A partir De cette relation d'indépendance, les équivalences de scénarios peuvent être calculées par la méthode des formes normales de Foata (section 5.4). Les algorithmes de calcul des relations d'indépendances sont également introduits.

Dans la section 5.5, nous verrons que les scénarios réduits obtenus ne sont pas suffisants pour effectuer la vérification sur un système et avec un observateur, du fait de son incomplétude. Nous verrons comment y remédier et de quelle manière la reconstruction de scénarios exécutables peut réduire le coût de la vérification en les regroupant sous forme de graphes de scénarios.

Enfin, nous expérimentons la nouvelle relation d'indépendance sur le système du passage à niveau dans la section 5.6.2, en utilisant le *model-checker* SPIN de deux manière différentes : tout d'abord par la **méthode classique** du *model-checking*, et ensuite la méthode proposée dans la thèse (vérification sur chaque scénario réduit composé avec le système et l'observateur appelée **méthode OBP**). Afin de comparer les performances de SPIN avec la méthode que nous proposons, nous complexifions incrémentalement le cas d'étude 1 (section 5.6.3) pour confronter les limites des deux approches.

La section 5.3 a été publié dans l'article [DBDB10b]. La section 5.5 a été publiée dans l'article [DBDB]

5.1 Cas d'étude 1 : un contrôleur de passage à niveau

5.1.1 Présentation générale

Afin d'illustrer la méthode présentée dans ce chapitre, considérons un cas d'étude simple : un contrôleur de passage à niveau.

Description du système. Sur la figure 5.1 un passage à niveau est représenté. Le système est constitué de deux voies, deux barrières avec un signal lumineux, un contrôleur *ctl* gérant la levée et descente des barrières, un opérateur humain pouvant ordonner la levée et la descente des barrières ainsi qu'une horloge envoyant périodiquement des tops d'horloges.

Fonctionnement du système. Le système fonctionne de la manière suivante. Initialement les barrières sont fermées. Lorsqu'un train arrive sur la voie 1 (respectivement voie 2), alors il envoie le signal e_1 (respectivement e_2) pour annoncer son arrivée dans la zone du passage à niveau. Dès lors, le contrôleur envoie le signal de descente **des deux barrières** par le signal *down* ainsi que le signal de clignotement **des lampes** avec le signal *light_on*. Une fois que le train a quitté le passage à niveau, alors un signal s_1 (respectivement s_2) annonçant son départ est envoyé au contrôleur. Par la suite après avoir reçu un top d'horloge par le signal *tick*, les barrières se lèvent sur réception du signal *up* envoyé par le contrôleur. A noter qu'un opérateur humain peut contrôler manuellement la levée et la descente des barrières, par exemple pour des raison de maintenance.

Dans la suite, nous souhaitons vérifier des propriétés sur le contrôleur modélisé en langage SDL par un unique processus. Le contrôleur communique avec un environnement bien défini :

- l'opérateur humain peut demander des ordres d'ouverture et fermeture des barrières

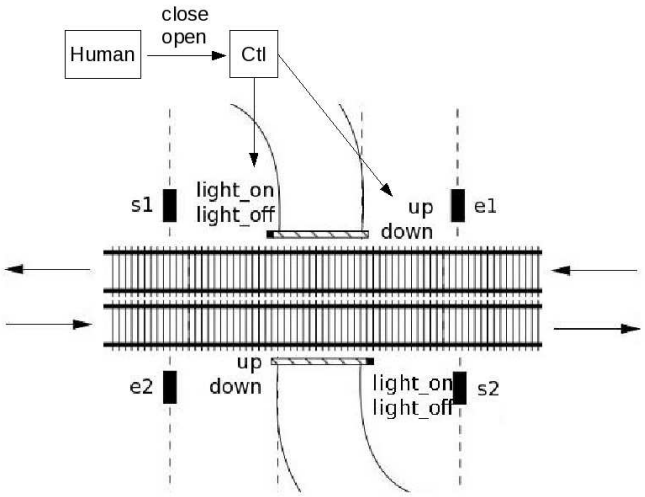


FIG. 5.1: Système de contrôle d'un passage à niveau

- les barrières peuvent recevoir des ordres d'ouverture et de fermeture de la part du contrôleur ; ainsi que des ordres sur l'allumage et l'extinction des feux.
- les trains de la voie 1 (respectivement de la voie 2) peuvent envoyer un signal d'entrée ou de sortie.
- une horloge envoie régulièrement un signal de temporisation *tick*.

Dynamique du système. Le passage à niveau considère deux trains ; un train sur chaque voie. Lorsqu'un train arrive dans la zone du passage à niveau, alors les barrières de chaque côté de la voie s'abaissent, en faisant clignoter les feux lumineux qui s'éteignent après abaissement complet des barrières. Au contraire, lorsque le train sort de la zone du passage à niveau alors les barrières peuvent à nouveaux se lever à condition qu'aucun train n'arrive sur l'une des deux voies. De plus, lorsque l'opérateur humain donne l'ordre d'abaisser les barrières, celles-ci ne pourront se lever que si l'ordre leur est donné à nouveau par cet opérateur. De la même manière lorsqu'aucun train n'est détecté sur les deux voies, les barrières devront être levées avec les lumières éteintes.

5.1.2 Formalisation

Modélisation SDL. La figure 5.2 représente le processus SDL implantant le contrôleur du passage à niveau *ctl* et gérant la fermeture et l'ouverture des barrières (événements *down* et *up*), ainsi que l'allumage et l'extinction des feux (événements *light_on* et *light_off*).

En reprenant la formalisation du chapitre 4.1 (page 52), ce processus SDL est défini par $\mathcal{S} = \langle \Sigma, idle, T, Sig_{in}, Sig_{out}, Sv \rangle$ avec :

- $\Sigma = \{idle, busy_1, busy_2, busy_{12}, safe, tempo\}$.
- T est l'ensemble des 14 transitions représentées figure 5.2.
- $Sig_{in} = \{e_1, s_1, e_2, s_2, close, open, tick\}$ les événements d'entrée du modèle : e_i (resp. s_i) est l'événement émis par un train lorsqu'il rentre dans le (resp. sort du) passage à niveau sur la voie i pour $i = 1, 2$; *close* est l'ordre de fermeture impérative des barrières émis par l'opérateur humain ; enfin *open* est l'ordre manuel d'ouverture des barrières après une fermeture par *close*.

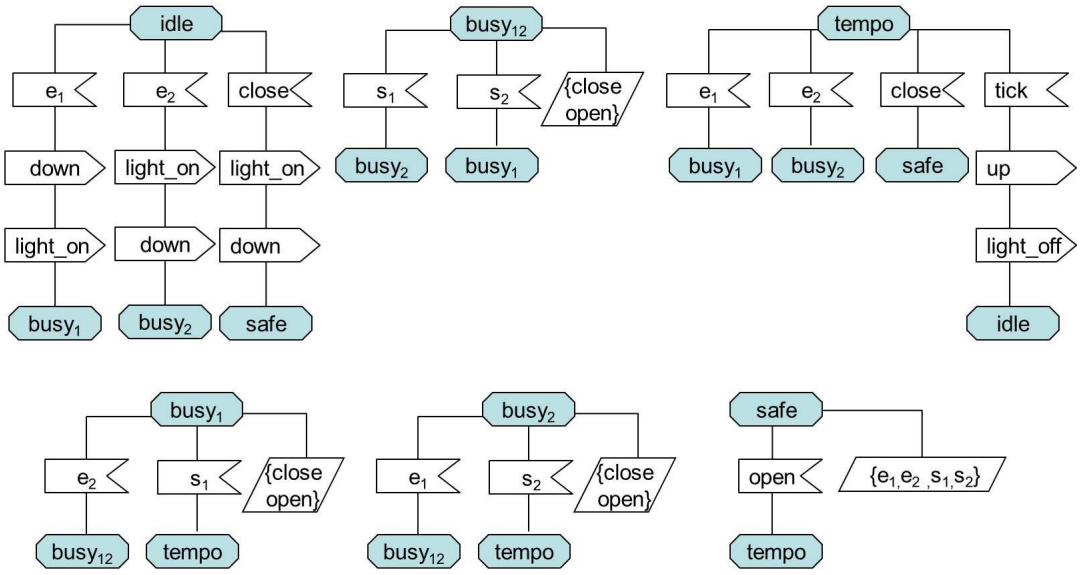


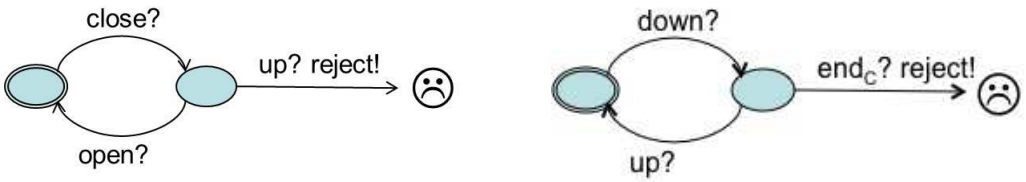
FIG. 5.2: Exemple de processus SDL : contrôle d'un passage à niveau

- $Sig_{out} = \{down, up, light\_on, light\_off\}$ les événements commandant les barrières et les feux.
- Sv la fonction de sauvegarde (mémorisation) qui associe l'ensemble vide aux états *idle* et *tempo*, et qui associe $\{open, close\}$ à $busy_1$, $busy_2$, et $busy_{12}$, ainsi que $\{e_1, e_2, s_1, s_2\}$ à *safe* (graphiquement, la fonction Sv est définie par un parallélogramme oblique associé à chaque état s ; l'absence de parallélogramme signifie que $Sv(s) = \emptyset$.

Modélisation des observateurs. La vérification du système mono-processus présenté ci-dessus, porte sur les deux propriétés suivantes :

- P_3 : après un ordre *close* (envoyé par le contexte), il ne peut y avoir par ordre d'ouverture *up* (décidé par le système) sans que le contexte ait auparavant fait *open* ;
- P_4 : à la fin du contexte, les barrières sont ouvertes.

Ces deux propriétés peuvent être modélisées par les deux observateurs figure 5.3a et 5.3b.



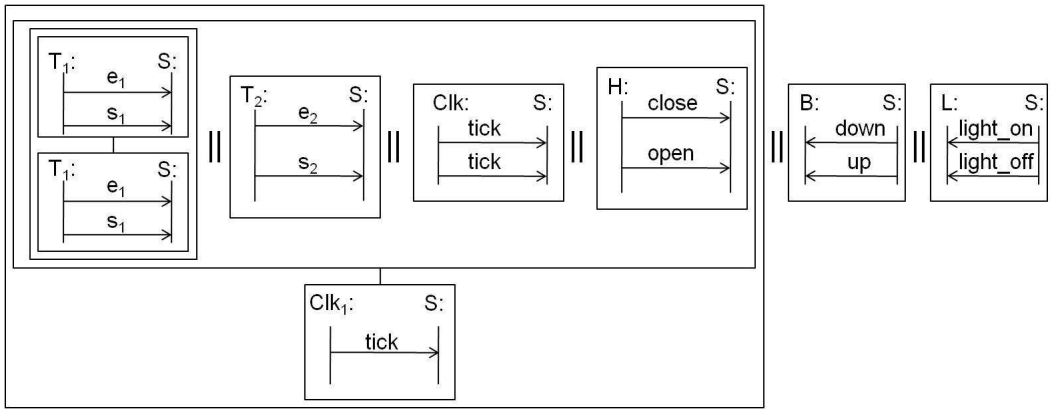
(a) Observateur Obs_3 de P_3 (b) Observateur Obs_4 de P_4

FIG. 5.3: Observateurs

Formellement, l'observateur Obs_3 de la figure 5.3a exprime que si le message *up* est "reçu" (c'est à dire émis par le système), après la réception par le système du message *close* et avant la réception

par le système du message *open*, alors l'événement *reject* est émis, et l'état *"unhappy"* est atteint (signifiant que la propriété est falsifiée).

De la même manière, l'observateur Obs_4 exprime que si le message *endC* (signifiant la fin de l'exécution du contexte) est reçu (émis par le système), après la réception par le contexte du message *down* et avant la réception du message *up*, alors l'événement *reject* est émis, et l'état *"unhappy"* est atteint.



(a) Contexte d'un passage à niveau

$T_1 = e_1!; s_1!; \mathbf{0}$ $\quad H = close!; open!; \mathbf{0}$
$T_2 = e_2!; s_2!; \mathbf{0}$ $\quad B = down?; up?; \mathbf{0}$
$Clk = tick!; tick!; \mathbf{0}$ $\quad L = light\_on?; light\_off?; \mathbf{0}$
$Clk_1 = tick!; \mathbf{0}$

(b) Description formelle du contexte

FIG. 5.4: Exemple d'un contexte CDL : arrivée de trains dans un passage à niveau

Modélisation du contexte. Considérons le contexte C de la figure 5.4a. Ce contexte décrit l'environnement sous lequel nous souhaitons vérifier les observateurs Obs_3 et Obs_4 sur le système présenté figure 5.2 (page 66). Ce contexte est composé de deux trains successifs sur la voie 1 et un unique train sur la voie 2 (T_1 puis à nouveau T_1 sur la voie 1, et en parallèle T_2 sur la voie 2), un opérateur humain (H) envoyant en séquence un ordre de fermeture puis un ordre d'ouverture, et enfin une horloge battant 2 *tick*. En parallèle, le contexte peut recevoir des ordres concernant la fermeture puis l'ouverture des barrières mais aussi l'allumage et l'extinction des feux. Le contexte envoie finalement le signal *tick* au système. En reprenant la formalisation du contexte de la figure 5.4a ce contexte s'écrit en CDL :

$$C = (((T_1.T_1) \| T_2 \| Clk \| H).Clk_1) \| B \| L$$

où chaque acteur T_1, T_2, H, Clk, B et L est décrit par une expression CDL sur la figure 5.4b.

5.2 Une relation d'indépendance entre événements : principes généraux

Comme mentionné en introduction, la principale difficulté rencontrée dans la vérification de systèmes de taille industrielle est l'explosion combinatoire du nombre des états. Une solution proposée par [DPC+09, Rog06], et rappelée en introduction section 1.2 à 1.4, consiste à circonscrire cette explosion en fermant le système à vérifier par un modèle de son contexte opérationnel. Si cette approche conduit généralement à une diminution sensible de l'espace des états à explorer, elle ne suffit pas toujours à rendre la vérification faisable pour des systèmes de grandes tailles (ou avec des

contextes peu limitant). J. C. Roger a alors proposé une approche complémentaire qui consiste à partitionner les contextes en des ensembles de scénarios. Ces scénarios sont des automates linéaires qui sont séparément composés avec le système et la propriété à vérifier.

Comme précisé en introduction, l'objectif de ce mémoire est d'étudier une méthode de réduction de l'espace des états reposant sur l'exploitation des symétries du modèle et de son environnement. Ces symétries résultent de relation de commutativité (ou d'indépendance) entre des signaux émis par le contexte à destination du système. Considérons à nouveau le système de gestion d'un passage à niveau figure 5.2 (page 66) ainsi que son contexte figure 5.4a et les deux propriétés figures 5.3a et 5.3b. Ce système présente à l'évidence certaines symétries. Considérons en particulier, les deux séquences d'événements $e_1 \cdot e_2$ et $e_2 \cdot e_1$. Quelque soit l'état dans lequel se trouve le système, l'enchaînement de ces deux événements conduit le système dans le même état. Par exemple, à partir de *idle*, ces deux scénarios mènent tous les deux dans $busy_{12}$. Ils produisent en revanche les mêmes événements de sortie (*down* et *light_on*) mais dans un ordre différent. Or, ces événements étant consommés dans l'environnement par des MSC différents et s'exécutant en parallèle, et d'autre part l'événement *light_on* n'impactant pas les observateurs Obs_3 et Obs_4, les deux traces de sorties *down*·*light_on* et *light_on*·*down* seront considérées comme équivalentes du point de vue du contexte et des observateurs. En conséquence, les événements e_1 et e_2 commutent (ils sont indépendants), c'est-à-dire que leur ordre d'occurrence n'a pas d'impact sur le comportement du système, de son contexte et sur la vérification des observateurs. La vérification des observateurs Obs_3 et Obs_4 pourra donc faire abstraction des scénarios d'exécution contenant la séquence $e_2 \cdot e_1$ pour ne garder que les scénarios similaires contenant $e_1 \cdot e_2$.

Notons, et c'est là le point important, que ce rapide raisonnement pour démontrer que e_1 et e_2 commutent n'a pas nécessité l'exploration du graphe des états du système composé avec son environnement (graphe qui dans les cas réalistes explose très vite), mais uniquement l'analyse du modèle SDL du système, du modèle CDL de son contexte, et du modèle SDL des observateurs. Ces modèles étant petits (ils sont écrits par des ingénieurs), nous pouvons raisonnablement espérer que cette analyse ne souffrira pas de l'explosion combinatoire rencontrée au niveau du graphe généré. C'est là l'intérêt (mais aussi la limite) de l'approche explorée dans ce mémoire : proposer une définition d'une relation d'indépendance au niveau des modèles, puis guider la vérification par cette relation d'indépendance.

5.3 Formalisation de la relation d'indépendance

On dira que deux événements a_1 et a_2 produits par le contexte sont indépendants si et seulement si :

- ils sont indépendants vis-à-vis du modèle SDL du système, c'est-à-dire vérifient la propriété du "losange" (les sorties produites et les états atteints après $a_1 \cdot a_2$ et après $a_2 \cdot a_1$ sont identiques),
- ces événements et leurs conséquences (i.e., les sorties produites par le système en réaction) sont indépendants vis-à-vis de l'observateur à vérifier,
- et ces événements et leurs conséquences sont indépendants vis-à-vis du contexte lui-même.

La première indépendance sera appelée $S\_Indep_{\mathcal{S}}$ et ne portera que sur la structure d'automate du modèle du système. La seconde indépendance appelée $O\_Indep_{\mathcal{S},\mathcal{O}}$ portera sur les runs du système et sur l'observateur. Enfin la troisième indépendance appelée $C\_Indep_{\mathcal{S},C}$ portera sur les runs du système et sur le modèle du contexte.

Dans la suite, nous considérons un système $\mathcal{S} = \langle \Sigma, init, T, Sig_{in}, Sig_{out}, Sv \rangle$, un contexte C et un observateur $\mathcal{O} = \langle \Sigma_o, init_o, T_o, Voc(C) \cup Sig_{in} \cup Sig_{out}, \{reject\}, Sv_o \rangle$.

(Note : Les signaux d'entrés de l'observateur sont les signaux produits ou reçus par le système et son contexte.)

5.3.1 Notations préliminaires.

La relation d'indépendance entre deux événements a_1 et a_2 de Sig_{in} reposant sur la propriété du "losange" sur le modèle SDL et sur une notion de causalité entre événements du contexte, trois notations préliminaires sont introduites :

- $tgt\_states_{\mathcal{S}} : Sig_{in}^* \times \Sigma \rightsquigarrow 2^{(\Sigma \times Sig_{in}^*)}$. $tgt\_states_{\mathcal{S}}(\sigma, s)$ est une fonction qui retourne l'ensemble des couples (état,buffer) de l'automate SDL $\mathcal{S}$ atteints à partir d'un état s et après avoir reçu la séquence d'entrée σ. L'exécution de σ dans l'état s revient à dépiler itérativement une pile d'événements. Cette fonction est définie par :

$$tgt\_states_{\mathcal{S}}(\sigma, s) \stackrel{\text{def}}{=} \{(s', \sigma') \mid (s, \sigma) \xrightarrow[r1_{\mathcal{S}}]{} \mathcal{S} \quad (s_1, \sigma_1) \xrightarrow[r2_{\mathcal{S}}]{} \mathcal{S} \quad \dots \xrightarrow[r_{n_{\mathcal{S}}}]{} \mathcal{S} \quad (s', \sigma') \not\rightarrow_{\mathcal{S}} \quad \}$$

  Exemple. Sur le système de la figure 5.2 (page 66), $tgt\_states_{\mathcal{S}}(e_1 \cdot e_2, idle) = \{(busy_{12}, null_{\sigma})\}$.

- $out_{\mathcal{S}} : Sig_{in}^* \times \Sigma \rightsquigarrow 2^{(Sig_{in} \cup Sig_{out})^*}$. $out_{\mathcal{S}}(\sigma, s)$ est une fonction qui retourne l'ensemble des séquences d'événements de sortie, entrelacés avec le flot des événements d'entrée σ, produits par l'automate SDL $\mathcal{S}$ à partir d'un état s en réponse à σ. $out_{\mathcal{S}}(\sigma, s)$ est définie par :

$$out_{\mathcal{S}}(a_1 \cdot \ldots \cdot a_n, s) \stackrel{\text{def}}{=} [\![(a_1!; \cdot \ldots \cdot a_n!; \mathbf{0}) | (s, \mathcal{S})]\!]$$

  .
  C'est-à-dire : $out_{\mathcal{S}}(a_1 \cdot \ldots \cdot a_n, s)$ est la réponse de $\mathcal{S}$ dans l'état s confronté au contexte formé par un unique MSC envoyant en séquence les événements $a_1, \ldots, a_n$.

  Exemple. Sur le système représenté sur la figure 5.2 (page 66),

$$out_{\mathcal{S}}(e_1 \cdot e_2, idle) = \{(e_1 \cdot down \cdot light\_on \cdot e_2), (e_1 \cdot e_2 \cdot down \cdot light\_on)\}.$$

  La première trace correspond au scénario où le système à le temps de réagir entre l'arrivée de e_1 et e_2. La seconde trace résulte de l'arrivée de e_1 et de e_2 (dans cet ordre) avant que le système ne commence sa réaction.

- $<_C \subseteq Voc(C) \times Voc(C)$ une relation de causalité entre événements d'un contexte C. Soient a et b deux événements du vocabulaire de C ; b dépend causalement de a dans C, noté $a <_C b$ si b apparaît en séquence après a. $<_C$ est la plus petite relation définie inductivement par :

  ☐ si $b \in Voc(M)$ alors $a <_{a!;M} b$ et $a <_{a?;M} b$
  ☐ si $a \in Voc(C_1)$ et si $b \in Voc(C_2)$ alors $a <_{C_1.C_2} b$
  ☐ si $a <_{C_1} b$ alors $a <_{C_1\ op\ C_2} b$ et $a <_{C_2\ op\ C_1} b$ pour $op \in \{., +, \|\}$

  Ainsi, dans le contexte C figure 5.4a (page 67), $e_1 <_C s_1$, en revanche $e_1 \not<_C e_2$ puisque e_1 et e_2 apparaissent dans deux MSC différents en parallèle l'un de l'autre.

5.3.2 Indépendance vis-à-vis du système : $S\_Indep_{\mathcal{S}}$.

Deux événements d'entrée a_1 et a_2 de $\mathcal{S}$ sont indépendants vis-à-vis du système si et seulement si ils commutent, c'est-à-dire si à partir de n'importe quel état du système les deux séquences $a_1 \cdot a_2$ et $a_2 \cdot a_1$ conduisent aux mêmes états SDL avec les mêmes buffers. Formellement, on définit la relation d'indépendance $S\_Indep_{\mathcal{S}} \subseteq Sig_{in} \times Sig_{in}$ par :

$$(a_1, a_2) \in S\_Indep_{\mathcal{S}} \Leftrightarrow \forall s \in \Sigma, tgt\_states_{\mathcal{S}}(a_1 \cdot a_2, s) = tgt\_states_{\mathcal{S}}(a_2 \cdot a_1, s)$$

Exemple. Par exemple, les événements e_1 et e_2 du système figure 5.2 (page 66) sont $S\_Indep_{\mathcal{S}}$. En effet, quelque soit l'état du système, les scénarios $\sigma_1 = e_1 \cdot e_2$ et $\sigma_2 = e_2 \cdot e_1$ conduisent dans les mêmes états. En revanche, *close* et e_1 ne sont pas $S\_Indep_{\mathcal{S}}$ puisque dans l'état *idle*, $e_1 \cdot close$ mène dans $busy_1$ avec *close* mémorisé. Alors que $close \cdot e_1$ mène dans *safe* avec e_1 mémorisé.

Le tableau ci-dessous donne les relations d'indépendance vis-à-vis du système de la figure 5.2 (page 66), dans tous les états.

state	$tgt\_states_{\mathcal{S}}(e_1 \cdot e_2, state)$	$tgt\_states_{\mathcal{S}}(e_2 \cdot e_1, state)$	$S\_Indep_{\mathcal{S}}(e_1, e_2, state)$
idle	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_{12}$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
tempo	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_1$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
$busy_2$	$\{busy_{12}\}$	$\{busy_{12}\}$	*true*
safe	$\{safe\}$	$\{safe\}$	*true*

Dans tous les états s, $tgt\_states_{\mathcal{S}}(e_1 \cdot e_2, s) = tgt\_states_{\mathcal{S}}(e_2 \cdot e_1, s)$. e_1 et e_2 sont indépendants. Finalement, l'ensemble des couples d'événements indépendants sont les suivants :

$$S\_Indep_{\mathcal{S}} = \{(e1, e2), (e1, s2), (e2, s1), (s1, s2), (s1, open), (s2, open)\}.$$

5.3.3 Indépendance vis-à-vis de l'observateur : $O\_Indep_{\mathcal{S},\mathcal{O}}$.

Soit deux événements d'entrée a_1 et a_2 de $\mathcal{S}$. Considérons un état s de ce système ; considérons enfin les traces obtenues par stimulation de $\mathcal{S}$ dans s par $a_1 \cdot a_2$ puis par $a_2 \cdot a_1$. Par définition, ces traces sont :

$$out_{\mathcal{S}}(a_1 \cdot a_2, s) \quad \text{et} \quad out_{\mathcal{S}}(a_2 \cdot a_1, s)$$

a_1 et a_2 sont indépendants vis-à-vis de l'observateur si les traces engendrées par $a_1 \cdot a_2$ et $a_2 \cdot a_1$ (contenant les événements émis par le système en réponse) produisent le même comportement interne de l'observateur, c'est-à-dire atteignent le même état :

$$(a_1, a_2) \in O\_Indep_{\mathcal{S},\mathcal{O}} \Leftrightarrow \forall s \in \Sigma, \begin{cases} \forall t_1 \in out_{\mathcal{S}}(a_1 \cdot a_2, s), \exists t_2 \in out_{\mathcal{S}}(a_2 \cdot a_1, s) \\ \quad \text{tq } \forall s_0 \in \Sigma_o, tgt\_states_{\mathcal{O}}(r_1, s_0) = tgt\_states_{\mathcal{O}}(r_2, s_0) \\ \forall t_2 \in out_{\mathcal{S}}(a_2 \cdot a_1, s), \exists t_1 \in out_{\mathcal{S}}(a_1 \cdot a_2, s) \\ \quad \text{tq } \forall s_0 \in \Sigma_o, tgt\_states_{\mathcal{O}}(r_2, s_0) = tgt\_states_{\mathcal{O}}(r_1, s_0) \end{cases}$$

Il est trivial alors de constater que, si a_1 et a_2 sont indépendants vis-à-vis de l'observateur, soit les deux scénarios $a_1 \cdot a_2$ et $a_2 \cdot a_1$ satisfont l'observateur, soit aucun des deux ne le satisfait.

Proposition 1

$$\forall (a_1, a_2) \in Sig_i^2, \forall s \in \Sigma, (a_1, a_2) \in O\_Indep_{\mathcal{S},\mathcal{O}} \Rightarrow \begin{cases} (a_1; a_2.\mathbf{0})|(s, \mathcal{S}) \vDash \mathcal{O} \\ \Leftrightarrow (a_2; a_1; \mathbf{0})|(s, \mathcal{S}) \vDash \mathcal{O} \end{cases}$$

Cette proposition se généralise par :

Proposition 2 $\forall (a_1, a_2) \in Sig_i^2, \forall s \in \Sigma, \forall \sigma, \sigma' \in Sig_i^*$,

$$(a_1, a_2) \in O\_Indep_{\mathcal{S},\mathcal{O}} \Rightarrow \begin{cases} M_\sigma.(a_1; a_2.\mathbf{0}).M_{\sigma'}|s, \mathcal{S} \vDash \mathcal{O} \\ \Leftrightarrow M_\sigma.(a_2; a_1.\mathbf{0}).M_{\sigma'}|s, \mathcal{S} \vDash \mathcal{O} \\ \textit{où } M_\sigma \textit{ et } M_{\sigma'} \textit{ sont les MSCs exécutant } \sigma \textit{ et } \sigma'. \end{cases}$$

Exemple. Les événements e_1 et e_2 du système figure 5.2 (page 66) sont $O\_Indep_{\mathcal{S},\mathcal{O}_4}$. En effet, les scénarios engendrés par $e_1 \cdot e_2$ et $e_2 \cdot e_1$, restreints aux événements observés par $\mathcal{O}_4$, sont égaux à l'unique événement *down*. Il vient trivialement que ces observateurs produisent le même comportement sur $\mathcal{O}_4$ (et donc le même état).

5.3.4 Indépendance vis-à-vis du contexte : $C\_Indep_{\mathcal{S},C}$.

Soit deux événements d'entrée a_1 et a_2 de $\mathcal{S}$. Considérons un état s de $\mathcal{S}$. Considérons à nouveau les traces obtenues par stimulation de $\mathcal{S}$ dans s par $a_1 \cdot a_2$ puis par $a_2 \cdot a_1$:

$$out_{\mathcal{S}}(a_1 \cdot a_2, s) \quad \text{et} \quad out_{\mathcal{S}}(a_2 \cdot a_1, s)$$

Ici, nous ne pouvons plus raisonner comme précédemment sur l'état du contexte atteint par ces deux ensembles de scénarios à partir d'un état quelconque de ce contexte ; le nombre d'états du contexte étant potentiellement très grand, la construction de cette relation d'indépendance se heurterait à nouveau à une explosion combinatoire. Une relation plus forte est donc construite, ne nécessitant qu'un parcours syntaxique du contexte. a_1 et a_2 sont indépendants vis-à-vis du contexte C si les deux ensembles de traces $out_{\mathcal{S}}(a_1 \cdot a_2, s)$ et $out_{\mathcal{S}}(a_2 \cdot a_1, s)$ restreintes au vocabulaire de C (on élimine les événements non considérés par C) sont tels que pour toute trace r_1 du premier ensemble (engendrée par le premier scénario), il existe une trace r_2 du second ensemble

- qui ne diffère de r_1 que par l'ordre des événements, r_1 et r_2 contiennent le même nombre d'occurrences de chaque événement,
- et telle que si (b, b') est un couple d'événements différent de (a_1, a_2) ou (a_2, a_1) et ordonné différemment dans r_1 et dans r_2 (par exemple b est avant b' dans r_1, et inversement dans r_2), alors b et b' apparaissent dans des parties parallèles du contexte (le contexte ne spécifie pas l'ordre de b et b').

L'idée en particulier est de distinguer parmi les scénarios ceux qui induisent des interactions dans un ordre conforme à l'ordre spécifié par le contexte, et ceux qui au contraire ne respectent pas cet ordre. Considérons par exemple un contexte $C' = light\_on?; down?; \mathbf{0}$ qui spécifie que *light_on* doit être reçu (par le contexte) avant *down*. Considérons les deux événements e_1 et e_2 du système figure 5.2 (page 66). Le scénario d'entrée $e_1 \cdot e_2$ produit la séquence de sortie $down \cdot light\_on$, alors que le scénario inverse produit la séquence de sortie inverse. Or, C' attendant ces deux derniers événements dans un ordre particulier : il vient que, par rapport à C', e_1 et e_2 ne sont pas indépendants.

- Dans le cas $e_1 \cdot e_2$, le contexte C' reçoit *down* alors qu'il ne l'attend pas, et donc le perd. Puis il reçoit *light_On* et attend ensuite indéfiniment *down*.
- Dans le cas $e_2 \cdot e_1$, C' reçoit *light_On*, le consomme, puis reçoit *down* et le consomme également. Le contexte C' est alors terminé.

Formellement, la relation $C\_Indep_{\mathcal{S},C}$ est définie par :

$$(a_1, a_2) \in C\_Indep_{\mathcal{S},C} \Leftrightarrow \forall s \in \Sigma, \begin{cases} \forall t_1 \in out_{\mathcal{S}}(a_1 \cdot a_2, s), \exists t_2 \in out_{\mathcal{S}}(a_2 \cdot a_1, s) \\ \quad \text{tq } t_{1\setminus Voc(C)} \sim_C t_{2\setminus Voc(C)} \\ \forall t_2 \in out_{\mathcal{S}}(a_2 \cdot a_1, s), \exists t_1 \in out_{\mathcal{S}}(a_1 \cdot a_2, s) \\ \quad \text{tq } t_{2\setminus Voc(C)} \sim_C t_{1\setminus Voc(C)} \end{cases}$$

où $t_{\setminus Voc(C)}$ est la trace t restreinte au vocabulaire de C (les événements qui ne sont pas dans C sont supprimés), et où $\sim_C$ est une relation d'équivalence entre traces définie par

$$t \sim_C t' \Leftrightarrow \begin{cases} t \text{ et } t' \text{ sont de même longueur} \\ \forall a, \text{ les nombres d'occurrences de } a \text{ dans } t \text{ et } t' \text{ sont égaux} \\ \forall a, b \text{ tq } a \text{ et } b \text{ sont inversés dans } t \text{ et } t', \text{ alors } a \not<_C b \text{ et } b \not<_C a \end{cases}$$

5.3.5 Relation d'indépendance globale.

Soit a_1 et a_2 deux événements d'entrée de $\mathcal{S}$; a_1 et a_2 sont indépendants, noté $(a_1, a_2) \in Indep_{\mathcal{S},\mathcal{O},C}$ s'ils sont indépendants vis-à-vis du système, de l'observateur, et du contexte :

$$Indep_{\mathcal{S},\mathcal{O},C} \stackrel{\text{def}}{=} S\_Indep_{\mathcal{S}} \cap O\_Indep_{\mathcal{S},\mathcal{O}} \cap C\_Indep_{\mathcal{S},C}$$

5.4 Equivalence de scénarios

5.4.1 Principes.

Application à la réduction des scénarios. Soit $\mathcal{S}$ un système SDL, C son contexte et $\mathcal{O}$ un observateur à vérifier. En considérant le monoïde libre partiellement commutatif dont les mots sont les scénarios d'événements produits par C et dont la relation d'indépendance est $Indep_{\mathcal{S},\mathcal{O},C}$, et en appliquant la méthode de [DM97] il est maintenant facile de calculer l'ensemble des scénarios distincts (i.e., qui ont un effet différent sur le système ou son observateur). Ces scénarios sont les classes d'équivalence $[w]$ du monoïde partiellement commutatif $\mathbb{M}(Sig_{in}, Indep_{\mathcal{S},\mathcal{O},C})$ où Sig_{in} est l'ensemble des événements d'entrée du système.

Le résultat important de cette réduction, énoncé ci-après, est que tous les scénarios d'une même classe d'équivalence satisfont de la même manière (i.e., satisfont ou ne satisfont pas) l'observateur $\mathcal{O}$.

Calcul des classes d'équivalences. Les classes d'équivalences correspondent aux formes normales de Foata présentées dans la section 3.3.2 (page 35). Chaque forme normale de Foata sera représentatif d'une classe. C'est cette méthode qui est utilisée par la suite pour appliquer les méthodes de réduction par ordres-partiels sur le contexte CDL.

Proposition 3 *Pour toute classe $[w]$ de scénarios équivalents au sens du monoïde libre partiellement commutatif $\mathbb{M}(Sig_{in}, Indep_{\mathcal{S},\mathcal{O},C})$, alors :*
$\forall \sigma_1, \sigma_2 \in [w], \quad C_{\sigma_1} \| (init, \mathcal{S}) \models \mathcal{O} \quad \Leftrightarrow \quad C_{\sigma_2} \| (init, \mathcal{S}) \models \mathcal{O}$ *où C_{σ_1} et C_{σ_2} sont des contextes séquentiels jouant respectivement les scénarios σ_1 et σ_2.*

Preuve. *Ce résultat découle directement de la définition de la relation d'indépendance entre événements. Par définition des traces de Mazurkiewicz, il est possible de passer de σ_1 à σ_2 par une succession de permutations d'événements indépendants. Or par définition de la relation d'indépendance, une permutation de deux événements indépendants ne change pas le comportement du système, ni de l'observateur. En conséquence, une telle permutation ne changera pas la valeur de vérité de l'observateur. Et donc au final, σ_1 et σ_2 satisferont (ou ne satisferont pas) l'observateur.*□

Ce résultat est fondamental. C'est lui qui permet de réduire, parfois de façon considérable (voir les résultats dans la section suivante) le nombre de scénarios différents issus du contexte, et donc au final l'espace de la vérification.

5.4.2 Algorithmes

Cette section définit les procédures de décision implantant les définitions formelles des relations d'indépendances au niveau système, au niveau du contexte et au niveau des observateurs. Ces algorithmes permettent ainsi de générer l'indépendance globale entre événements du contexte.

Indépendance au niveau système. L'algorithme 2 (page 73) énonce que deux événements a et b sont indépendants vis-à-vis du système si et seulement si l'ensemble des états atteints à partir de l'état s en exécutant le scénario $a \cdot b$ et le scénario $b \cdot a$ sont identiques.

La complexité de l'algorithme 2 est de $O(\mid \Sigma \mid \times d_{indet}^4)$, où d_{indet} est le degré d'indéterminisme de $\mathcal{S}$; c'est à dire le nombre de transitions maximal franchissables à partir d'un même état et par le même événement. La complexité est donc linéaire par rapport au nombre d'états du système, et polynomiale sur le nombre de branchements non déterministes.

Indépendance au niveau des observateurs. L'algorithme 3 (page 73) décrit l'indépendance de deux événements a et b vis-à-vis d'un observateur. Les traces d'exécutions r_1 obtenues en faisant $a \cdot b$ (respectivement r_2 en faisant $b \cdot a$) sont tout d'abord récupérées. Pour chaque trace de r_1, nous regardons s'il existe une trace dans r_2 telle que ces deux traces conduisent aux mêmes états dans l'observateur, et inversement.

```
S_Indep_S(a, b, S) :
foreach state s ∈ S.Σ do
    if (¬(tgt_states_S(a · b, s) = tgt_states_S(b · a, s))) then
        return false;
    end
end
return true;
```

Algorithme 2 : Algorithme de la relation d'indépendance au niveau système

```
O_Indep_{S,O}(a, b, S, O) :
r1 ← ∅; r2 ← ∅;
foreach state s ∈ S.Σ do
    r1 ← out_S(a.b, s); r2 ← out_S(b.a, s);
    if (¬(indep_o(r1, r2, O) && indep_o(r2, r1, O))) then
        return false;
    end
end
return true;

indep_o(r1, r2, O) :
okForT1 ← false;
foreach trace t1 ∈ r1 do
    foreach state s_0 ∈ O do
        okForT1 ← false;
        foreach trace t2 ∈ r2 do
            if (tgt_o(t1, s_0) == tgt_o(t2, s_0)) then
                okForT1 ← true;
            end
            if (¬(okForT1)) then
                return false;
            end
        end
    end
end
return true;
```

Algorithme 3 : Algorithme de l'indépendance par rapport à l'observateur

La complexité de l'algorithme 3 est de $O(\mid \Sigma \mid \times d^4_{indet} \times \mid \Sigma_o \mid)$. La complexité est linéaire par rapport au nombre des états de l'observateur et linéaire par rapport au nombre des états du système. En revanche, elle est polynomiale sur le nombre de branchements non déterministes du système.

Indépendance au niveau du contexte. L'algorithme 4 (page 75) décrit l'indépendance de deux événements a et b vis-à-vis d'un contexte. Le contexte est parcouru de manière inductive. Si les événements a et b appartiennent à une même séquence, alors ils sont dépendants dans le contexte. Si les événements appartiennent à une alternative ou à deux séquences concurrentes (parallélisme) alors ils sont indépendants dans le contexte. La complexité de l'algorithme est de $\mathcal{O}(L^2 \mid \mathcal{C} \mid)$ avec L la longueur maximale d'une transition (le nombre d'événements figurant dans la transition).

Indépendance globale. La complexité du calcul des indépendances est finalement de :

$$\mathcal{O}(\mid \Sigma \mid d^4_{indet}(\mid \Sigma_o \mid + L^2 \mid \mathcal{C} \mid))$$

La complexité est donc linéaire sur le nombre d'états du système SDL et polynomiale sur le nombre de branchements non déterministes.

5.4.3 Implantation des algorithmes de réduction

Cette partie présente la méthode et le langage utilisé pour l'implantation des algorithmes de réduction. Le langage de programmation utilisé est un langage fonctionnel, très bien adapté pour la représentation et la définition d'ensembles et fonctions mathématiques. Les algorithmes précédents ont été implantés dans le langage OCAML.

Modules OCAML. Le programme réalisé se décompose en 3 modules complémentaires représentant approximativement 5000 lignes de code OCAML.

- Le premier module est un analyseur syntaxique du langage CDL permettant de générer l'arbre syntaxique du contexte et ainsi effectuer des analyses statiques comme introduit dans la partie sur l'indépendance due au contexte (section 5.3.4 page 71).
- Un deuxième module implante les opérateurs pour le calcul de la relation d'indépendance mono-processus.
- Un dernier module contenant le programme principal implante les opérateurs pour le cas multi-processus ainsi que les algorithmes de calcul des formes normales de Foata (cf section 3.3.2 page 35).

Ce dernier algorithme a été réécrit en JAVA pour des questions de performances.

5.5 Reconstruction du graphe de scénario global

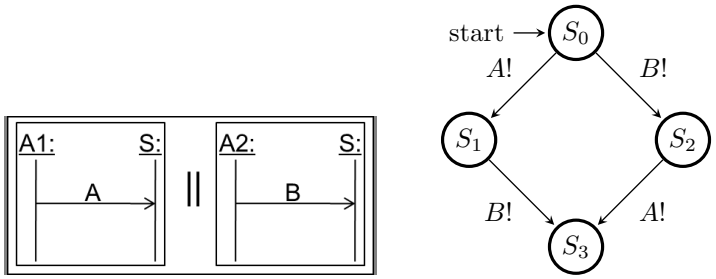
Les procédures de décisions précédentes permettent de calculer des équivalences entre les scénarios du contexte. Or, ces scénarios ne contiennent actuellement que les messages d'envoi du contexte ; c'est à dire des messages envoyés du contexte vers le système. Ces messages d'envoi étant les seuls stimulis permettant de modifier le comportement d'un système, il devenait alors inutile de conserver les messages de réception dans les scénarios. Cependant, pour la vérification effective d'un scénario avec le système et l'observateur, il devient nécessaire de reconstruire le scénario d'exécution complet. Cette section définit une méthode pour reconstruire ce scénario complet.

```
C_Indep_{c,(}(a, b, S, C) :
r1 ← ∅; r2 ← ∅;
foreach state s ∈ S.Σ do
    r1 ← out_S(a.b, s); r2 ← out_S(b.a, s);
    if (¬(indep_c(r1, r2, C) && indep_c(r2, r1, C))) then
        return false;
    end
end
return true;

indep_c(r1, r2, C) :
okForT1 ← false;
foreach trace t1 ∈ r1 do
    okForT1 ← false;
    foreach trace t2 ∈ r2 do
        if (equiv(t1, t2)) then
            okForT1 ← true;
        end
    end
    if (¬(okForT1)) then
        return false;
    end
end
return true;

equiv(t1, t2) :
if (¬(|t1| == |t2|)) /* |t| = taille de t */ then
    return false;
else
    foreach letter a ∈ t1 do
        if (¬(|t1|_a == |t2|_a)) /* |t|_a = nombre d'occurences de a dans t */ then
            return false;
        else
            foreach (a, b) tel que a precedes b dans t1 et b precedes a dans t2 do
                if (¬(<_C (a, b, C))) then
                    return false;
                end
            end
        end
    end
end
return true;

<_C (a, b, C) :
if (C = C_1 ∥ C_2) then
    return <_C (a, b, C_1) ∥ <_C (a, b, S, C_2);
end
if (C = C_1; C_2) then
    return ((<_C (a, b, C_1) ∥ <_C (a, b, C_2)) ∥ (a ∈ C_1, b ∈ C_2));
end
if (C = C_1 + C_2) then
    return <_C (a, b, C_1) ∥ <_C (a, b, C_2);
end
if (C = c · M && c ≠ a) then
    return <_C (a, b, M);
end
if (C = 0) then
    return false;
end
```

Algorithme 4 : Algorithme de l'indépendance par rapport à un contexte

Dualité : messages d'envoi/messages de réception. Dans l'introduction nous avions résumé les différentes étapes pour la réduction du contexte. Tout d'abord, le contexte et les propriétés, devant être vérifiées, sont modélisés par l'utilisateur via le langage CDL. La première étape consiste alors à calculer la relation d'indépendance entre événements du contexte constitué d'acteurs en parallèle. Cela est effectué par une analyse statique du modèle SDL. Une fois l'ensemble des indépendances calculé, la procédure de décision basée sur les formes normales de Foata est appliquée de telle manière à obtenir les classes d'équivalence de scénarios.

Dissociation des messages. Lorsque le contexte n'est constitué que de messages d'envoi du contexte vers le système, la méthode de réduction qui consiste à générer l'ensemble des scénarios possibles et par la suite réduire ceux qui sont équivalents par les formes normales de Foata peut être appliquée directement. Considérons le contexte de la figure 5.5.



(a) Contexte (b) Contexte déplié

FIG. 5.5: Contexte uniquement constitué de messages d'envoi

Le contexte représenté sur la figure 5.5a est composé de deux acteurs A_1 et A_2 en parallèles envoyant respectivement les messages A et B au système S. L'entrelacement de ces deux acteurs donne le contexte déplié de la figure 5.5b. Les traces de ce contexte déplié sont alors :

- $A! \cdot B!$,
- $B! \cdot A!$.

Supposons à présent que A et B sont indépendants ; alors ces deux traces appartiennent à la même trace de Mazurkiewicz. En appliquant la méthode des ordres-partiels, un seul scénario sera nécessaire alors, pour la vérification du système correspondant.

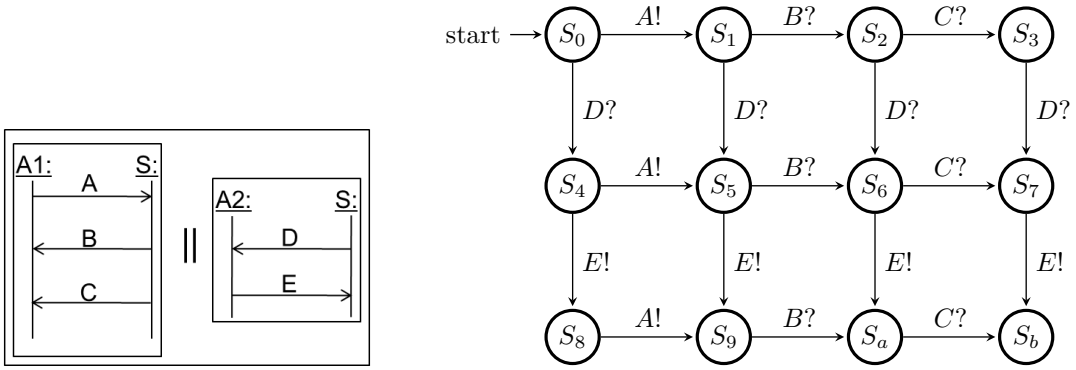
Dissocier pour régner. Il est important de noter que les réductions effectués dans ce mémoire se basent uniquement sur les messages d'envoi du contexte (que l'on appellera **les sorties** du contexte) vers le système sans prendre en compte les messages de réception (que l'on appellera **les entrées** du contexte) par le contexte (messages envoyés du système vers le contexte). La principale raison est que le contexte ne peut prévoir à l'avance l'ordre dans lequel le système enverra ses messages en réponse à une sollicitation de l'environnement. Or, il peut arriver que l'envoi d'un message du contexte vers le système nécessite au préalable qu'un message envoyé du système au contexte ait été reçu. L'idée ici est alors de ne prendre en compte dans un premier temps uniquement les messages contrôlables par l'environnement du système et qui peuvent être envoyés à ce dernier dans un ordre quelconque. Ceci est effectué en éliminant tous les messages envoyés par le système vers le contexte.

Néanmoins, lorsque des messages d'envois sont initialement inhibés par un message de réception provenant du système, alors une synchronisation dans l'exécution des scénarios d'envoi est nécessaire. Pour prendre en compte ces synchronisations, dans un deuxième temps (une fois que l'ensemble des scénarios d'envoi équivalents ont été déterminés), les messages de réception qui ont été explicitement décrits dans l'environnement initial sont réinjectés. Ainsi, un message d'envoi du contexte ne pourra plus être envoyé si le contexte spécifie qu'un message du système doit d'abord

être consommé. Cette contrainte n'était pas présente dans le cas de scénarios composé uniquement d'envois. C'est pourquoi nous décrivons dans la suite une méthode de reconstruction des scénarios.

Exemple. Pour illustrer ce point technique, considérons le contexte CDL de la figure 5.6a constitué par deux acteurs envoyant respectivement les messages **contrôlables** A et E à un même processus S (mono-processus). Contrairement à l'exemple précédent, E ne peut être envoyé que si l'acteur $A2$ a consommé le message D. Dans ce cas E est inhibé et dépend du message de réception D. Sur le graphe résultat de l'entrelacement de A_1 et A_2 (figure 5.6b), nous constatons que E ne peut être envoyé qu'après les possibles entrelacements entre D et l'acteur $A1$. Or, ce qui nous intéresse ici, c'est de déterminer si les scénarios du contexte sont équivalents ou non (ont le même impact sur le système). Puisque le contexte ne possède aucune information sur la dynamique d'exécution du système, le calcul de la relation d'indépendance se fait sur les messages d'envoi du contexte uniquement.

Continuons avec l'exemple. Son graphe totalement déplié est représenté sur la figure 5.5b avec l'ensemble des messages d'envoi et de réception. Bien entendu pour des contextes volumineux une explosion combinatoire peut être engendrée ; c'est pourquoi dans ce mémoire nous souhaitons éviter sa construction explicite. D'un point de vue du contexte, il ne peut pas déterminer dans quel ordre B C et D lui seront envoyés par le système qui est alors vu comme une **boîte noire**.



(a) Contexte (b) Contexte déplié

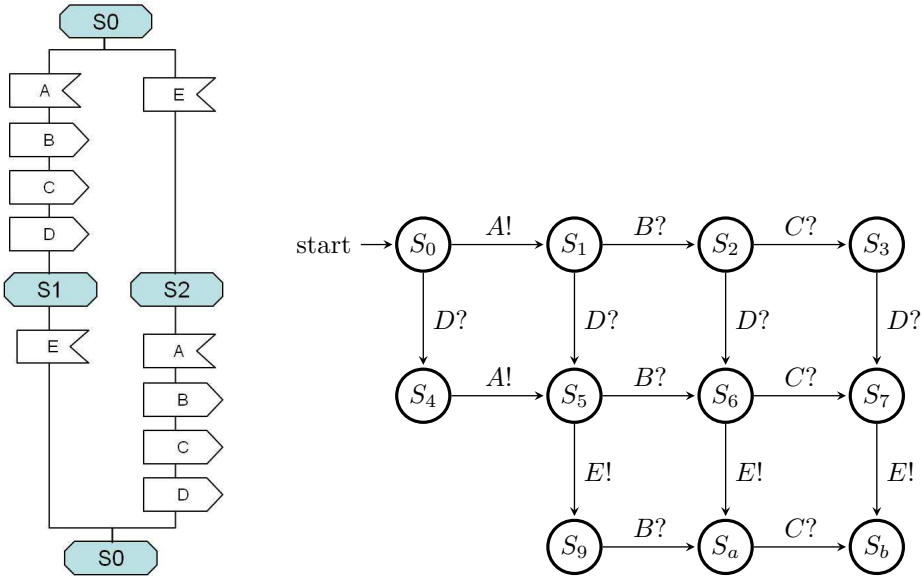
FIG. 5.6: Dissociation de messages

Maintenant supposons que le système S soit modélisé comme sur la figure 5.7a. Une analyse statique informe que A et E sont indépendants. Le graphe de la figure 5.6b, après réduction devient le graphe de la figure 5.7b.

Reconstruction des graphes de contexte. A cette étape, pour effectuer la composition et la simulation entre le scénario du contexte et le système, nous devons reconstituer le scénario complet qui sera en fait, un graphe avec les messages de réception, pertinents pour la vérification. Ces derniers sont présents dans le contexte initial. Le graphe de scénarios correspondant à la trace $A.E$ est alors construit. Puisque le contexte ne possède aucune information sur l'ordre de réception des messages de réception de l'acteur $A1$:B, C, avec les messages de réception de l'acteur $A2$:D, et puisque l'on ne souhaite pas explicitement exécuter le système, afin de déterminer la dynamique exacte de ce dernier, nous rajoutons les messages $B?$ et $C?$ et $D?$ avec $C?$ après $B?$, $B?$ après $A?$ et $D?$ avant $E!$, de manière à ce que le système déterminise lui-même le scénario jouable dans le graphe présenté. Ceci peut être fait en construisant le scénario complet avec un ensemble de processus parallèles, synchronisés par des variables et des gardes booléennes (figure 5.8).

Si nous regardons le graphe obtenu après réduction et reconstruction du graphe à partir de la trace $A \cdot E$, nous obtenons le graphe réduit de la figure 5.7b.

Avantage des graphes de contexte. Si nous déplions entièrement le graphe réduit de la figure 5.7b, alors les 9 traces complètes suivantes sont obtenues :



(a) Système SDL (b) Construction du graphe de scénario réduit

FIG. 5.7: Graphe réduit du contexte

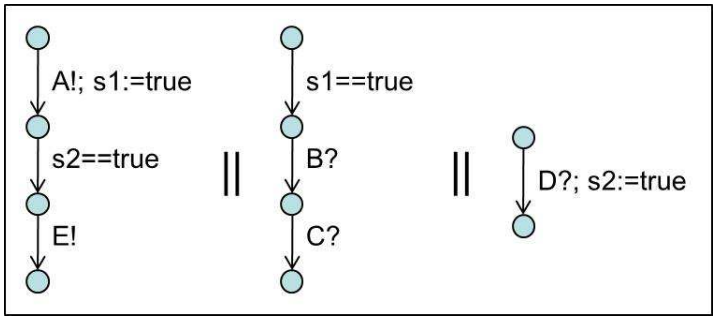


FIG. 5.8: Construction des scénarios complets

1. $A! \cdot D? \cdot E! \cdot B? \cdot C?$
2. $A! \cdot D? \cdot B? \cdot E! \cdot C?$
3. $A! \cdot D? \cdot B? \cdot C? \cdot E!$
4. $A! \cdot B? \cdot D? \cdot E! \cdot C?$
5. $A! \cdot B? \cdot D? \cdot C? \cdot E!$
6. $A! \cdot B? \cdot C? \cdot D? \cdot E!$
7. $D? \cdot A! \cdot B? \cdot C? \cdot E!$
8. $D? \cdot A! \cdot B? \cdot E! \cdot C?$
9. $D? \cdot A! \cdot E! \cdot B? \cdot C?$

Composer chacune de ces traces avec le système n'est pas utile car on constate que seul le scénario 6 est exécutable. En effet les scénarios 1,2,3 ne peuvent pas être exécutés jusqu'au bout car une fois $A!$ envoyé, le scénario se met en attente de $D?$. Alors, B puis C sont envoyés par $\mathcal{S}$. Ils sont donc perdus (car non attendus par les scénarios, qui ne pourront plus les consommer lorsqu'ils les attendront ultérieurement). Enfin, les scénarios 7, 8 et 9 sont bloqués sur l'attente de $D?$ dès que le système attend $A?$ ou $E?$.

Ainsi, 8 scénarios sur 9 ne peuvent être joués jusqu'au bout. Si nous devions tous les jouer, cela entraînerait par conséquent 9 compositions avec le système pour effectuer la vérification d'un seul scénario valide. Au contraire, en composant directement les scénarios complets (figure 5.8) avec le système, ce dernier va déterminiser le scénario en jouant le seul scénario "jouable".

5.6 Expérimentations

5.6.1 Protocole d'expérimentations.

Configurations de la machine utilisée. Les résultats donnés dans la suite ont été produits sur un ordinateur possédant un processeur Intel dual core cadencé à 2GHz avec une capacité de mémoire de 3.5Go DDR2[1].

Vérification avec SPIN avec décomposition des scénarios (méthode OBP). Comme le souligne le schéma de la figure 5.9a, la méthode de vérification proposée dans ce mémoire se décompose de la manière suivante :

- Le modèle SDL, le contexte ainsi que l'observateur (propriété) sont parcourus par le *lexer* et le *parser* du langage OCAML de telle manière à pouvoir effectuer une analyse statique de ces trois entités.
- Le calcul des relations d'indépendances est effectué en parcourant statiquement le système, le contexte et l'observateur.
- Ensuite, l'algorithme de réduction par les formes normales de Foata (section 3.3.2) génère les scénarios réduits à partir des indépendances d'événements du contexte.
- Finalement, chaque scénario réduit est composé avec le système et la propriété à vérifier avec le *model-checker* SPIN

Vérification avec SPIN sans décomposition de scénarios A titre de comparaison, nous avons effectué la vérification du système et des propriétés par la méthode classique du *model-checking*, consistant à composer l'automate global du contexte avec le système et la propriété. Pour tester les limites de SPIN nous avons utilisé les options de compilation suivantes :

- La réduction par ordres-partiels activée par défaut,
- l'option COLLAPSE pour la compression des états,
- l'option VECTORSZ pour augmenter la taille du vecteur d'un état

Il existe, de plus, deux options de compilation supplémentaires : l'option BITSTATE et l'option MA. L'option BITSTATE utilise l'algorithme du *bitstate hashing* permettant de déterminer si un état à déjà été parcouru ou non en l'indexant par une valeur unique et codant l'état en un minimum de bits. Cependant, pour éviter les collisions dans la hashtable, une liste chaînée contenant les états ayant la même valeur d'indexation est ajoutée. Néanmoins, cette méthode ne permet pas d'éviter toutes les collisions et des états peuvent ne pas être stockés dans la hashtable. Cette approximation du nombre d'états peut ainsi générer des "faux positifs (*false positive*)"[2] ; c'est à dire considérer qu'une propriété est vraie alors qu'elle est falsifiée. Nous n'avons donc pas utilisé cette option.

L'option MA est une variante de l'option BISTATE, mais qui cette fois-ci n'effectue plus d'approximations. Cela implique un temps de calcul relativement long pour permettre une minimisation d'occupation de la mémoire. Cette option n'a donc pas été utilisée. Le principe de vérification du *model-checker* XSPIN[3] est résumé sur la figure 5.9b :

- L'outil prend en entrée un modèle formalisé en langage Promela,
- Une propriété définie sous format LTL peut être transformée automatiquement en automate de Büchi par le traducteur ltl2ba. Ainsi la propriété LTL est transformée en code Promela

[1] Double Data Rate
[2] Ceci a été mis en évidence dans [OMHG03]
[3] version graphique de SPIN codée en Tcl/Tk.

(appelé *never claim*).

- Le parser de SPIN permet la détection des erreurs de syntaxe,
- Un mode interactif permet d'exécuter le système pas à pas en choisissant en ligne de commande le numéro des transitions exécutables dans l'état courant,
- Pour effectuer la vérification, SPIN génère un exécutable C.
- La vérification est effectuée à la volée et un contre-exemple est généré dans un fichier nommé *trail* avec la trace complète, à la première erreur découverte.

Note : Pour simplifier la lecture, dans la suite du document, nous notons par :

- **méthode classique**, la méthode de vérification utilisant le *model-checker* SPIN avec les diverses options disponibles sans génération ni réduction de scénarios du contexte.
- **méthode OBP**, la méthode de vérification utilisant le *model-checker* SPIN avec les diverses options disponibles et avec génération et réduction de scénarios du contexte.

Transformation des modèles SDL et CDL en Promela.

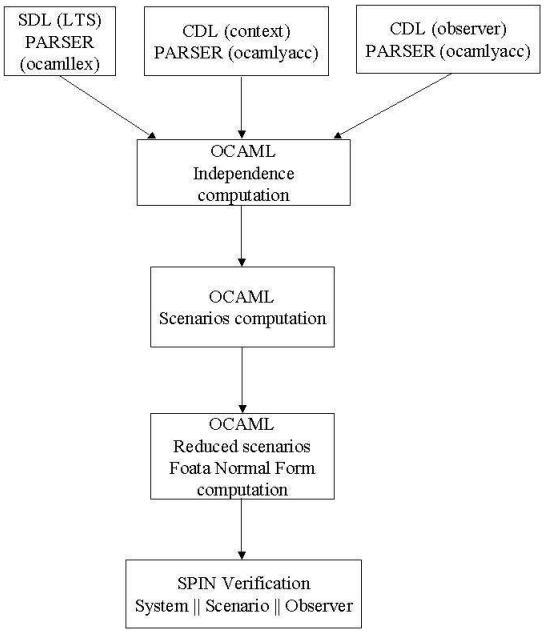
La transformation du langage SDL en Promela (qui est le langage d'entrée de SPIN) est directe car chaque clause SDL ayant une correspondance avec des clauses Promela excepté la clause *SAVE* SDL que nous utilisons pour la modélisation des systèmes.

Les sauvegardes SDL en Promela. La principale limitation de Promela est le fait qu'il ne supporte pas la sauvegarde des messages. Pour cela nous avons implanté la méthode proposée par [PCDR02] qui consiste à modéliser un processus à l'aide de deux buffers. Un premier buffer contenant les messages effectifs échangés et consommés, et un deuxième buffer complémentaire et temporaire au premier permettant de replacer les messages dans le bon ordre lorsque des sauvegardes sont présentes. Ainsi, si il existe des messages sauvegardés dans un état, ces derniers devront se trouver en tête du buffer dans l'état suivant. Le buffer temporaire permet donc d'effectuer ces modifications.

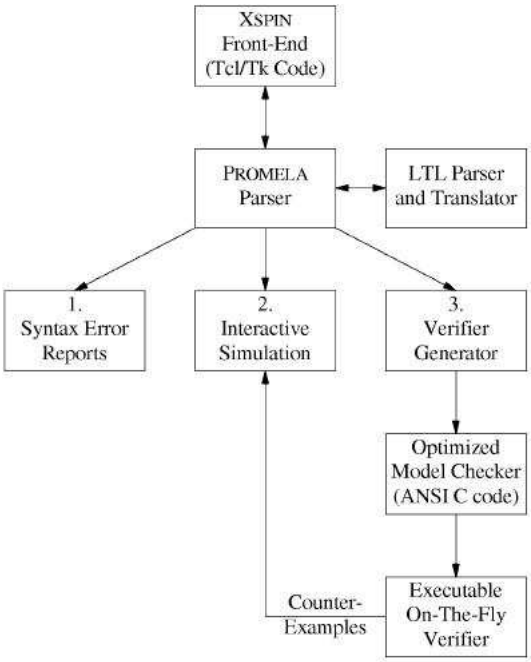
Les messages *discardés*. Dans la sémantique du langage SDL, un message qui n'est pas consommable est "discardé" c'est à dire implicitement consommé et donc perdu. En Promela cette notion n'existe pas. En effet un message est perdu dès lors que le buffer est plein. La gestion des messages non consommables est définie par une transition vide gardée par ce message. Lorsqu'un message est "discardé", l'état du système SDL ne change pas. C'est pourquoi une boucle doit être implantée dans chaque état pour pouvoir éliminer tous les messages non consommables. Dès lors qu'un message devient consommable, on quitte la boucle en changeant d'état.

Atomicité des transitions. Dans la sémantique de SDL, une transition est tirée sans possibilité d'être préemptée par d'autres processus. C'est ce que l'on appelle une transition atomique. Un opérateur a été défini dans le langage Promela pour prendre en compte cette spécificité.

Atomicité dans la vérification de propriétés Il faut toutefois faire attention à une notion très importante : SPIN est basé sur un système d'états. Par conséquent les propriétés qui sont vérifiées sont des propriétés d'états. Le fait de rendre atomique une transition empêche parfois de vérifier une propriété qui s'avère vrai en théorie. Par exemple, supposons que, comme dans le cas d'étude 1, nous souhaitons détecter l'envoi (et non la réception) d'un message par un processus. Supposons que la propriété que nous souhaitons vérifier est que le message a bien été envoyé. Pour détecter l'envoi nous pourrions envoyer le message et le recevoir immédiatement par une transition



(a) Vérification avec SPIN avec la méthode OBP



(b) Vérification avec SPIN sans la méthode OBP

FIG. 5.9: Stratégie de vérification

atomique. Or, le fait de rendre atomique cette séquence d'envoi et de réception de message empêche l'automate de Büchi de détecter la présence de ce dernier. La raison est la suivante : lorsque l'on rend une transition atomique, aucun état n'est créé entre le moment de l'envoi et le moment de la réception du message. Par conséquent lorsque le message est envoyé dans le buffer, il est automatiquement consommé. A partir de là, l'état atteint ne contient plus le message *up* qui a pourtant été envoyé mais qui n'est plus présent dans le *buffer* et par conséquent pas détectable par une *never claim*.

Le problème de la compilation. La technique du *model-checking* consiste à décrire le modèle formellement et ensuite le composer avec la propriété à vérifier pour obtenir un diagnostic pour des modèles pouvant être contenus en mémoire. Cela implique qu'une seule compilation du modèle et de la propriété est suffisante pour lancer la vérification complète.

Dans le cadre de ce mémoire, nous obtenons N scénarios réduits que nous devons composer avec le modèle ainsi que la propriété à vérifier pour effectuer la vérification globale. Cela nécessite à priori N compilations et ainsi N *model-checking*. Néanmoins, dès lors que le nombre de scénarios N devient très important, le sur-coût engendré par le temps de compilation deveient rédhibitoire. Dans la suite, nous effectuons les expérimentations en identifiant à chaque fois, le temps de compilation nécessaire. Toutefois, pour ne pas s'arrêter du fait des mauvaises performances induites par les temps de compilations, nous conjectuerons que la compilation séparée est possible et peut diminuer de façon drastique le temps de vérification total. En effet, les travaux d'optimisation sur les *model-checker* sont essentiellement focalisés sur la réduction de l'explosion combinatoire et non pas sur les méthodes de parallélisation ou encore de compilation séparée. Notons que pour le cas du *model-checker* SPIN, il est possible de compiler séparément le modèle et la propriété, mais il n'est pas possible d'effectuer une compilation séparée du modèle et du contexte.

Vérification des scénarios. Au vu du nombre de scénarios imposant, nous n'avons pas effectué la vérification sur l'ensemble des scénarios, mais nous avons considéré le temps de vérification d'un scénario comme constant afin d'extrapoler les résultats. Les résultats obtenus par la méthode OBP ne sont donc pas réels, mais des **extrapolations de résultats partiels**.

5.6.2 Résultats et discussions

Nous donnons à présent les résultats expérimentaux sur le système mono-processus du contrôle du passage à niveau tout d'abord pour l'observateur 3 puis pour l'observateur 4.

Réduction des scénarios. Sur le tableau 5.10, nous constatons que l'algorithme de réduction génère en 0.38 secondes les 3444 scénarios réduits. Une réduction de 81.7% a été obtenu (81.7% des 18900 scénarios initiaux ont été éliminés).

Nombre de scénarios			Temps de calcul des scénarios
avant réduction	après réduction	Réduction(%)	avec réduction
18900	3444	81.7 %	0.38s

FIG. 5.10: Coût du calcul de l'ensemble réduit des scénarios

Vérification de l'observateur 3 (section 5.1.2 page 66)

Le tableau de la figure 5.11 compare les résultats obtenus par la méthode classique avec SPIN, avec la méthode proposée dans ce mémoire (vérification par la décomposition de scénarios : méthode OBP). La vérification est effectuée pour l'observateur 3.

Vérification méthode classique		**Vérification méthode OBP**		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios	Total
NONE	73.2s	0.38s	5.23s	5.61s
DCOLLAPSE	42.3s			

FIG. 5.11: Résultats de la vérification sous SPIN pour l'observateur 3 : Train version 1

- La partie gauche du tableau représente le temps de la vérification par SPIN de l'observateur 3, composé avec le système et le contexte global (*model-checking* classique). Différentes options de compilations ont été utilisées : les ordres-partiels (activés par défaut) représenté par NONE et l'option COLLAPSE pour la compression des états. Le *model-checking* SPIN avec la réduction par ordres-partiels termine en 73.2s. La propriété n'est pas falsifiée. Avec l'option COLLAPSE, la vérification termine en 42.3s sans falsification de la propriété.

- Dans la partie droite du tableau la méthode présentée dans ce mémoire est utilisée (méthode OBP). Chaque scénario est composé avec le système et la propriété à vérifier. Le temps de réduction des scénarios calculés précédemment se rajoute au temps de vérification de chaque scénarios par SPIN pour obtenir le temps de calcul final de notre méthode. Ainsi, les 3444 scénarios sont vérifiés par SPIN en $5.23s$ auquel nous rajoutons les $0.38s$ (temps de calcul de la réduction des scénarios). La vérification s'effectue au total en 5.61s.

Interprétation des résultats. Nous constatons que pour cette expérimentation, la méthode OBP est plus efficace que SPIN. Nous allons maintenant expliquer pourquoi. Observons la trace obtenue par SPIN lorsque les ordres-partiels et la compression d'états sont activés sur l'automate global :

```
(Spin Version 5.2.5 -- 17 April 2010)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 468 byte, depth reached 242, errors: 0
  2117609 states, stored
  4555731 states, matched
  6673340 transitions (= stored+matched)
 11261914 atomic steps
hash conflicts:   8688535 (resolved)

Stats on memory usage (in Megabytes):
  985.521       equivalent memory usage for states (stored*(State-vector + overhead))
  132.453       actual memory usage for states (compression: 13.44%)
                state-vector as stored = 46 byte + 20 byte overhead
    8.000       memory used for hash table (-w21)
    0.305       memory used for DFS stack (-m10000)
  140.676       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 888164 11 5 3 3 3 2 1 1 2 ]

pan: elapsed time 42.3 seconds
pan: rate 50026.199 states/second
```

Nous constatons que 2117609 d'états ont été stockés, 6673340 de transitions ont été exécutées, et $140.676Mo$ de mémoire a été utilisée. Sans l'option COLLAPSE il aurait fallu $985.521Mo$ de mémoire.

Observons à présent la trace obtenue par SPIN avec la méthode OBP ; c'est à dire lorsqu'une seule trace est composée avec le système et l'observateur. Considérons par exemple le scénario

$$e1; s1; tick; e2; tick; s2; e1; s1; close; open; tick$$

En effectuant la vérification avec SPIN nous obtenons la trace suivante :

```
(Spin Version 5.2.5 -- 17 April 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 448 byte, depth reached 68, errors: 0
       18 states, stored
       14 states, matched
       32 transitions (= stored+matched)
      176 atomic steps
hash conflicts:         0 (resolved)

    2.501       memory usage (Mbyte)

pan: elapsed time 0 seconds
```

Le nombre d'états mémorisés est 18 et le nombre de transitions exécutées est 32. Seulement $2.501mo$ de mémoire ont été consommés. La vérification est quasi immédiate.

Nous constatons donc que pour un faible nombre de scénarios et un nombre important d'acteurs en parallèles, la méthode OBP est ici plus efficace que la méthode classique.

Remarque : la spécification de propriétés en SPIN. SPIN ne connaît pas encore d'implémentation de l'opérateur *weak until* (W). Ici nous obtenons une erreur si nous spécifions l'observateur 3 avec le strong until car dans ce cas là, l'opérateur until oblige à détecter la consommation d'un message *open* ; dans le cas contraire l'observateur est falsifié.

Avec l'opérateur *weak until*, il est permis cette fois-ci de ne jamais recevoir ni le message *open* ni le message *up* ; puisque nous voulons vérifier que le message *up* ne peut être consommé qu'au moment où un message *open* a d'abord été consommé.

Même si SPIN n'implante pas cet opérateur, il est toutefois possible de le retranscrire à l'aide des opérateurs restants de la manière suivante :

$$\begin{aligned} \text{p W q} &= (\Box \text{p}) \mid \text{p U q}) \\ &= \Diamond(\,!\text{p}) \rightarrow (\text{p U q}) \\ &= \text{p U } (\text{q} \mid \Box \text{p}) \end{aligned}$$

Ainsi l'observateur 3 est encodée en LTL par la formule

$$!\,\Box\,(\mathit{close} \rightarrow ((!\mathit{up})\ U\ (\mathit{open} \mid\ \Box\,(!\mathit{up}))))$$

Vérification de l'observateur 4 (section 5.1.2 page 66)

Pour l'observateur 4, nous souhaitons vérifier qu'à la fin de l'exécution du contexte, c'est à dire une fois que les trains sont passés et sortis du passage à niveau, le système sera dans un état où les barrières sont ouvertes.

La propriété encodée dans SPIN est :

$$!\,\Box\,(\mathit{down} \rightarrow ((!\mathit{end}_C)\ U\ (\mathit{up} \mid\ \Box\,(!\mathit{end}_C)))).$$

Les résultats sont résumés sur le tableau de la figure 5.12.

De la même manière que pour l'observateur 3, la vérification par la méthode OBP est plus performante pour les mêmes raisons. Notons toutefois que la vérification de l'observateur 4 nécessite plus de temps avec SPIN car le contexte doit être parcouru entièrement pour vérifier la propriété.

Vérification méthode classique		Vérification méthode OBP		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios	Total
NONE	95.4s	0.38s	4.28s	4.66s
DCOLLAPSE	67.6s			

FIG. 5.12: Résultats de la vérification sous SPIN pour l'observateur 4

5.6.3 Extension du contrôleur de passage à niveau

Expérimentation. Pour comparer la méthode proposée dans ce mémoire avec les performances du *model-cheker* SPIN, nous conservons le même modèle du contrôleur, mais nous complexifions l'environnement, en incrémentant le nombre de passage du même train T_1 sur la voie 1. Le train T_2 n'effectue toujours qu'un seul passage sur la voie 2 et reste en parallèle avec le train 1. Cela a pour conséquence d'augmenter la taille de l'acteur T_1. Nous avons effectué l'expérimentation sur l'observateur 3 et l'option des ordres-partiels ainsi que l'option COLLAPSE pour la vérification avec la méthode classique. La configuration de l'expérimentation est représentée sur la figure 5.13.

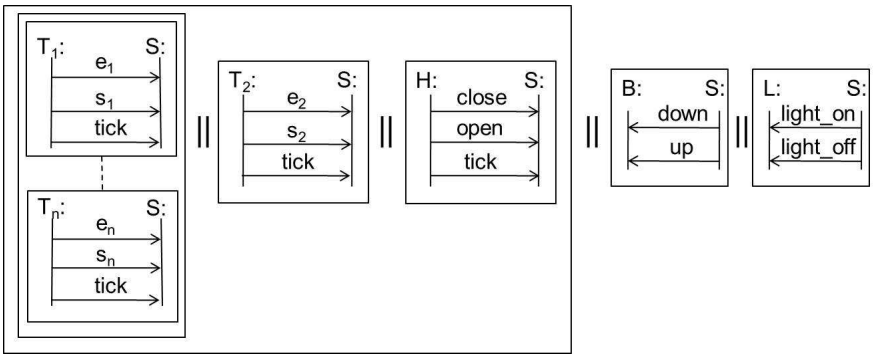


FIG. 5.13: Complexification du contexte avec n passage du Train T_1 sur la voie 1

Les résultats du temps de calcul des scénarios réduits sont résumés dans le tableau 5.14.

Nombre de passages du train	Nombre de scénarios (initiaux)	Nombre de scénarios (réduits)	Temps (sec) de réduction
1	1 680	582	0.15
2	18.10^3	4 515	0.3
3	1.10^5	19.10^3	0.63
4	371.10^3	62.10^3	1.11
5	1.10^6	164.10^3	1.8

FIG. 5.14: Temps de calcul des scénarios réduits sur 1 machine

Les résultats du temps de vérification de l'observateur 3 sur une seule machine est illustrée par le tableau 5.15 représentée par le graphique de la figure 5.16. Dans la suite, l'acronyme NA signifie *Not Applicable*.

Analyse des résultats. Nous constatons que la limite de SPIN est atteinte lorsque le passage du train 1 est simulé 4 fois en séquence par le contexte, sur la voie 1. Avec la méthode OBP, nous pouvons effectuer la vérification au-delà de 5 passages du train 1 sur la voie 1 toujours en gardant

Nb passages de trains	Tps vérif.(sec) (méthode classique)	Tps vérif. (sec) (méthode OBP sans compilation)	Tps vérif. (sec) (méthode OBP avec compilation)
1	1.53	0.91	559.63
2	39.6	6.55	4431.25
3	340	27.33	19.10^3
4	NA	89.81	62.10^3
5	NA	241.2	173.10^3

FIG. 5.15: Temps de vérification de l'observateur 3 sur 1 machine

les autres acteurs en parallèle. En interprétant le tableau précédent sous forme d'un graphique, nous obtenons les courbes de la figure 5.16.

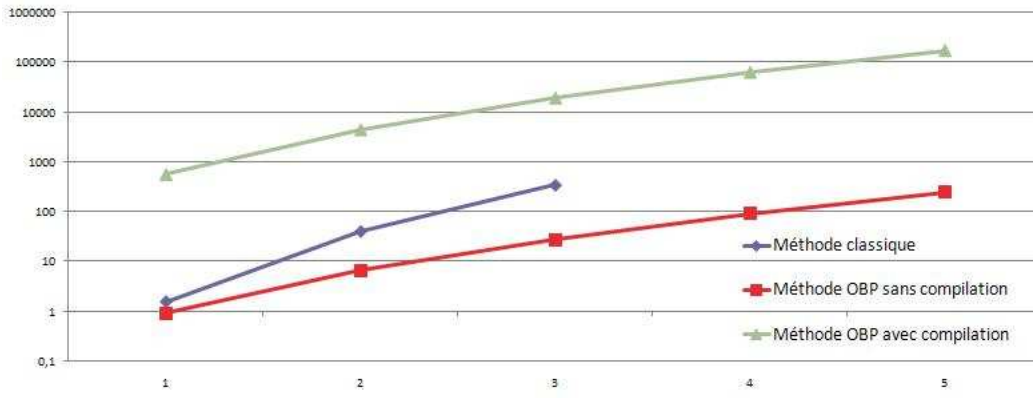


FIG. 5.16: Résultats avec 1 machine

- l'ordonnée représente le temps de la vérification en secondes
- l'abscisse représente le nombre de passages de trains sur la voie 1
- la courbe verte correspond au temps de vérification avec la méthode OBP en tenant compte du temps de la compilation et la vérification des scénarios réduits,

- la courbe bleu correspond au temps de vérification avec la méthode classique,

- la courbe rouge correspond au temps de vérification avec la méthode OBP sans prise en compte du temps de compilation de chaque scénario avec l'observateur et le système.

La figure 5.16 donne les résultats lorsque la vérification est effectuée sur une seule machine.

Parallélisation de la méthode OBP. Par nature de la méthode OBP, la vérification de chaque scénario est indépendante les unes aux autres ; les vérifications de ces scénarios peuvent être réparties sur n machines simulatanément, sans aucunes pénalités. Dans ce cas, l'utilisation de n machines permettra de diviser le temps de compilation de chaque scnéario avec le système et la propriété à vérifier par n. Nous pouvons donc proposer une première optimisation en parallélisant la vérification dès lors qu'un parc de machines est à notre disposition. Dans notre cas, nous extrapolons les résultats en considerant que le temps global des vérifications des scénarios sur n machines est divisé par n par rapport aux vérifications menées sur une seule machine. Notons toutefois que le *model-checker* SPIN ne permet pas de paralléliser la vérification des modèles.

Les tableaux des figures 5.17 et 5.19 représentent les temps de calculs de la méthode OBP lorsque la vérification est parallélisée sur 20 et 50 machines respectivement. Les courbes correspondantes sont illustrées par les figures 5.18 et 5.20 respectivement. Nous constatons qu'à partir de 50

Nb passages de trains	Tps vérif.(sec) (méthode classique)	Tps vérif. (sec) (méthode OBP sans compilation)	Tps vérif. (sec) (méthode OBP avec compilation)
1	1.53	0.19	28.12
2	39.6	0.61	221.85
3	340	1.97	966.63
4	NA	5.55	3149.85
5	NA	13.77	8663.51

FIG. 5.17: Temps de vérification de l'observateur 3 sur 20 machines

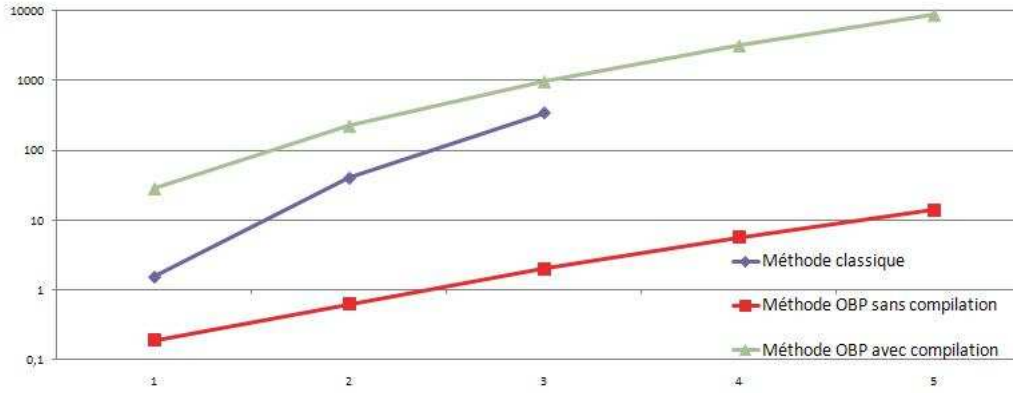


FIG. 5.18: Résultats avec 20 machines

Nb passages de trains	Tps vérif.(sec) (méthode classique)	Tps vérif. (sec) (méthode OBP sans compilation)	Tps vérif. (sec) (méthode OBP avec compilation)
1	1.53	0.17	11.34
2	39.6	0.43	88.92
3	340	1.16	387.03
4	NA	2.88	1260.6
5	NA	6.59	3466.49

FIG. 5.19: Temps de vérification de l'observateur 3 sur 50 machines

machines, la méthode OBP égale les performances de la méthode classique. Nous constatons donc que la méthode OBP est moins efficace lorsque la vérification s'effectue sur une seule machine. Néanmoins, dès lors que l'on parallélise la vérification, la méthode OBP permet de vérifier des modèles plus imposants que la méthode classique.

Bilan. Les résultats de ce chapitre montrent que le *model-checker* SPIN atteint rapidement ses limites lorsque le contexte contient des acteurs en parallèle et lorsque la longueur des acteurs augmente, générant ainsi un automate global inexploitable pour SPIN. En revanche, la décomposition de la vérification avec la méthode OBP montre que l'explosion combinatoire peut être maîtrisée. En revanche, le fait de ne pouvoir effectuer une compilation séparée en SPIN nécessite N recompilations (inutiles) du système avec les scénarios pour la vérification avec la méthode OBP. Des travaux sur l'optimisation de la compilation séparée peuvent ainsi améliorer de manière drastique les performances de la méthode OBP.

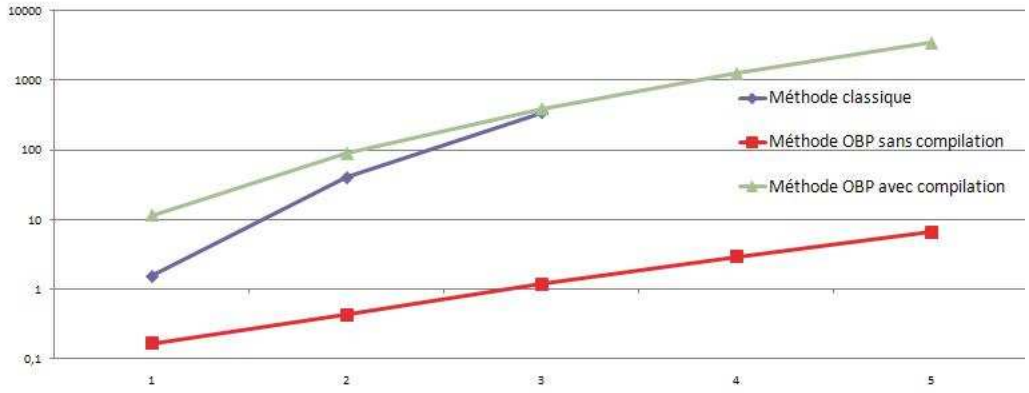


FIG. 5.20: Résultats avec 50 machines

Chapitre 6

Extension de l'approche par ordres-partiels au cas multi-processus

Sommaire

Résumé. Le chapitre précédent a défini une procédure de décision permettant de déterminer dans quelles conditions deux stimulis $a \cdot b$ et $b \cdot a$ envoyés au système ne sont pas différenciés par le comportement du système, de son environnement et de la propriété en cours de vérification. Ces relations d'indépendances ont été calculées pour le cas mono-processus SDL. Cette relation d'indépendance incorporée dans la méthodologie OBP a permis de vérifier le système de contrôle d'un passage à niveau composé d'un processus SDL.

Par la suite, on étend cette relation d'indépendance au cas multi processus où un processus peut communiquer non seulement avec son environnement mais aussi avec d'autres processus SDL. Pour cela on propose tout d'abord un nouveau cas d'étude qui étend le système du passage à niveau mono processus. Ce chapitre formalise un système SDL multi-processus et étend la relation d'indépendance mono-processus au cas multi-processus. Par la suite une nouvelle procédure de décision est décrite pour calculer les relations d'indépendances en utilisant la notion d'**ordre de l'indépendance**. Ce chapitre a été publié dans l'article [DBDB10a].

6.1 Cas d'étude multi-processus

6.1.1 Présentation générale

Cette section reprend le modèle du contrôleur d'un passage à niveau décrit pour le cas mono-processus dans le chapitre 5.1. Nous construisons une extension multi-processus qui se décompose de la manière suivante :

- un processus *Gate* qui transfère des ordres d'ouverture et de fermeture des barrières et des lumières. Ces ordres proviennent soit d'un opérateur humain (avec les événements *open* et *close*) pour des besoins de maintenances par exemple, soit du second processus ;
- P_1 et P_2 ayant pour fonction de détecter la présence des trains à l'entrée (événements e_1, e_2) et à la sortie (événements s_1,s_2) du passage à niveau sur les voies 1 et 2.

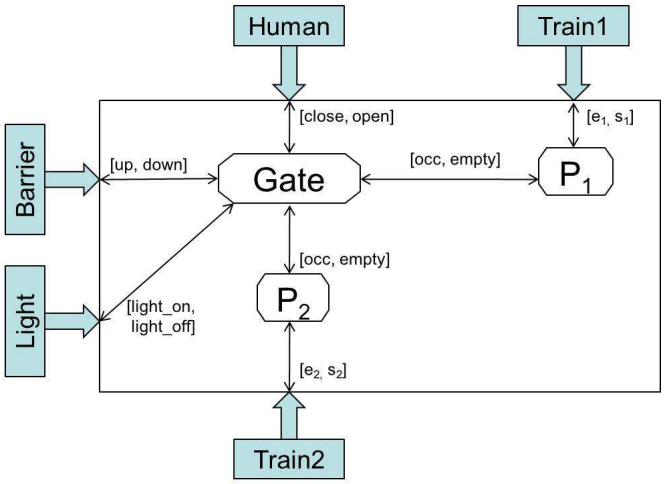


FIG. 6.1: Présentation générale du système et de son environnement

P_1 et P_2 communiquent avec *Gate* par le biais de deux messages. Si un train est détecté sur l'une des deux voies (arrivée de e_1, e_2), alors le processus correspondant P_1 ou P_2 envoie l'événement *occ* (occupé) au processus *Gate* pour lui signifier d'abaisser les barrières. A l'inverse, si un train sort (signalss_1 ou s_2), le processus correspondant (P_1 ou P_2) envoie *empty* à *Gate* qui remonte les barrières uniquement si tous les trains sont sortis. Le système multi-processus est défini par les modèles SDL représentés sur les figures 6.2 et 6.3

6.1.2 Modélisation du contexte et des observateurs

Dans cette seconde étude de cas, nous considérons le contexte de la figure 6.4. L'environnement simule l'arrivée de deux trains en parallèle sur les voies 1 et 2 (représentés par les acteurs T_1 et T_2). Un opérateur humain représenté par l'acteur H en parallèle, envoie en séquence un ordre de fermeture de barrières suivi d'un ordre d'ouverture. En parallèle, les acteurs B et L représentant respectivement les barrières et les signaux lumineux peuvent recevoir des ordres de descente (resp. levée) des barrières ainsi que des ordres de clignotement ou d'extinction des feux. Des *top* d'horloges représentés par le signal *tick* ajoutent une temporisation entre l'envoi des différents signaux.

Observateurs. Les propriétés à vérifier sont reprises du cas d'étude 1 de telle manière à homogénéiser les résultats et effectuer des comparaisons dans le chapitre 8. Les deux propriétés concernées sont :

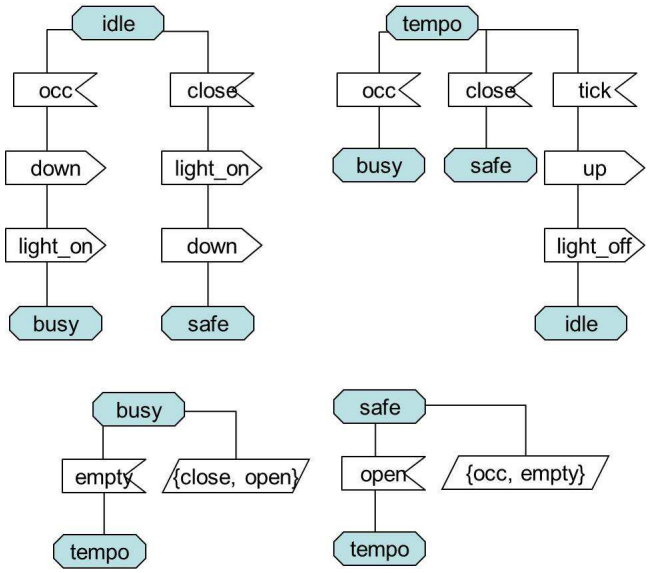


FIG. 6.2: Gate

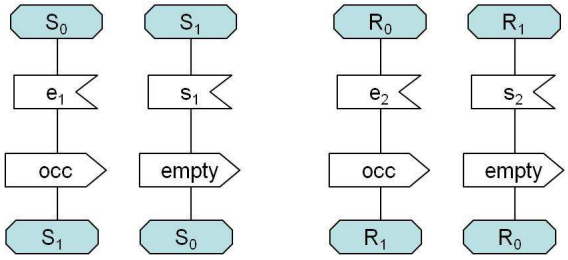


FIG. 6.3: Rail

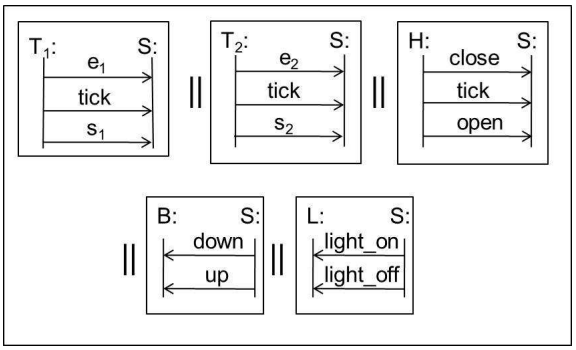
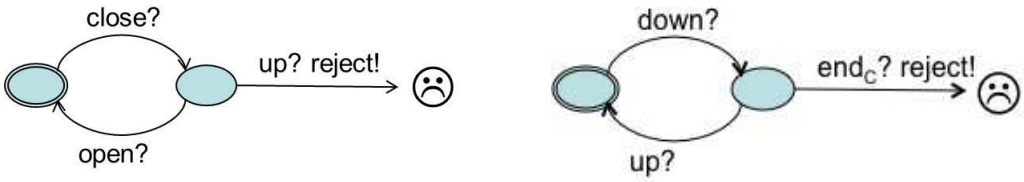


FIG. 6.4: Contexte multi-processus

- P_3 : après un ordre *close* (envoyé par le contexte), il ne peut y avoir par ordre d'ouverture *up* (décidé par le système) sans que le contexte ait auparavant fait *open* (observateur de la figure 5.3a).

- P_4 : à la fin du contexte, les barrières sont ouvertes (observateur de la figure 5.3b).



(a) Observateur Obs_3 de P_3 (b) Observateur Obs_4 de P_4

FIG. 6.5: Observateurs multi-processus

6.2 SDL multi-processus

Cette section formalise la représentation d'un système SDL multi-processus ainsi que sa sémantique afin d'étendre les relations d'indépendances présentées dans le cas mono-processus. Par la suite, nous supposons qu'un message ne peut être envoyé par un processus qu'à un seul et même destinataire (la diffusion par *broadcast* n'existe pas en SDL).

Dans la suite nous considérons un système composé de n systèmes mono-processus concurrents :

$$\mathcal{S} = \mathcal{S}_1 \parallel \mathcal{S}_2 \parallel ... \parallel \mathcal{S}_{n-1} \parallel \mathcal{S}_n$$

avec pour $i = 1 \dots n$, $\mathcal{S}_i = \langle \Sigma^i, s_0^i, T^i, Sig_{in}^i, Sig_{out}^i, Sv^i \rangle$. Nous supposons dans la suite que les Sig_{in}^i sont disjoints deux à deux (communication point à point sans *broadcast*).

La sémantique de SDL présentée dans la section 4.1.2 page 54 est étendue par la règle suivante :

$$\frac{(s_i, B_i) \xrightarrow[\sigma]{} \mathcal{S} \quad (s_i', B_i') \text{ et } \sigma_j = \sigma \backslash Sig_{in}^j, \quad j \in 1 \dots n, \quad i \neq j}{(s_1, B_1) \parallel \dots \parallel (s_i, B_i) \parallel \dots \parallel (s_n, B_n) \xrightarrow[\sigma]{} \mathcal{S} \quad (s_1, B_1) \parallel \dots \parallel (s_i', B_i') \parallel \dots \parallel (s_n, B_n)} \text{ [parSDL]}$$

Le comportement d'une composition parallèle est obtenue par l'entrelacement des comportements de chaque processus. Si le processus i évolue en produisant la sortie σ, alors ces sorties sont injecties dans les buffers des processus récepteurs et le système évolue globalement.

La sémantique de la composition d'un contexte avec un système multi-processus se réécrit par les nouvelles règles suivantes :

- règle *cp1-multi* : si $\mathcal{S}$ peut produire σ (c'est-à-dire, si l'un des processus parallèle peut produire σ), alors $\mathcal{S}$ évolue et σ est placé en fin du buffer de C.

$$\frac{(s_1, B_1) \parallel \dots \parallel (s_n, B_n) \xrightarrow[\sigma]{} \mathcal{S} \quad (s_1', B_1') \parallel \dots \parallel (s_n', B_n')}{< (C, B) | (s_1, B_1) \parallel \dots \parallel (s_n, B_n) > \xrightarrow[\sigma]{null_e} \mathcal{S} \quad < (C, B.\sigma) | (s_1', B_1') \parallel \dots \parallel (s_n', B_n') >} \text{ [cp1-multi]}$$

- règle *cp2-multi* : si C peut émettre a, C évolue et a est placé dans les buffers des processus pouvant consommer a.

$$\frac{(C, B) \xrightarrow{a!} (C', B') \text{ et } a \in Sig_{in}^i}{< (C, B) | (s_1, B_1) \parallel \dots \parallel (s_n, B_n) > \xrightarrow[null_\sigma]{a} \mathcal{S} \quad < (C', B') | (s_1, B_1) \parallel \dots \parallel (s_i, B_i \cdot a) \parallel \dots \parallel (s_n, B_n) >} \text{ [cp2-multi]}$$

- règle *cp3-multi* : enfin, si C peut consommer a, alors il évolue tandis que $\mathcal{S}$ reste inchangé.

$$\frac{(C,B) \xrightarrow{a?} (C',B')}{< (C,B_1)|(s_1,B_1) \parallel \ldots \parallel (s_n,B_n) > \xrightarrow{null_e / null_\sigma} \mathcal{S} \quad < (C',B')|(s_1,B_1) \parallel \ldots \parallel (s_n,B_n) >} \text{[cp3-multi]}$$

Ces trois règles généralisent et remplacent les règles *cp1*, *cp2* et *cp3* page 59.
Dans la suite $\mathcal{P}$ dénote l'ensemble de tous les systèmes mono-processus $(S_1, S_2, \ldots, S_{n-1}, S_n)$.

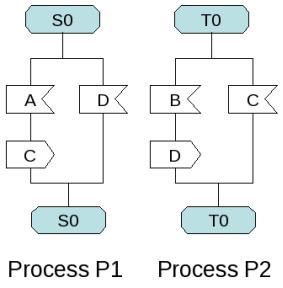
6.3 Indépendance multi-processus

On considère un système multi-processus $\mathcal{S} = \mathcal{S}_1 \parallel \mathcal{S}_2 \parallel ... \parallel \mathcal{S}_{n-1} \parallel \mathcal{S}_n$.

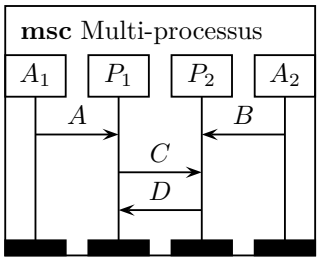
Cette partie étend le travail effectué pour les systèmes mono processus au cas multi processus. Les relations d'indépendance définies précédemment restent identiques. Toutefois la procédure de décision permettant de calculer l'indépendance entre deux événements du contexte diffère. Intuitivement, la relation d'indépendance entre deux événements peut être modifiée dès lors que les communications entre les acteurs de l'environnement et les processus sont interférées par des communications inter-processus.

Exemple. Considérons le système illustré sur la figure 6.6a. Considérons les deux acteurs prénommés A_1 et A_2 communiquant respectivement avec les processus P_1 et P_2. Dans un but de simplification on considère qu'un acteur ne communique qu'avec un seul processus. L'acteur A_1 envoie l'événement A au processus P_1 qui réagit en retournant le message C au processus P_2. L'acteur A_2 quand à lui, envoie l'événement B au processus P_2 qui réagit à son tour en envoyant le message D au processus P_1. Dans ce cas, les messages A et B sont globalement indépendants si et seulement si A et D sont indépendants dans P_1 et B et C sont indépendants dans P_2. Il en résulte que les processus voisins doivent être considérés comme des acteurs additionnels pour le calcul de la relation d'indépendance globale. Dans le cas où il existe de nombreux échanges de messages inter processus, on procède par pallier de messages échangés entre les différents processus de manière incrémentale ; c'est à dire en considérant le cas d'un seul échange inter processus, de deux échanges successifs de messages différents, et de proche en proche jusqu'à ce que tous les messages aient déjà été visités (pour éviter les échanges infinis de messages). On appellera par la suite un pallier : un ordre.

Dans la suite on calcule l'indépendance globale entre les événements du contexte par pallier en définissant l'ordre de l'indépendance : $I^n(A, B)$. Intuitivement, les messages A et B sont indépendants à l'ordre (au pallier) $n-1$ et les messages produits à l'ordre $n-1$ sont indépendants dans les processus correspondants.



(a) Système multi-processus



(b) Communication inter processus

FIG. 6.6: Exemple multi-processus

Formalisation. Dans cette partie on donne la sémantique formelle de l'ordre de l'indépendance notée I^n au niveau n. Soit $Sig_{in} = \bigcup_i Sig^i_{in}$. Dans la suite on considère la fonction : $I^n : Sig_{in} \times Sig_{in} \rightarrow \{true, false\}$ et $out^k : Sig_{in} \rightarrow 2^{Sig_{out}}$ qui est la fonction retournant l'ensemble des événements émis après la consommation de k messages. Sur la figure 6.6 l'ordre 0 correspond aux messages émis par les différents acteurs au processus.

- $out^0(A) = \{A\}$
- $out^0(B) = \{B\}$

L'ordre 1 pour l'acteur A_1 correspond aux messages envoyés dès lors qu'un message A provenant du contexte a été consommé. $out^1(A) = \{C\}$ car après la consommation du message A le message C est envoyé.

Définition de l'ordre d'indépendance. On donne à présent la définition formelle de la relation d'indépendance générale pour les systèmes multi processus qui découle de la relation d'indépendance issue du cas mono processus. Intuitivement, deux événements sont indépendants à l'ordre n après n consommation de messages si aucune relation de dépendance n'a été détectée dans la communication inter processus.

$$I^n(\alpha, \beta) \Leftrightarrow \forall k, 0 \leq k < n+1, \forall \varepsilon \in out^k(\alpha), \forall \gamma \in out^{n-k}(\beta), \forall i \in 1..n,\ (\varepsilon, \gamma) \in I_{glob}(\mathcal{S}_i)$$

Exemple. Les messages A et B sont indépendants à l'ordre 0 : $I^0(A, B)$ si ils sont globalement indépendants dans tous les processus P_1 et P_2 ; c'est à dire : $(A, B) \in I_{glob}(P_1)$ et $(A, B) \in I_{glob}(P_2)$. A et B sont indépendants à l'ordre 1 : $I^1(A, B)$ si A et B sont indépendants à l'ordre 0, si $(A, D) \in I_{glob}(P_1)$ et si $(B, C) \in I_{glob}(P_2)$.

6.4 Algorithme

Cette section présente l'algorithme permettant de calculer les relations d'indépendances dans le cas de systèmes composés de processus concurrents.

L'algorithme se décompose en deux parties. La fonction I calcule l'ordre de l'indépendance entre deux événements a et c. A et C sont deux ensembles contenant respectivement les éléments obtenus à l'ordre k. A l'ordre 0, chaque ensemble contient son propre élément a (respectivement c). Pour tous les ordres, l'indépendance entre les éléments de D et E sont calculés jusqu'à ce que les éléments de D et E soient vides.

6.5 Résultats

Dans cette section nous présentons les résultats de la vérification du système multi-processus en conservant les propriétés du cas d'étude 1, mais en considérant le nouveau contexte de la figure 6.4. Sur la figure 6.7 nous constatons que le contexte génère 1680 scénarios réduits à 866 en 0.15s.

Note. Les temps de compilation de chaque scénario avec le système et la propriété ne sont pas pris en compte dans la suite, excepté dans la partie consacrée au bilan de chapitre page 98.

Nombre de scénarios			Temps de calcul des scénarios
avant réduction	après réduction	Réduction(%)	avec réduction
1680	866	48.4 %	0.15s

FIG. 6.7: Coût du calcul de l'ensemble réduit des scénarios

$I(a, c)$:
$n \leftarrow 0$;
while $(true)$ **do**
  for $(k \leftarrow 0$ **to** $n)$ **do**
    if $(k == 0)$ **then**
      $A \leftarrow out^k(a); C \leftarrow out^{n-k}(c)$;
    else
      $A \leftarrow out^k(a) \setminus \biguplus_{l=0}^{k-1} out^l(a)$;
      /* union disjointe. A contient uniquement les événement qui occurrent à l'ordre k*/ $C \leftarrow out^{n-k}(c) \setminus \biguplus_{l=0}^{n-k-1} out^l(c)$;
    end
    if $\neg(A == \emptyset \wedge C == \emptyset)$ **then**
      $D \leftarrow out^{k-1}(a) \setminus \biguplus_{l=0}^{k-2} out^l(a)$;
      $E \leftarrow out^{n-k-1}(c) \setminus \biguplus_{l=0}^{n-k-2} out^l(c)$;
      $return\ IndepMono(D, E, \mathcal{P})$;
    end
    if $IndepMono(A, C, \mathcal{P})$ **then**
      $continue$;
    end
  end
  $n++$;
end

$IndepMono(F, G, \mathcal{P})$:
foreach $(\alpha \in F \wedge \beta \in G \wedge p \in \mathcal{P})$ **do**
  if $(\neg((\alpha, \beta) \in I_{glob}(p)))$ /*α *et* β *ne sont pas indépendants dans* p*/* **then**
    $return\ false$;
  end
end
$return\ true$;

Algorithme 5 : Algorithme de l'indépendance multi-processus

6.5.1 Vérification de l'observateur 3 (section 5.1.2 page 66)

Vérification méthode classique		**Vérification méthode OBP**		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios	Total
NONE	5.17s	0.15s	10.52s	10.67s
DCOLLAPSE	2.48s			

FIG. 6.8: Résultats de la vérification pour l'observateur 3

Analyse. La vérification exhaustive avec SPIN est effectuée efficacement en 2.48s en activant les méthodes par ordres-partiels et la compression des états. En revanche, la méthode OBP est moins efficace du fait de la réduction de scénarios moins importante (48%). La vérification s'effectue au total en 10.67s.

En observant la trace obtenue par SPIN lors du *model-checking* classique, nous obtenons :

```
Spin Version 5.2.5 -- 17 April 2010)
        + Partial Order Reduction
        + Compression

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 676 byte, depth reached 109, errors: 0
   172122 states, stored
   427031 states, matched
   599153 transitions (= stored+matched)
   295896 atomic steps
hash conflicts:     58955 (resolved)

Stats on memory usage (in Megabytes):
  114.247       equivalent memory usage for states (stored*(State-vector + overhead))
    9.939       actual memory usage for states (compression: 8.70%)
                state-vector as stored = 41 byte + 20 byte overhead
    2.000       memory used for hash table (-w19)
    0.305       memory used for DFS stack (-m10000)
   12.169       total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 49955 7 2 4 2 4 4 1 1 2 ]

pan: elapsed time 2.48 seconds
pan: rate 71419.917 states/second
```

Nous constatons que le nombre d'états mémorisés est très faible : 172 122, et la mémoire consommée est de 12.169mo. Comparons cette trace avec la vérification d'un scénario dans le cas de la méthode OBP. Considérons le scénario

$$e1; tick; s1; e2; tick; s2; close; tick; open$$

Nous obtenons la trace :

```
(Spin Version 5.2.5 -- 17 April 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 188 byte, depth reached 91, errors: 0
      984 states, stored
     1875 states, matched
     2859 transitions (= stored+matched)
     3014 atomic steps
hash conflicts:         3 (resolved)

    2.598       memory usage (Mbyte)

pan: elapsed time 0.01 seconds
```

Le nombre d'états explorés est beaucoup plus important que dans la version mono-processus, car il s'agit cette fois d'un système multi-processus. Le nombre d'états stockés est ainsi 10 fois supérieur. Nous constatons ainsi que lorsque le contexte est peu complexe et que la réduction obtenue par la méthode OBP est peu importante, le *model-checker* SPIN est plus performant.

6.5.2 Vérification de l'observateur 4 (section 5.1.2 page 66)

Les vérifications ont également été effectuées pour l'observateur 4.

Vérification méthode classique		**Vérification méthode OBP**		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios	Total
NONE	17.8s	0.15s	10.52s	10.67s
DCOLLAPSE	8.54s			

FIG. 6.9: Résultats de la vérification sous SPIN pour l'observateur 4

Nous observons que SPIN est toujours sensiblement plus performant sur les systèmes de petite taille et dont le contexte est peu complexe.

6.5.3 Extension du train multi-processus

Dans cette section, nous étendons le modèle du train multi-processus pour comparer les performances avec SPIN, entre la méthode classique et la méthode proposée dans ce mémoire (méthode OBP) dans le cas multi-processus. Les expérimentations consistent à complexifier progressivement le modèle du train (augmenter le nombre de voies par pas de 1) ainsi que son contexte (en simulant à chaque nouvelle voie ajoutée, un train sur cette voie en parallèle avec le reste du contexte), de telle manière à confronter les limites de la méthode classique sous SPIN aux limites de la méthode OBP. Ces configurations sont schématisées sur les figures 6.10 et 6.11. Dans la suite nous considérons uniquement la vérification de l'observateur 3. Les résultats sont résumés dans le tableau 6.12

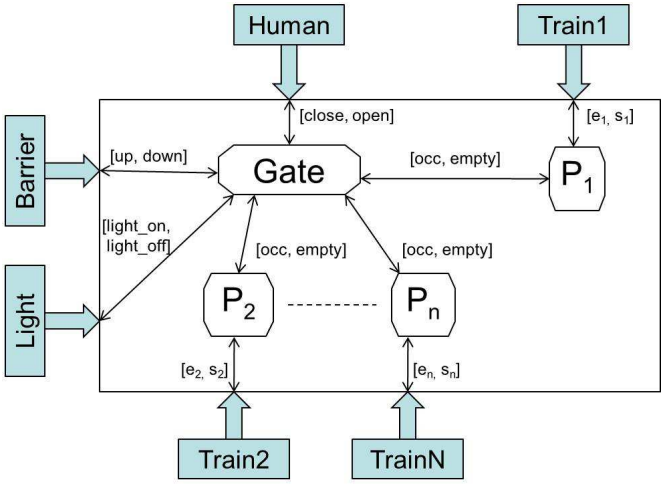


FIG. 6.10: Système multi-processus avec n voies

Ce tableau est représenté par les courbes de la figure 6.13 dans le cas d'une seule machine utilisée pour la vérification.

Nous observons que la méthode classique sous SPIN est plus performante dans le cas d'une seule machine utilisée. Par contre, la méthode classique n'aboutit pas dès lors que 4 trains ou plus sont mis en parallèle.

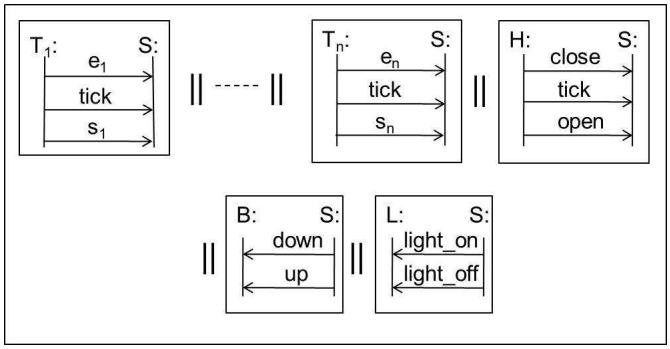


FIG. 6.11: Contexte simulant le passage de n trains concurrents sur n voies

Nb trains	Tps vérif. (sec) (méthode classique)	Scn [1]	Scn réduits	Réduction (sec)	Tps Vérif. (sec) (méthode OBP)
0	0	1	1	0	0
1	0.01	20	16	0.01	0.17
2	2.48	1 680	866	0.15	10.67
3	456	369.10^3	111.10^3	1.05	12.10^3
4	NA	168.10^6	28.10^6	123.13	9.10^6
5	NA	137.10^9	6.10^9	51.10^3	32.10^9

FIG. 6.12: Résultats de la vérification de l'observateur 3 sur 1 machine

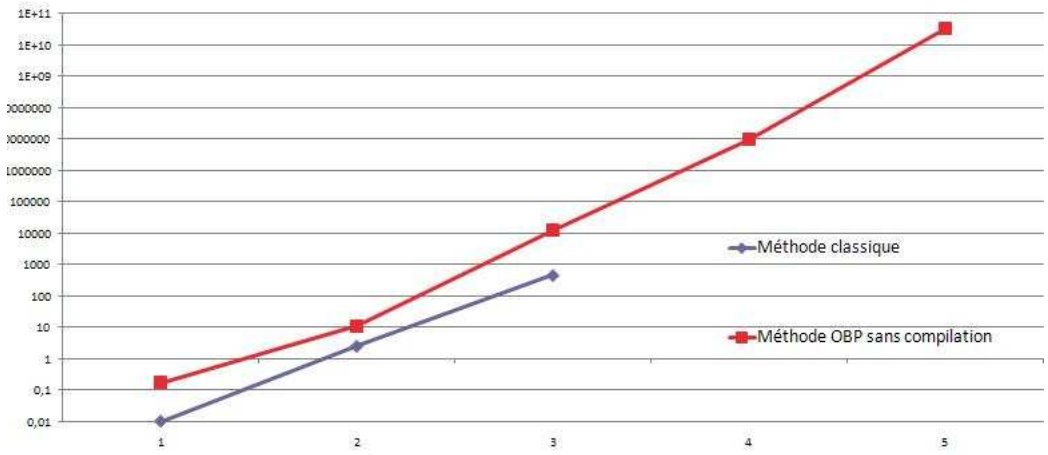


FIG. 6.13: Résultats sur 1 machine

Parallélisation de la méthode. La comparaison entre la méthode SPIN et la méthode OBP montrent que SPIN est plus performant lorsque la vérification est effectuée sur une seule machine. Néanmoins dans le cas où un parc de machine est à disposition, alors la méthode OBP égale voire dépasse les performances de SPIN (figures 6.15 et 6.17). Toutefois, dans cette expérimentation, les temps de compilation n'ont pas été pris en compte pour la méthode OBP, car ce temps est rédhibitoire pour la vérification. Nous pensons que les travaux d'optimisation sur la compilation séparée peuvent résoudre en partie cette explosion en temps.

Nb trains	Tps vérif. (sec) (méthode classique)	Scn	Scn réduits	Réduction (sec)	Tps Vérif. (sec) (méthode OBP)
0	0	1	1	0	0
1	0.01	20	16	0.01	0.01
2	2.48	1 680	866	0.15	0.68
3	456	369.10^3	111.10^3	1.05	609
4	NA	168.10^6	28.10^6	123.13	4.10^5
5	NA	137.10^9	6.10^9	51.10^3	1.10^9

FIG. 6.14: Résultats de la vérification de l'observateur 3 sur 20 machines

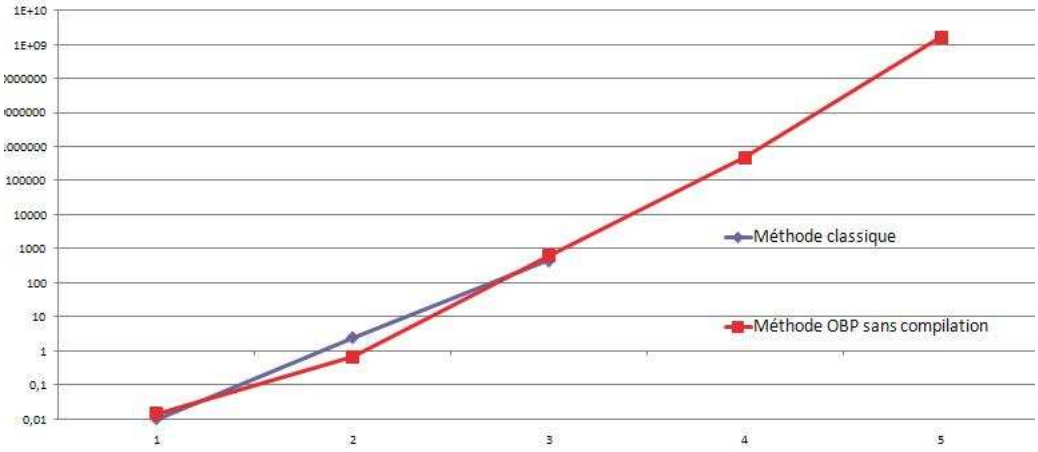


FIG. 6.15: Résultats sur 20 machines

Nb trains	Tps vérif. (sec) (méthode classique)	Scn	Scn réduits	Réduction (sec)	Tps Vérif. (sec) (méthode OBP)
0	0	1	1	0	0
1	0.01	20	16	0.01	0.01
2	2.48	1 680	866	0.15	0.36
3	456	369.10^3	111.10^3	1.05	244
4	NA	168.10^6	28.10^6	123.13	1.10^5
5	NA	137.10^9	6.10^9	51.10^3	6.10^8

FIG. 6.16: Résultats de la vérification de l'observateur 3 sur 50 machines

Bilan. Dans le cas de 3 trains en parallèle, la méthode classique effectue la vérification dans un temps raisonnable de 456 secondes. Pour égaler cette performance, et si l'on prend en compte les temps de compilation (temps de composition d'un scénario avec le système et la propriété ; nous supposonsce temps à peu près égal à 1 seconde), la méthode OBP nécéssiterait une parallélisation de la vérification sur 270 machines. En effet, le nombre de scénarios réduits pour la méthode OBP dans le cas de 3 trains est de 111.10^3, entraînant ainsi un temps de compilation de 111.10^3 secondes. A ce temps de compilation, nous devons ajouter le temps de vérification de ces 111.10^3 scénarios. Cela est effectué en 12.10^3 secondes comme indiqué sur le tableau de la figure 6.12. Le temps de vérification total pour la méthode OBP revient ainsi à $(111.10^3) + (12.10^3) = 123.10^3$ secondes ; soit environ 456 si la vérification est distribuée sur un parc de 270 machines.

Pour le cas de 4 trains concurrents, avec les mêmes suppositions et le même nombre de machines (270), nous obtenons un temps de compilation de 28.10^6 secondes auquel nous rajoutons le temps

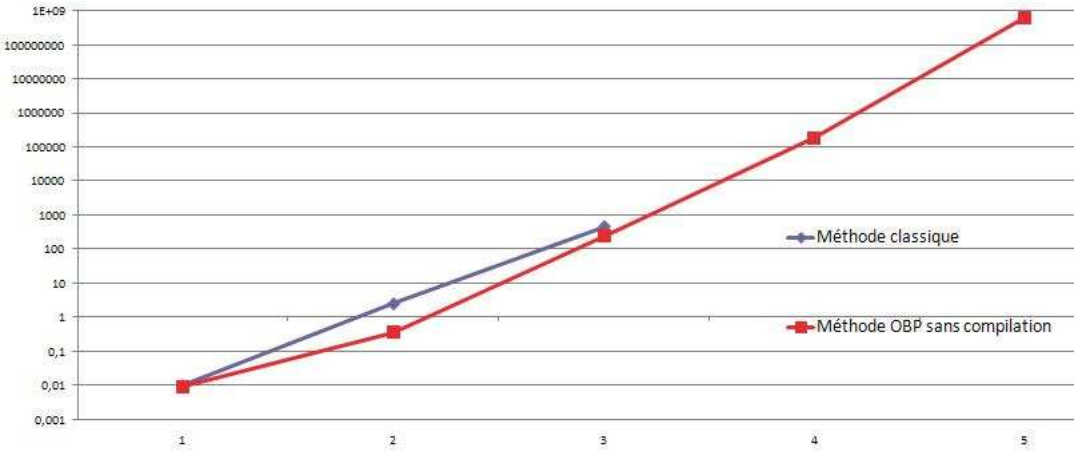


FIG. 6.17: Résultats sur 50 machines

de la vérification des 28.10^6 de scénarios qui s'élève à 9.10^6 de secondes ; soit un total de 37.10^6 de secondes sur une seule machine. La distribution de la vérification sur 270 machines revient à effectuer la vérification en 38 heures avec la méthode OBP alors que la méthode classique n'y parvient plus.

Ainsi, nous observons que la méthode OBP permet de vérifier des systèmes plus importants que la méthode classique. Le temps de vérification reste toutefois important, mais nous pouvons raisonnablement espérer une diminution drastique du temps de compilation par des méthodes d'optimisations telles que la compilation séparée ou encore la parallélisation de la vérification. Nous discutons ce point dans le chapitre 8.

Troisième partie

Cas d'étude industriels

Chapitre 7

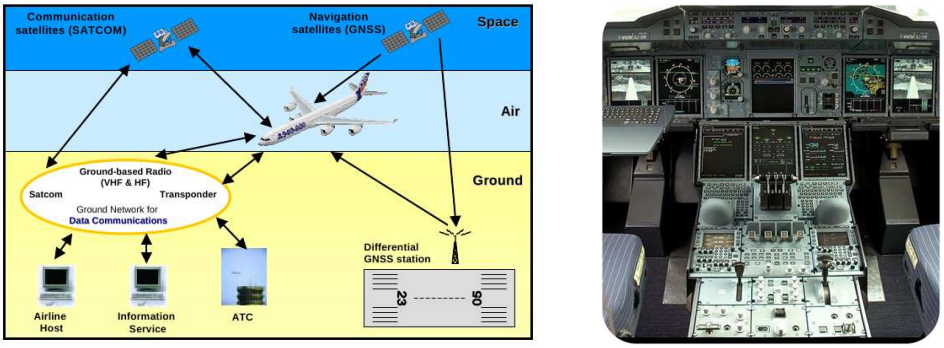
Application dans le domaine des ATC (Air Traffic Control)

Sommaire

Résumé. Dans ce chapitre, nous présentons deux cas d'étude issus d'applications avioniques. Ces modèles représentent des systèmes liés au trafic du contrôle aérien présent dans la cabine de pilotage. La première application concerne un système mono processus assurant le bon démarrage et redémarrage des applications environnantes. Ce processus est issu de l'application CPDLC permettant de remplacer les messages par voie radio en liaisons de données de telle manière à éviter les ambigüités et désencombrer les ondes radios.

La deuxième application concerne un système cette fois-ci multi-processus décrit par une application bord gérant les notifications de centre ATC et les changements de centre ATC [BBDD08]. En effet par exemple le territoire aérien Française décompose en cinq grandes zones. Chacune de ces zones est contrôlé par des centres ATC spécifiques permettant de guider les avions et de désencombrer l'espace aérien grâce aux moyens technologiques tels que les ondes radios ou encore par le biais des satellites 7.1a ; tout en ayant pour objectif d'assurer un maximum de sécurité et de ressources d'énergie. Une tour de contrôle n'exerçant son autorité que sur une parcelle du territoire Français elle doit relayer son autorité à un autre centre ATC dès lors que l'avion a quitté son domaine aérien. Cela s'effectue par le biais de l'application AFN en liaisons de données.

Dans la suite, les expérimentations sont effectuées en prenant en compte le temps de compilation dans la méthode OBP, comme il est de coutume dans les systèmes industriels.



(a) Air Traffic Control

(b) Applications bord ATC

FIG. 7.1: Les applications ATC

7.1 CPDLC : Controller Pilot Data Link Communication system

Dans cette partie nous présentons les résultats de la méthode de ce mémoire appliquée à une application avionique : le CPDLC.

7.1.1 Présentation du CPDLC

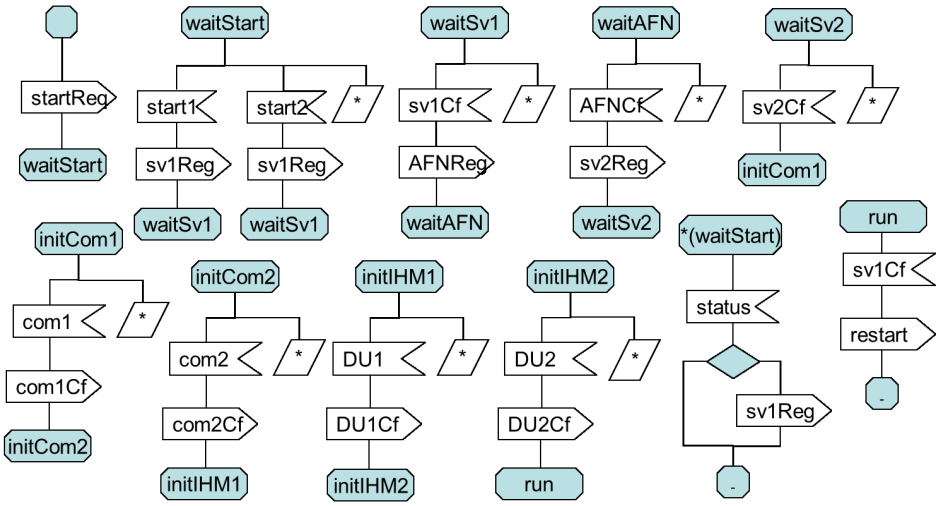
Le CPDLC est une application bit orientée devant permettre aux contrôleurs aériens et aux pilotes d'échanger des messages numériques dans le cadre du contrôle du trafic aérien. Ces messages sont visualisés sous forme écrite à bord de l'avion. Afin de mettre en œuvre les dialogues pilotes/contrôleurs, l'application CPDLC supporte une centaine de messages montant et descendant. Ces messages sont relatifs aux clairances[1] d'altitude, vitesse, changements de fréquences, rapport/confirmation de paramètres ...

Controller Pilot Data Link Communication system.

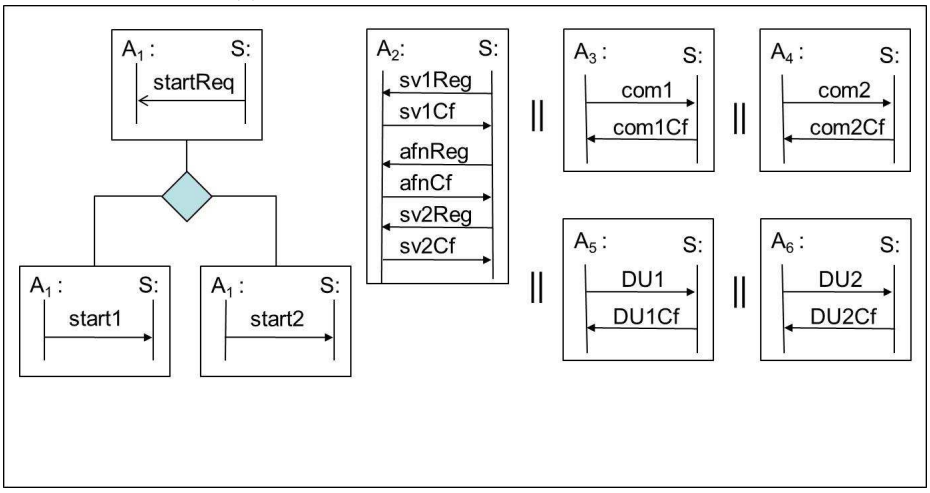
L'application CPDLC autorise les échanges de messages entre le pilote et la tour de contrôle par liaisons de données. La version simplifiée de ce système est représentée sur la figure 7.2a. Le système débute par l'envoi d'une requête de démarrage avec le message *startReq* au système global et atteint l'état *waitStart* en attente d'une réponse du système. Il y a deux manières distinctes de démarrer le système : un démarrage à froid ou bien un démarrage à chaud correspondant respectivement aux signaux *start1* et *start2*. On constate que ces deux événements mènent à un même état de destination. Par conséquent, on peut raisonnablement s'attendre que l'ordre de ces deux événements soit sans importance. La consommation de ces deux événements mène à une requête d'enregistrement du serveur 1 avec le message *sv1Reg*. Ce serveur connecte le CPDLC avec différentes applications bord tel que le FMS. Notons que ce serveur doit être démarré avant toutes les autres applications.

Une fois que le CPDLC s'est enregistré auprès du serveur 1 par la confirmation du message *sv1Cf* dans l'état *waitSv1*, le CPDLC demande l'enregistrement de l'application AFN permettant la notification d'un centre ATC avec le message *AFNReg*. Puis le processus atteint l'état *waitAFN* dans lequel il attend l'enregistrement effectif avec le message *AFNCf*. A partir de là, le CPDLC doit s'enregistrer à un serveur supplémentaire qui gère toutes les interfaces graphiques, avec le message *sv2Reg*, et doit s'attendre à la réponse *sv2Cf* pour confirmation. A ce moment, le système doit établir un protocole de communication pour pouvoir échanger des messages avec la tour de contrôle notifiée. Ceci est effectué par l'envoi des messages *com1* et *com2* aux deux couches de

[1] Les clairances correspondent à des autorisations



(a) Modèls simplifié d'un processus du CPDLC



(b) Contexte du CPDLC

FIG. 7.2: Cas d'étude du CPDLC

communication correspondantes. Le CPDLC doit ensuite attendre les confirmations des messages respectifs *com1Cf* et *com2Cf*.

Finalement, le CPDLC initialise les écrans d'affichage sur lesquels le pilote et le co-pilote pourront éditer et recevoir les messages. Cette étape est effectuée avec les messages *DU1* et *DU2* avec les confirmations respectives *DU1Cf* et *DU2Cf*. L'état **(waitStart)* signifie que toutes les transitions décrites peuvent être exécutées dans tous les états excepté dans l'état *waitStart*. En effet les services offerts par le serveur 1 peuvent potentiellement être indisponibles ou le serveur peut lui-même être indisponible. Dans ce cas, le statut du serveur change et un redémarrage du serveur est effectué. Dans l'état *running*, toutes les initialisations sont achevées. Par conséquent, lorsque le serveur 1 redémarre, le CPDLC doit prendre en compte la confirmation du redémarrage du serveur 1, et redémarrer les services lui étant associés.

La principale différence dans les clauses de sauvegarde du CPDLC réside dans le fait que tous les messages sont sauvegardés (exceptés ceux qui peuvent être consommés dans l'état courant). Ce mécanisme est présent dans pratiquement tous les états contrairement au cas d'étude du TRAIN où seulement un petite partie des messages étaient sauvegardés dans les états. En effet, le CPDLC ne peut déterminer avant l'exécution dans quel ordre les applications vont envoyer leurs requêtes et

doit par conséquent prévoir toutes les éventualités, par l'ajout de clauses de sauvegardes dans les états. Sur la figure 7.2b nous avons défini un contexte particulier composé de 6 acteurs permettant de vérifier l'initialisation correcte du système depuis son état initial jusqu'à l'état final. Nous avons pour cela identifié les acteurs suivant :

- Le système de démarrage A_1,
- les serveurs et l'application AFN représentés par A_2,
- les moyens de communications et l'affichage. représentés par A_3, A_4, A_5 et A_6.

Notons toutefois que les acteurs A_1 et A_2 sont en séquence. Initialement, le CPDLC doit prendre en compte tous les ordres possibles de requêtes après le démarrage du système. Maintenant si nous calculons les scénarios équivalents pour la vérification du système, nous pouvons constater que seuls deux scénarios sont suffisants pour la vérification des 3360 présents initialement ; ce qui représente une réduction de 99,9%. En effet, ces deux scénarios correspondent aux deux types d'initialisations possibles du système : avec *start1* ou bien *start2*. La raison de cette grande réduction vient du fait que le système dans pratiquement tous ses états sauvegarde l'intégralité des messages rendant ainsi l'ordre des messages sans importance, puisque seuls les messages autorisés dans l'état courant pourront être consommés. Par conséquent, les clauses de sauvegardes rendent le système plus linéaire et renforcent sa robustesse, puisque les clauses de sauvegarde permettent de réorganiser les messages dans un ordre pertinent.

Observateur. Ce système ayant la charge de démarrer et d'initialiser toutes les autres applications, nous souhaitons vérifier que l'initialisation s'effectue correctement ; c'est à dire que le contexte termine son exécution. La propriété LTL correspondante est définie par $\diamond end_C$ où end_C représente la détection de la fin de l'exécution du contexte. L'observateur correspondant est représenté sur la figure 7.3 et définie par la formule LTL :$\diamond end_C$.

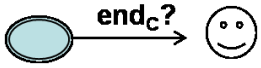


FIG. 7.3: Observateur

7.1.2 Résultats

Dans cette section nous comparons les résultats obtenus entre la méthode OBP et la méthode classique. Le tableau de la figure 7.1.2 donne les temps de calculs des scénarios réduits pour le cas du CPDLC avec 6 acteurs. Notons que le temps de compilation des scénarios pour la méthode OBP a été prise en compte.

Nombre de scénarios			Temps de calcul des scénarios
avant réduction	après réduction	Réduction(%)	avec réduction
3360	2	99.9 %	0.02s

FIG. 7.4: Coût du calcul de l'ensemble réduit des scénarios

Bilan. Nous remarquons que le temps de vérification est très rapide avec la méthode classique et la méthode OBP. En effet, le système est de petite taille et le contexte très linéaire. De plus, les réductions obtenues sur le calcul des scénarios est très important. Nous notons toutefois que la vérification avec la méthode OBP est plus efficace car l'espace d'état parcouru est très faible rendant ainsi la composition d'un scénario avec le système et la propriété quasi instantanée..

Vérification méthode classique		Vérification méthode OBP		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios (tps de compilation inclu)	Total
NONE	0.79s	0.02s	0.02s	0.04s
DCOLLAPSE	0.68s			

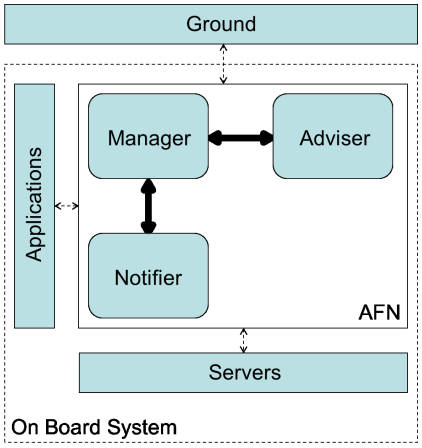
FIG. 7.5: Résultats de la vérification sous SPIN pour le CPDLC

7.2 Atis Facility Notification

L'application AFN a pour rôle de renseigner les centres ATC sur le contexte applicatif de l'avion relatif aux applications data-link embarqués (nature et version des applications disponibles sur l'avion) au travers de la procédure de contact. Elle assure aussi la fonction de relayeuse de contact qui permet au centre ATC connecté d'indiquer à l'avion l'adresse du prochain centre qu'il doit contacter.

7.2.1 Présentation de l'AFN

A présent nous appliquons la méthode OBP dans le cas d'un système composé de plusieurs processus concurrents. Nous appliquons la méthode sur le système AFN (Atis Facility Notification) permettant la notification entre un avion et le sol pour échanger des informations sur les applications de l'avion (versions des logiciels, serial number, tail number, ...). Le système AFN est composé de 3 composants comme illustré sur la figure 7.6.



- Le processus Manager crée les processus Notifier et Adviser,
- le processus Notifier établit la connexion avec le centre, ATC
- le processus Adviser gère la migration entre deux centres.

FIG. 7.6: Le système AFN et son environnement

Du point de vu du contexte, l'AFN peut communiquer avec le sol pour établir une communication avec un centre ATC. L'AFN peut communiquer avec le CPDLC pour lui notifier le centre actuel. L'AFN peut aussi échanger des messages avec différents serveurs ; par exemple, pour rassembler ou afficher des informations. Nous avons donc identifié 4 acteurs. les réductions obtenues ici sont moins impressionnantes que pour le cas du CPDLC, car les communications induites par les échanges inter-processus peuvent interférer dans le calcul des relations d'indépendance. Le système étant modélisé par un ingénieur, la complexité du système reste toutefois maîtrisée, et le nombre d'états SDL ne sera pas en général très important. La différence avec le CPDLC réside dans le fait que dans la phase opérationnelle, la succession des opérations est moins linéaire et moins déterministe. Par exemple, deux messages d'un contexte indépendants dans un processus peuvent mener à des comportements différents, une fois que les messages d'envoi du processus correspondant ont

été émis. Dans ce cas, nous devons vérifier que les messages envoyés d'un processus à un autre ne modifient pas les relations d'indépendance par propagation.

Note. Les processus SDL simplifiés du modèle AFN ainsi que le contexte de vérification se trouvent en annexe A page 127.

En appliquant la méthode de réduction étendue pour le cas multi-processus, nous obtenons une réduction de scénarios de 81.8% pour le système AFN.

Observateur. La propriété que nous souhaitons vérifier sur l'AFN est qu'une requête de changement de centre est toujours finalement effectuée. Cette propriété est encodée par l'observateur de la figure 7.7 et définie par la propriété LTL : □ (Requête → ◇ notifCentre).

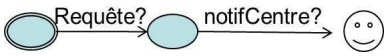


FIG. 7.7: Observateur de l'AFN

7.2.2 Résultats

Calcul des scénarios. Le tableau de la figure 7.8 donne les temps de calcul pour la réduction du contexte de l'AFN. Les 495 scénarios initiaux sont réduits à 90 scénarios en 1.1s. Nous obtenons ainsi une réduction de 81.8%

Nombre de scénarios			Temps de calcul des scénarios
avant réduction	après réduction	Réduction(%)	avec réduction
495	90	81.8 %	1.1s

FIG. 7.8: Coût du calcul de l'ensemble réduit des scénarios

Vérification Cette partie compare les résultats obtenus par le *model-checker* SPIN et par la méthode OBP. La compilation a été effectué avec la méthode de réduction par ordres-partiels activée par défaut ainsi que les options COLLAPSE et VECTORZ pour la compression des états. On s'aperçoit ainsi que même avec 13.10^3mo de mémoire la vérification par *model-checking* aurait échouée. Ainsi $42.6.10^6$ d'états ont été mémorisés ainsi que $90.8.10^6$ de transitions. L'explosion combinatoire a lieu au bout de 721s.

```
(Spin Version 5.2.5 -- 17 April 2010)
Warning: Search not completed
+ Partial Order Reduction
+ Compression

Full statespace search for:
never claim          +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 300 byte, depth reached 1190, errors: 0
 42698816 states, stored
 48198813 states, matched
 90897629 transitions (= stored+matched)
 27376788 atomic steps
```

```
hash conflicts:  64700206 (resolved)

Stats on memory usage (in Megabytes):
13030.645 equivalent memory usage for states (stored*(State-vector + overhead))
 2878.004 actual memory usage for states (compression: 22.09%)
          state-vector as stored = 51 byte + 20 byte overhead
  128.000 memory used for hash table (-w25)
    0.305 memory used for DFS stack (-m10000)
 3005.657 total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 15685672 4 3 25 16 2 3 9 2 5 2 1 ]

pan: elapsed time 721 seconds
pan: rate 59210.983 states/second
```

Parallélisation de la méthode Maintenant, considérons la vérification avec la méthode OBP par décomposition des scénarios. La vérification termine au bout de 65 161.1s en prenant en compte le temps de compilation de chaque scénario avec le système et la propriété. Nous remarquons que l'explosion combinatoire en espace de SPIN a été déplacée en explosion combinatoire temporelle par OBP. Toutefois, le temps de vérification n'est pas rédhibitoire car la méthode OBP est parallélisable sur un parc de machines. Les résultats sont résumés dans le tableau de la figure 7.9.

Vérification méthode classique		**Vérification méthode OBP**		
Option SPIN	Vérification Avec contexte global	Réduction des scénarios	Vérification avec les scénarios (tps de compilation inclu)	Total
NONE	out of memory	1.1s	65.10^3s	65.10^3s
DCOLLAPSE				

FIG. 7.9: Résultats de la vérification sous SPIN pour l'observateur 3

Si l'on effectue la vérification sur 90 machines, le temps de vérification par machine retombe à 724s. La vérification serait donc effectuée en 12 minutes.

Quatrième partie

Conclusion générale

Chapitre 8

Bilan, discussion et perspectives

Sommaire

Résumé. Dans ce mémoire, nous avons proposé une technique permettant d'optimiser la décomposition des modèles CDL en des ensemble de scénarios de contextes. La méthode initiale qui avait été proposée, au départ de ce travail de thèse, consistait à remplacer la vérification globale de propriétés, par *model-checking*, sur un modèle plongé dans son environnement, par un ensemble de N petites vérifications.

Cela est effectué tout d'abord, en restreignant le comportement du système par l'identification d'un ensemble de modèles CDL décrivant des environnements spécifiques. Chaque environnement est lui-même décomposé en un ensemble de scénarios élémentaires (ou chemins de contexte). Chaque scénario élémentaire est composé avec le modèle à valider et les propriétés à vérifier. Cette composition élimine, lors de la simulation, les comportements du système qui n'ont pas d'impacts sur la vérification d'une propriété à vérifier. Cette méthode (dite "méthode OBP") de décomposition permet d'effectuer une vérification à l'aide d'un *model-checker* sans engendrer d'explosion combinatoire. L'inconvénient de la méthode est que ce nombre de scénarios peut potentiellement exploser selon le nombre d'acteurs concurrents présents dans le contexte. Une optimisation est donc nécessaire pour restreindre le nombre de scénarios générés par l'outil OBP et permettre de mener les vérifications de chaque propriété.

Dans notre travail, nous avons montré que nous pouvons profiter de la robustesse des modèles à valider en trouvant des équivalences entre scénarios au regard des propriétés à vérifier. Nous avons développé une nouvelle relation d'indépendance d'ordres-partiels, de manière à éliminer des scénarios équivalents, en utilisant la sémantique des traces de Mazurkiewicz. Pour démontrer l'intérêt d'une telle approche, nous avons comparé notre méthode avec l'utilisation standard de l'un des meilleurs *model-checker* actuel : SPIN. Notre méthode est mise en œuvre également avec SPIN mais en prenant en compte la décomposition entre scénarios optimisée avec les calculs des équivalences. Nous avons illustré notre méthode sur des modèles SDL mono-processus et multi-processus, mais sans données ni timers. Les résultats obtenus ont montré, qu'avec cette méthode, nous arrivons à mener des preuves sur des modèles et environnements plus complexes que dans le cas d'une vérification sans décomposition et optimisation. Nous considérons que ces premiers résultats sont encourageants et doivent donc se poursuivre et faire l'objet d'extension et d'amélioration.

Dans ce chapitre, nous commentons nos résultats et tirons un bilan en proposant une analyse critique. Ensuite nous proposons des perspectives d'extensions et d'améliorations de la méthode.

8.1 Bilan et discussion

8.1.1 Travaux effectués

Dans le chapitre 5, nous avons conçu des algorithmes de décisions permettant de classer des scénarios équivalents dans un même ensemble, pour un modèle à valider mono-processus, sans données et sans temporisation. Nous avons ensuite étendu, dans le chapitre 6, cette relation d'équivalence pour le cas d'un modèle multi-processus. Nous avons illustré ces méthodes de réduction sur un cas d'étude modélisant un contrôleur de passage à niveau dans un configuration mono-processus (section 5.1), et ensuite dans une configuration multi-processus (section 6.1). La méthode proposée a été ensuite illustrée sur deux cas d'étude industriels qui sont les modèles AFN et CPDLC fournis par la société CS-SI et AIRBUS.

Dans ces expérimentations, nous avons utilisé le *model-checker* $SPIN$. Ce vérifieur a connu, durant ces 20 dernières années, de nombreuses optimisations avec notamment les méthodes par ordres-partiels appliquées sur le modèle global mais aussi la méthode de compression d'états permettant de réduire l'occupation mémoire. Nous avons comparé les performances entre la vérification par *model-checking* classique, et la vérification exploitant la décomposition de scénarios équivalents (méthode OBP). Les résultat sont décrits dans les sections (section 5.6.3 et section 6.5.3).

Pour le cas d'étude du "passage à niveau", nous avons artificiellement complexifié le contexte, en augmentant le nombre de comportements de l'environnement, c'est à dire en augmentant le nombre d'acteurs (nombre de trains et nombre de passages d'un train).

8.1.2 Résultats obtenus

Cas mono-processus. Dans le cas mono-processus (section 5.6.3), nous constatons que SPIN atteint rapidement ses limites, dès lors que le nombre de comportements du contexte augmente. Dans l'expérimentation, le contexte comprend les acteurs H, B, L, Clk, T_1 et T_2. Dans cette configuration, on fait évoluer le nombre de passages du train 1 sur la voie 1. Pour l'affichage des résultats, nous avons fait, ici, une première hypothèse simplificatrice : nous négligeons le temps de compilation du code du simulateur SPIN qui exécute la vérification. Nous rappelons, que, dans le cas de la méthode OBP, tous les scénarios générés devront être composés avec le modèle du système et les propriétés. Ceci nécessite d'avoir une phase de compilation, puis une phase de vérification pour chaque scénario généré.

On constate alors que, lorsque le train 1 passe 4 fois sur la voie, SPIN ne peut plus vérifier les propriétés avec la vérification classique, à cause d'une explosion combinatoire. En cas d'utilisation de la méthode OBP, SPIN peut continuer à vérifier les propriétés pour 4 et 5 passages de train 1. Ce qui constitue pour nous un premier apport de la méthode. Avec l'hypothèse simplificatrice ci-dessus, la méthode OBP se montre plus performante en temps de vérification.

Si nous supprimons cette hypothèse simplificatrice, en prenant en compte le temps de compilation, nous constatons, cette fois, que SPIN est beaucoup plus performante avec la méthode classique, mais tant que l'explosion combinatoire ne survient pas.

Nous avons proposé, dans ce travail, une première optimisation en parallélisant la vérification sur un parc de machines mis à notre disposition. Nous n'avons pas effectué réellement les tests sur ce parc, mais nous avons extrapolé les résultats en considerant que le temps global des vérifications des scénarios sur m machines était divisé par m par rapport aux vérifications menées sur une seule machine. Nous négligeons ici les temps de fourniture des scénarios à l'ensemble des machines et les temps de récuperation des résultats car nous estimons qu'ils sont négligeables par rapport au temps de compilation et de vérification pour un ensemble des scénarios sur une machine. Ainsi, l'expérimentation a montré que la méthode OBP (en tenant compte du temps de calcul de la compilation des scénarios) égale les performances de la méthode classique lorsque la vérification est effectuée simultanément sur 20 machines. Au-delà de 50 machines, la méthode OBP obtient des performances supérieures à celles de la méthode classique tout en faisant reculer l'instant où une explosion combinatoire survient dans la méthode classique.

Nous tirons donc un premier enseignement : **pour un modèle, un ensemble de propriétéss et un modèle de contexte CDL donnés, la parallélisation des vérifications élémentaires, appliquée sur les scénarios générés par l'outil OBP, réduit, de manière linéraire en**

nombre de machines, le temps de vérification global. Il est à noter que cette parallélisation est peu coûteuse, voire gratuite si on néglige la diffusion des scénarios à l'ensemble des machines et la récuperation des résultats. L'existence même de la décomposition du comportement d'un environnement en un ensemble de comportements nous apporte cette facilité. Il est à noter également, que dans un processus d'ingénierie industriel, l'utilisateur devra concevoir un ensemble de modèles CDL en se basant sur la connaissance du système en cours de conception et sur une métodologie outillée. Son expertise doit permettre de repousser l'explosion combinatoire en complément de la technique de parallélisation. Nous sommes convaincus qu'à l'avenir, c'est un ensemble de techniques complémentaires, judicieusement intégrées ensemble (décomposition pertinente des contextes, parallélisation des vérifications, optimisation des performances des *model-checkers*) qui permettra de traiter des modèles de tailles de plus en plus importantes. Des travaux de [GMS01] et [BBvRar] s'intéressent actuellement à la parallélisation des algorithmes de $model - checking$. Il semble que ce soit encore un problème difficile. Il sera intéressant de comparer leurs résultats aux notres quand des évaluations de ces travaux seront disponibles.

Cas multi-processus. Dans l'expérimentation multi-processus (section 6.5.3), le contexte comprend les acteurs H, B, L et n trains. Dans cette configuration, on fait évoluer le nombre de trains, de 1 à 5 sur des voies différentes et complexifions articificiellement le modèle du contrôleur. Nous notons ici que nous sommes dans le pire cas, en terme d'explosion du nombre de comportements du contexte, puisque les comportements des trains s'entrelacent librement.

Si nous négligeons le temps de compilation du code de SPIN, nous constatons que, dans un cas où le contexte est peu complexe (exemple : pour 2 trains, 1680 scénarios avant réduction), alors les probabilités d'obtenir une grande réduction sont plus faibles. Dans le cas traité ici, nous obtenons $48,4\%$ de réduction. Dans ce cas, même sans tenir compte du temps de compilation, SPIN effectue la vérification, avec la méthode classique, de manière plus efficace, tant que l'explosion combinatoire ne survient pas.

En augmentant la complexité du contexte, mais aussi du système de manière incrémentale, les performances de la méthode OBP sont toujours inférieures à celles de la méthode classique tant que l'explosion combinatoire ne survient pas. Compte tenu du nombre de scénarios générés (pour 4 trains, 28 millions de scénarios), le temps de vérification rend inexploitable la méthode OBP.

En cas de parallélisation de la vérification sur un parc de machine, la méthode classique reste plus efficace mais tant que l'explosion combinatoire ne survient pas.

Si on ne tient pas compte des temps de compilation, la méthode OBP devient plus efficace avec une distribution sur plus de 20 machines. En revanche, même si la méthode classique ne permet plus de vérifier pour 4 trains, pour cause de l'explosion combinatoire, la méthode OBP n'est toujours pas utilisable si on prend en compte le temps de compilation.

Nous rappelons que le cas d'étude traité ici nous place dans le pire des cas où le comportement des acteurs s'entrelacent. Même si la méthode OBP permet d'aller plus loin en terme d'explosion combinatoire, le temps de compilation des scénarios obtenus après réduction génère une explosion combinatoire temporelle. Dans les perspectives, nous proposerons un axe de travail qui peut contribuer à réduire ce temps de compilation.

Cas d'étude industriels. Les cas d'étude industriels, que nous avons expérimentés dans ce travail, sont représentatifs des contraintes des systèmes réels. En effet, une bonne maîtrise du système et de son environnement permet de facilement d'identifier des contextes, d'une part disjoints, car ils correspondent à des cas d'utilisation bien spécifiques (initialisation, modes opératoires spécifiques, modes dégradés, etc.) qui sont limités en nombre de comportemements. En effet, souvent nous ne sommes pas dans une configuration où le comportement de tous les acteurs de l'environnement s'entrelacent d'une manière libre comme dans le cas du cas expérimental "passage à niveau" multi-processus.

Les résultats obtenus sur les deux cas d'étude industriels montrent une réduction forte du nombre de scénarios à traiter. Les performances de la méthode OBP, même en prenant en compte les temps de compilation, sont bien meilleures que la méthode classique. Sur le cas CPDLC, l'apport de la méthode par réduction des scenarios est trés intéressante car nous sommes dans un cas où la réduction par équivalence est très forte (réduction de 99%). Pour ce cas, SPIN n'est pas confronté à

une explosion combinatoire mais le temps de vérification par la méthode clasique est supérieure au temps avec la méthode OBP. Pour le modèle AFN, SPIN est confronté à une explosion combinatoire même avec l'option de compression. En revanche, avec la méthode OBP, la vérification se termine en 724 secondes (12 minutes) dès lors que l'on la parallélise sur un parc de 90 machines.

8.2 Perspectives

Dans ce travail, nous avons décrit des résultats qui nous semblent encourageants pour une contribution à la réduction de l'explosion combinatoire pour les techniques de *model-checking*. Nous pensons que ces techniques doivent pouvoir mieux s'intégrer dans les processus d'ingénierie industrielle, et en particulier dans les activités de validation formelle de modèles logiciels menées par les ingénieurs. Mais, dans les cas industriel, la complexité des modèles est grande. Il faut donc que la communauté accadémique poursuive ses efforts pour apporter des solutions compatibles avec une pratique industrielle de la preuve formelle d'exigences. La méthodologie d'emploi de ces techniques doit être identifiée et outillée pour espérer être utilisable. Ces travaux décrits dans ce document se veulent être une contribution pour cet objectif.

L'approche que nous avons décrite doit faire l'objet de travaux complémentaires et nous décrivons quelques perspectives que nous avons identifiées.

8.2.1 Réduire l'explosion combinatoire temporelle

Parallélisation et compilation séparée. Dans notre approche, la complexité intrinsèque aux preuves de propriétés n'a pas disparu par magie, mais s'est déporté. La complexité spatiale (occupation mémoire) s'est mutée en complexité temporelle par la multiplication des preuves à exécuter sur des ensembles de scénarios générés. Cette mutation se manifeste par une surcharge de temps de calcul, lors de la compilation et la vérification pour chaque scénario du système et des propriétés à vérifier. Pour tirer avantage de la méthode de réduction des scénarios développée dans ce mémoire, nous avons proposé de mettre en œuvre une parallélisation des vérifications, pour réduire le temps global de vérification.

En complément de cette technique de parallélisation, une autre idée est de pouvoir optimiser les *model-checkers* eux-mêmes. De nombreux travaux tentent à rendre plus efficace les techniques mise en œuvre de *model-checking*. Mais une solution, qui à notre connaissance n'est pas encore disponible, est d'exploiter les techniques de compilation séparée lors de la génération des codes exécutables qui exécutent les preuves. Pour un scénario donné, le code est exécutable est généré par compilation du scénario, du modèle du système et d'une ou de plusieurs propriétés. Du fait de la génération de N scénarios, il faut donc compiler N fois. La proposition est de compiler (compilation séparée) une seule fois le modèle du système avec la ou les propriétés et de générer un premier code intermédiaire. Ensuite, chaque scénario est compilé (compilation séparée) et génère un deuxième code intermédiaire, puis donne lieu à une édition de lien des deux codes intermédiaires. Dans le cas d'un grand nombre de scénarios, cette technique réduirait ainsi le temps de compilation globale.

Regroupement de scénarios. Pour un scénario donné, le temps de compilation n'est pas proportionnel à la taille de celui-ci. Si nous considérons un ensemble de scénarios impliqués dans une compilation, nous pouvons donc espérer diminuer le temps de compilation ramené à chaque scénario. Si nous estimons que le temps de compilation d'un scénario est égal à la somme des 2 temps $t_{commun} + t_{scenario}$, où : t_{commun} est la partie du temps de compilation commun à tous les scénarios, et $t_{scenario}$ est la partie du temps de compilation spécifique à un scénario, nous pouvons dire que le temps global t_{compil} de la compilation (sans compilation séparée) pour N scénarios est égal à : $(t_{commun} + t_{scenario}) * N$. En cas de compilation séparée, le temps global $t_{compilSeparee}$ est approximativement égal à : $t_{commun} + (t_{scenario} * N)$. Si N est grand, $t_{compilSeparee}$ sera très inférieur à t_{compil}. Le temps de vérification d'un scénario peut dépendre de sa taille ; néanmoins, dans notre raisonnement, nous considérons un temps de compilation constant pour chaque scénario composé avec le même système et la même propriété.

L'idée, qui reste encore à valider formellement, est donc de regrouper d'une manière adéquate des ensembles de scénarios en des graphes de scénarios. Un graphe de scénarios est compilé et

soumis à une vérification en une seule passe. La taille du graphe, c'est à dire le nombre de scénarios contenus dans un graphe, sera choisie en fonction du nombre de scénarios atteint lorsque que le *model-checking* est confronté à une explosion combinatoire (figure 8.1). En cas de parallélisation de la vérification sur un ensemble de machines, ce sont des ensembles de graphes de scénarios qui seront diffusés, équitablement, vers chaque machine avant l'exécution de la vérification.

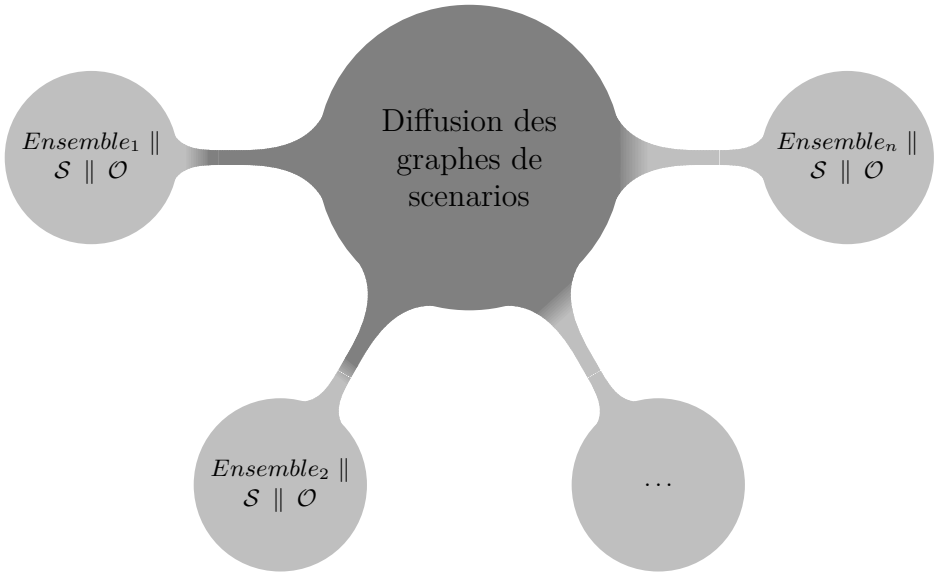


FIG. 8.1: Constitution des ensembles de graphes de scénarios avant vérification

La figure 8.1 illustre la stratégie de constitution et la diffusion des ensembles de graphes de scénarios. Soit M machines disponibles, N scénarios au total à traiter (après réduction de l'ensemble de scénarios), N_e le nombre de scénarios maximum contenu dans un graphe de scénarios pouvant être traités en une vérification par SPIN, avant d'atteindre une explosion combinatoire.

Au lieu d'effectuer la vérification des N / M scénarios sur chaque machine, nous proposons de constituer des ensembles graphes $Ensemble_1$, $Ensemble_2$, ..., $Ensemble_n$ à répartir équitablement sur les M machines.

Nous pouvons alors dire d'une manière approximative : le nombre de graphes devant être traités par une machine est de : N / (N_e * M). Ceci donne donc un temps de compilation de (N * t_{compil}) / (N_e * M)) avec t_{compil}, le temps de compilation d'un graphe constitué de N_e scénarios. Par la suite, t_{compil} pourrait être optimisée par $t_{compilSeparee}$ si la compilation séparée était utilisée.

En résumé, la mise en œuvre de la parallélisation, de la compilation séparée, ainsi que la constition des graphes de scénarios, constituent une méthode très efficace pour contribuer à faire reculer l'explosion combinatoire temporelle.

8.2.2 Prise en compte d'un ensemble SDL plus complet

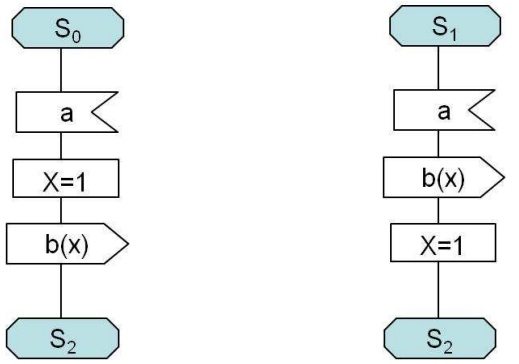
Dans les sections précédentes, nous avons effectué un bilan sur les travaux effectués et les résultats obtenus. Les réductions étaient effectués sur une restriction des modèles SDL de telle manière à démontrer l'intérêt de notre approche. Nous discutons à présent des perspectives sur le calcul des relations d'indépendances sur des modèles SDL plus complets. Notamment en prenant en compte les données dans les modèles SDL ainsi que les temporisations des modèles SDL à l'aide des timers.

Raffinement des relations d'indépendance. Dans ce mémoire, les modèles SDL ont été modélisés en faisant abstraction des données. Les données sont introduites en SDL de plusieurs façons :

- par les paramètres des signaux,

- par les affectations,
- ou encore par des opérations : des fonctions spécifiques codées en langage C et appelées par des procédures SDL.

L'ordre d'apparition de ces différents éléments dans une transition SDL peut modifier l'état de l'automate global. En effet si l'on considère les exemples de la figure 8.2a et 8.2b ; en supposant qu'avant d'atteindre S_0 la variable x ait la valeur 0, nous constatons que selon la place de l'affectation, dans la figure 8.2a le paramètre du message B aura la valeur 1 alors que sur la figure 8.2b le paramètre x du message B sera toujours à 0. Dans le cas mono-processus, le message B est envoyé à l'environnement. Par conséquent l'indépendance dépend de la gestion du message par le contexte et de l'observateur. Dans le cas multi-processus, l'ordre de l'indépendance va déterminer si le message B reste indépendant avec les autres réceptions en présence dans le nouvel état du processus SDL atteint. Les relations d'indépendances de ce mémoire prennent en compte les décisions, en considérant chaque branche de la décision comme une transition à part entière. Par conséquent, les valeurs de X ne modifieront pas les indépendances obtenues au niveau du système, mais uniquement au niveau de l'observateur et du contexte.



(a) Affectation avant

(b) Affectation après

FIG. 8.2: Exemples avec les affectations SDL

Comme décrit dans la thèse de Godefroid, le raffinement des indépendances à l'aide de données, permet d'ajouter des indépendances entre les événements. Cela est effectué au prix d'un coût de calcul supplémentaire.

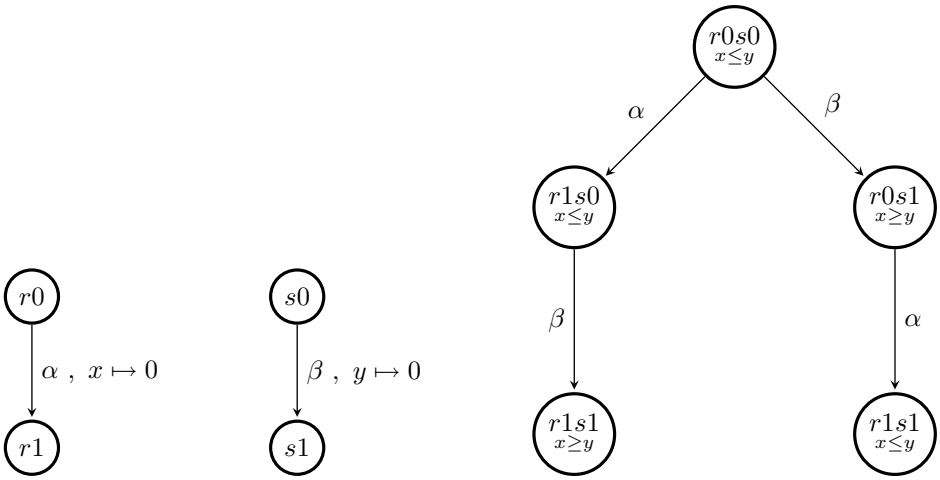
8.2.3 Extension de l'approche : prise en compte des timers

Approche classique des ordres-partiels sur les automates temporisés Dans la littérature, les travaux effectués sur les ordres-partiels sur des automates cette fois ci temporisés tiennent compte de la valeur des timers lors de l'entrelacement des transitions du système.

La sémantique du temps en SDL n'est pas définie formellement et une transformation de timers SDL dans un langage formel tel que IF a été proposée [BFG+99]. A noter que dans cette transformation les horloges sont remises à zéro à chaque fois, pour éviter des collisions entre ce même timer activé dans des instances du même processus.

Dépendance induite par les timers. Le temps peut introduire de nouvelles dépendances dans le système, dans l'environnement et dans l'observateur.

Considérons les deux processus représentés par les automates des figures 8.3a et 8.3b. Ces deux automates exécutent respectivement les actions α et β, et effectuent en même temps, une remise à zéro des horloges x (respectivement y). Le résultat de leur composition est illustré sur la figure 8.3c.



(a) automate1 (b) automate2 (c) automate3 : automate1 × automate2

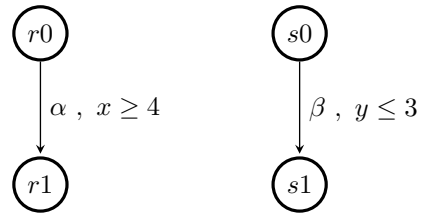
FIG. 8.3: Exemples SDL

Sur la figure 8.3c nous constatons que les actions α et β ne sont pas indépendantes car nous n'obtenons pas la forme du losange classique. Cela est dû aux valeurs des horloges x et y qui sont différentes dans les derniers états atteints. En effet, à partir de l'état r_0s_0, si nous exécutons d'abord α puis β, l'horloge x est remise à zéro avant y ; par conséquent la valeur de x sera supérieure à celle de y (car x est remise à zéro avant y). Ainsi, l'état r_1s_1 est atteint avec la condition $x \geq y$. Réciproquement si β est exécutée avant α, l'horloge y est remise à zéro avant l'horloge x. Par conséquent l'état r_1s_1 atteint possède la condition $x \leq y$. Les actions α et β indépendantes sans la prise en compte des timers, le deviennent en prenant compte la valeur de ces derniers.

Temps continu. L'introduction des horloges dans les automates produit un nombre considérable d'états supplémentaires, puisque le temps peut prendre une infinité de valeurs dans l'ensemble des réels positifs ou nuls. Par conséquent, dans un souci de déterminer une notion d'indépendance entre les transitions d'automates temporisés, une relation d'équivalence doit être déterminée. Dans les travaux issus de la littérature [Pag96, Min99, BJLY98, MMY02], les valeurs des horloges sont prises en compte, considérant des régions temporelles.

Travaux de Pagani. Les premiers travaux sur l'application des méthodes de réduction par ordres-partiels dans le domaine des automates temporisés ont débuté par la thèse de Pagani [Pag96]. Dans ses travaux, elle définit deux relations d'indépendance pour prendre en compte les indépendances temporelles. Ainsi, deux actions sont indépendantes, lorsque elles possèdent la propriété du losange, mais aussi lorsque chaque transition est indépendante avec les transitions temporelles (régions temporelles).

Pour illustrer l'idée de Pagani, considérons les automates des figures 8.4a et 8.4b.



(a) automate temporisé 1 (b) automate temporisé 2

FIG. 8.4: Exemple d'indépendance temporelle

Nous constatons que si α est exécuté en premier, alors il peut potentiellement empêcher l'exécution de β si x atteint la valeur 4. En revanche, si β est exécutée en premier, alors dans ce cas, α pourra toujours être exécuté par la suite car la valeur de l'horloge x est supérieure à celle de l'horloge y. Sur le schéma de la figure 8.5 ; la partie bleu représente l'exécution de β suivie de l'exécution de α. Dans ce cas la région temporelle bleue est atteinte sans blocage.

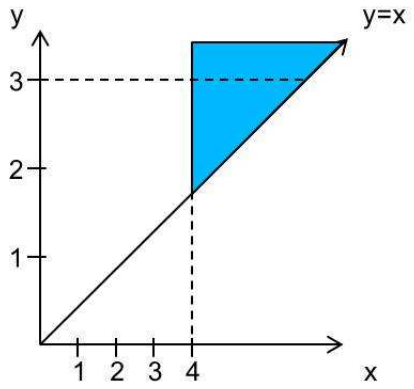


FIG. 8.5: Région temporelle atteinte après l'exécution de β suivie de α

Travaux de Bengtson. Les travaux de Bengtson repris dans la thèse de [Min99] partent du constat qu'il est difficile de déterminer une relation d'indépendance sur un réseau d'automates temporisés [AD94], car toutes les horloges se synchronisent sur une même horloge globale. Pour pallier à cette difficulté, il propose de désynchroniser toutes les horloges : en définissant une horloge à temps local pour chaque horloge ; c'est à dire se comportant indépendamment de l'horloge globale. Il montre ainsi que les relations d'indépendance peuvent être appliquées directement sur ces modèles désynchronisés. En revanche pour la composition finale il est nécessaire de resynchroniser les horloges afin d'établir les indépendances globales. Cela ajoute de la complexité dans les calculs.

Application aux modèles SDL temporisés. En SDL les timers sont implantés par des messages, à la manière des événements d'envoi et de réception classiques. Néanmoins, une valeur d'expiration du timer est définie pour chacune d'entre elles. La définition d'une relation d'indépendance peut ainsi s'effectuer de la manière suivante :

- Tout d'abord en déterminant les indépendances entre les timers et les événements,
- puis en considérant les intervalles temporels de timers.

L'introduction des notions temporelles dans le système va ainsi entraîner la temporisation des scénarios du contexte. Il faut de plus établir une sémantique du temps qui soit identique non seulement dans le système SDL mais aussi dans l'environnement CDL.

Conclusion

L'objectif que nous nous sommes fixés dans ce travail est de contribuer à une meilleure intégration des outils de *model-checking* dans les processus d'ingénierie industriel. Pour pouvoir réellement utiliser les vérificateurs, il faut pouvoir contourner l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité intrinsèque du logiciel qui doit être validé.

L'approche, dans laquelle s'insère ce travail, est basée sur la prise en compte du comportement de l'environnement du système avec lequel il interagit. Cet environnement est décrit, par des modèles CDL et exploité par l'outil OBP, sous la forme de cas d'utilisation pris en compte lors de la vérification, permettant de restreindre le comportement du modèle du système. L'objectif est de guider le *model-checker* à concentrer ses efforts non plus sur l'exploration de l'automate global mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques. La vérification d'un système est ainsi décomposé en un ensemble de petites vérifications.

La description de l'environnement et la décomposition en vérifications élémentaires sur chaque chemin apporte une réduction importante de l'explosion combinatoire en espace. Néanmoins, l'inconvénient de la méthode initiale était de générer des cas d'utilisation en très grand nombre, avec un temps de vérification rendant la méthode inutilisable.

Dans ce travail, nous avons montré que nous pouvons profiter de la robustesse des modèles à valider en identifiant des équivalences entre scénarios au regard des propriétés à vérifier. Nous avons développé une nouvelle relation d'indépendance d'ordres-partiels, de manière à éliminer des scénarios équivalents. Il s'agit de déterminer parmi les événements contenus dans l'environnement, ceux qui sont "indépendants" afin de construire ainsi la plus grande relation d'indépendance entre les événements. Cette relation permet par la suite de calculer des équivalences entre les scénarios générés par l'outil OBP par l'application de la théorie des traces de Mazurkiewvicz et par la construction des formes normales de Foata.

Pour démontrer l'intérêt d'une telle approche, nous avons comparé notre méthode avec l'utilisation standard de l'un des meilleurs *model-checker* actuel : SPIN. Nous avons illustré notre méthode sur des modèles cas d'école au format SDL, mono-processus et multi-processus. Nous avons ensuite expérimenté cette technique sur deux cas industriels du domaine aéronautique. Les résultats obtenus ont montré, qu'avec cette méthode, nous arrivons à mener des preuves sur des modèles et environnements plus complexes que dans le cas d'une vérification sans décomposition et optimisation.

Compte tenu des premiers résultats encourageants, nous avons proposé des perspectives d'optimisation notamment pour réduire les temps de compilation pour générer les codes exécutables du *model-checker*. En effet, la génération d'arbres de scénarios et la parallélisation de la vérification sur un ensemble de machine sont des axes de travaux qui doivent permettre d'optimiser les temps de vérification.

Ce travail doit se poursuivre en adaptant cette méthode à d'autres langages de modélisation. En particulier, il serait opportun d'étudier son portage sur le langage pivot FIACRE connecté à d'autres *model-checker* comme TINA ou CADP.

Table des figures

Cinquième partie

Annexe

Annexe A

Modèles SDL de l'AFN

A.1 Les processus SDL

L'AFN est décrit par 3 processus concurrents, dont le processus *manager* représenté sur les figures A.1, A.2 et A.3, le processus *logon* (figure A.4) et le processus *cad* (figures A.5 et A.6).

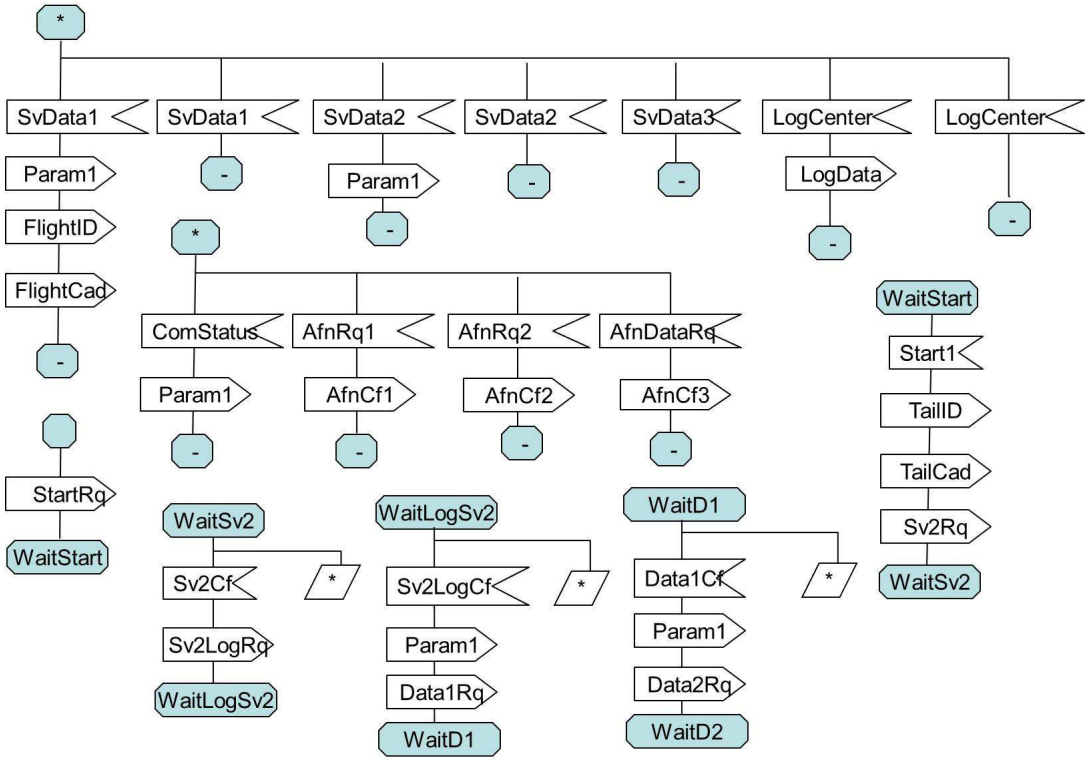


FIG. A.1: Processus manager partie 1

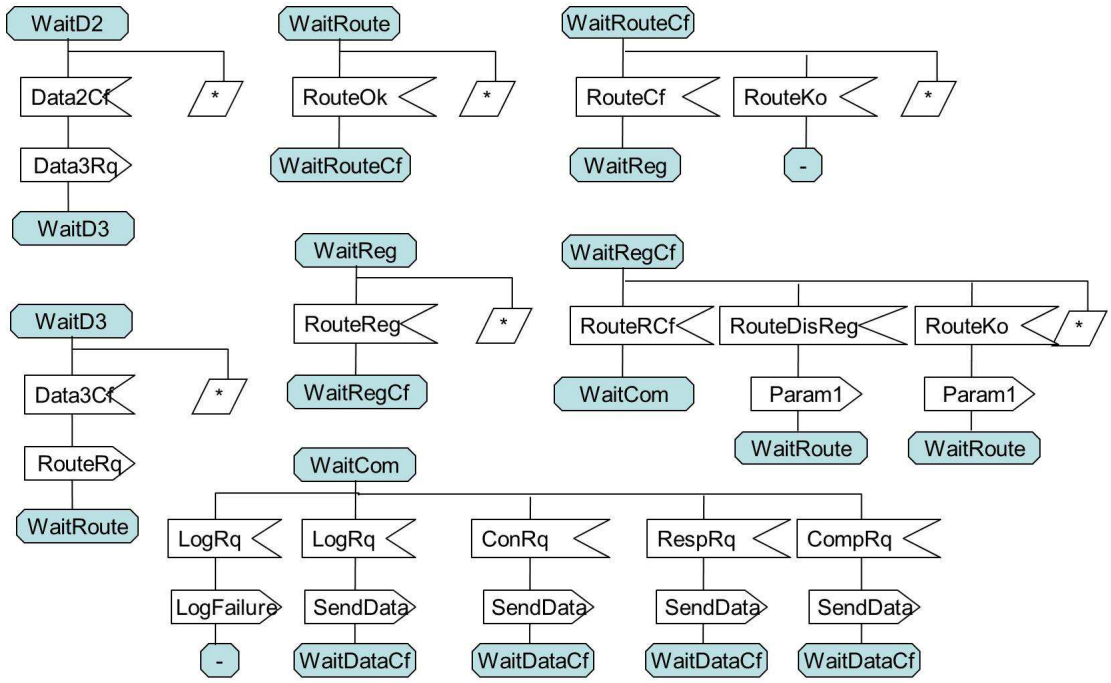


FIG. A.2: Processus manager partie 2

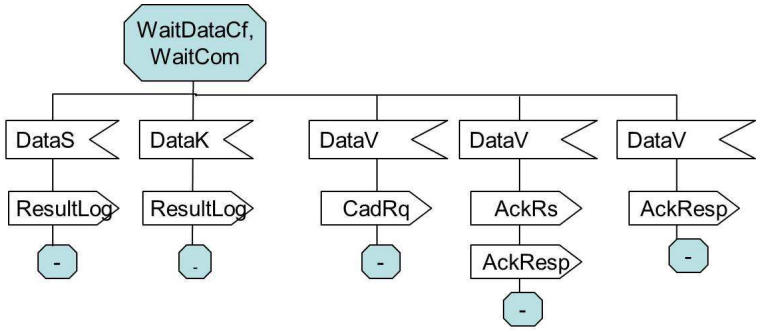


FIG. A.3: Processus manager partie 3

A.2 L'environnement de vérification de l'AFN

Le contexte de l'AFN illustré sur la figure A.7 représente 4 acteurs permettant de simuler les interactions entre le sol (A_1 et A_4), le serveur (A_3) ainsi que le CPDLC (A_2).

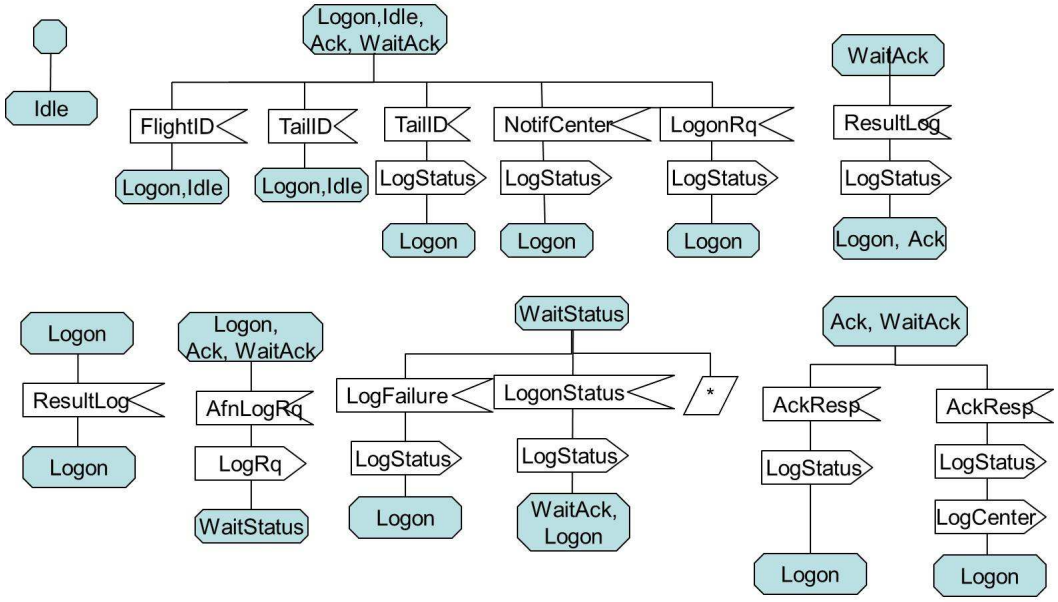


FIG. A.4: Processus Logon

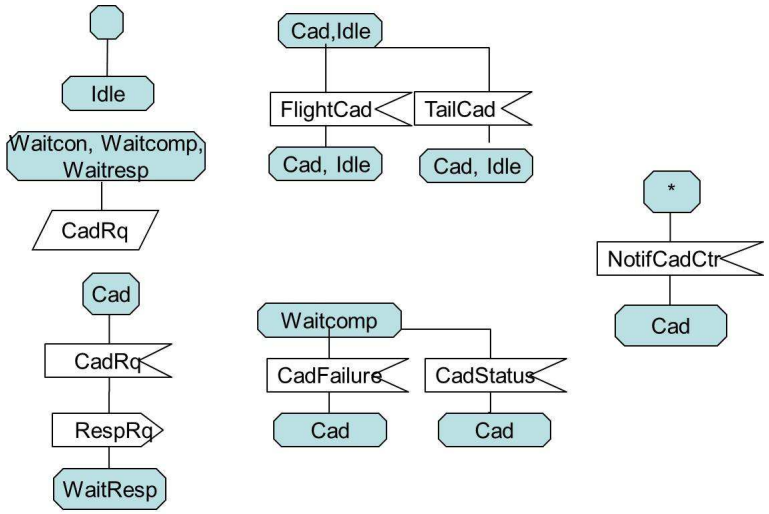


FIG. A.5: Processus Cad partie 1

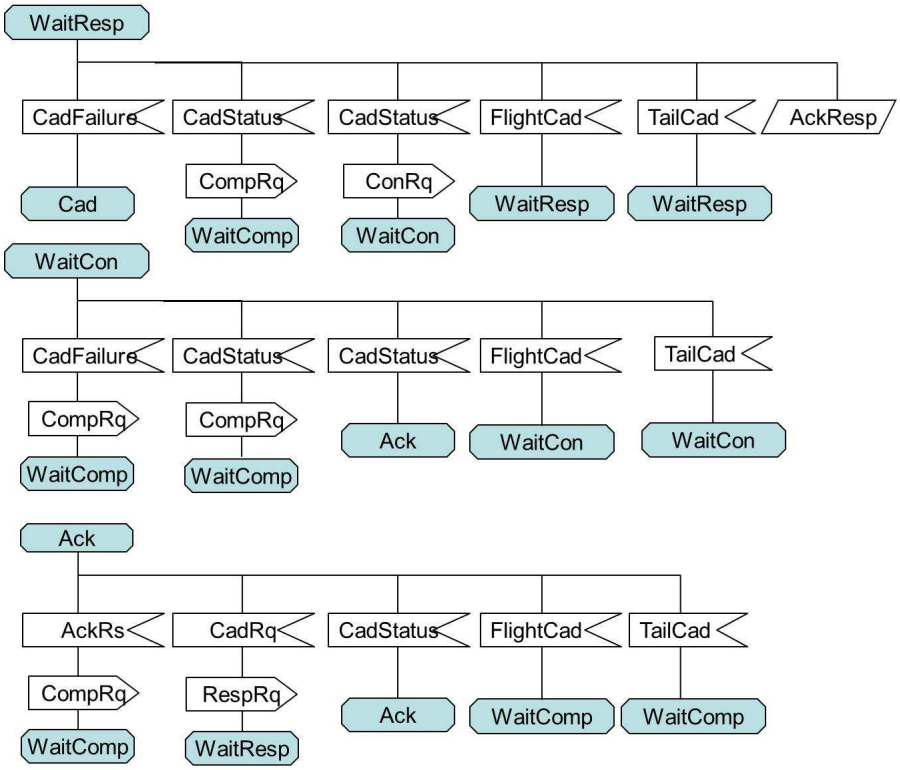


FIG. A.6: Processus cad partie 2

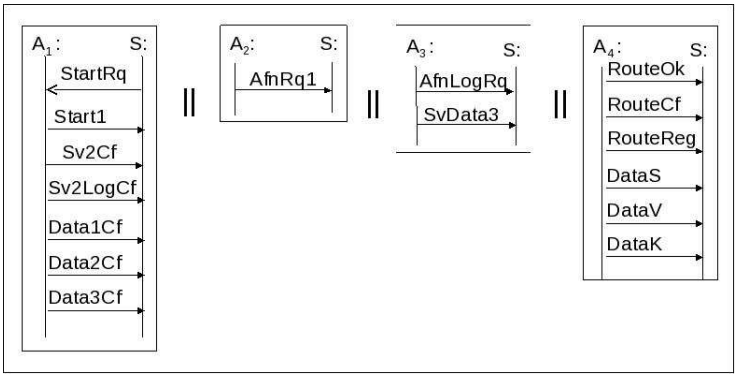


FIG. A.7: Contexte de l'AFN pour un changement de centre ATC

Bibliographie

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Computer Aided Verification*, pages 340–351, 1997.

[ABL98] Luca Aceto, Augusto Burgue no, and Kim Guldstrand Larsen. Model Checking via Reachability Testing for Timed Automata. In *TACAS '98 : Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 263–280, London, UK, 1998. Springer-Verlag.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AN82] A. Arnold and M. Nivat. Comportements de processus. *Colloque AFCET Les Mathématiques de l'Informatique*, pages 35–68, 1982.

[BB01] Twan Basten and Dragan Bosnacki. Enhancing Partial-Order Reduction via Process Clustering. In *Automated Software Engineering, ASE 2001, 16th. IEEE International Conference, Proceedings*, pages 245–253. IEEE Computer Society Press, 2001.

[BBDD08] Eric Bonnafous, Frédéric Boniol, Philippe Dhaussy, and Xavier Dumas. Experience of an efficient and actual MDE process : design and verification of ATC onboard system. In *Conference on UML & FORMAL METHODS*, Kitakyushu-city, Japan, October 2008.

[BBV04] P.-O. Ribet B. Berthomieu and F. Verdanat. The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets. *International Journal of Production Research*, 42, 2004.

[BBvRar] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE : Parallel Distributed Model Checker (Tool paper). In *Proceedings of joint HiBi/PDMC workshop (HiBi/PDMC 2010)*. IEEE, 2010. To appear.

[BDHS00] D. Bosnacki, D. Dams, L. Holenderski, and N. Sidorova. Model Checking SDL with Spin. In *TACAS '00 : Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 363–377, London, UK, 2000. Springer-Verlag.

[BFG+99] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. IF : An intermediate representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[BGP+02] Sergiy Boroday, Roland Groz, Alex Petrenko, Yves marie Quemener, and France Telecom R. Techniques for Abstracting SDL Specifications. In *Proc. Third SAM (SDL And MSC) Workshop*, pages 141–157, 2002.

[BH05] D. Bosnacki and G.J. Holzmann. Improving Spin's Partial-Order Reduction for Breadth-First Search. In *SPIN*, pages 91–105, 2005.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In *CONCUR '98 : Proceedings of the 9th International Conference on Concurrency Theory*, pages 485–500, London, UK, 1998. Springer-Verlag.

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3) :293–318, 1992.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CF69] P. Cartier and D. Foata. *Problèmes combinatoires de commutation et réarrangements*, volume 85 of *LNCS*. Springer-Verlag, Berlin, 1969.

[CFH+03] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science, 14(4)*, 2003.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5) :1512–1542, September 1994.

[CGMP99] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State Space Reduction Using Partial Order Techniques. *STTT*, 2(3) :279–287, 1999.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2000.

[CH90] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 11–23, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[CK96] Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4) :334–377, 1996.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15, 1998.

[DADB+08] Philippe Dhaussy, Julien Auvray, Stéphane De Belloy, Frédéric Boniol, and Eric Landel. Un langage de contexte de preuve pour la validation formelle de modèles logiciels. *Revue RNTI L-2*, 2008.

[Das03] Satyaki Das. *Predicate Abstraction*. PhD thesis, Stanford University, December 2003.

[DB07] Philippe Dhaussy and Frédéric Boniol. Mise en oeuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués. *revue RSTI-ISI*, 12(5) :133–157, 2007.

[DBDB] Xavier Dumas, Frédéric Boniol, Philippe Dhaussy, and Eric Bonnafous. Partial Order Application for Software Formal Verification.

[DBDB10a] Xavier Dumas, Frédéric Boniol, Philippe Dhaussy, and Eric Bonnafous. Context Modelling and Partial-Order Reduction : Application to SDL Industrial Embedded Systems. In *IEEE Symposium on Industrial Embedded Systems (SIES'10)*, 2010.

[DBDB10b] Xavier Dumas, Frédéric Boniol, Philippe Dhaussy, and Eric Bonnafous. Modélisation de Contextes et Réduction d'Ordres-Partiels pour la Vérification Efficace de Systèmes SDL. In *Conférence Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'10)*, 2010.

[DBL08] Philippe Dhaussy, Frédéric Boniol, and Eric Landel. Using context descriptions and property definition patterns for software formal verification. In *IEEE International Conference on Software Testing Verification and Validation Workshop, hosted by ICST 2008*, pages 89–96, 2008.

[DM97] Volker Diekert and Yves Métivier. Partial commutation and traces. pages 457–533, 1997.

[Dol03] Laurent Doldi. *Validation of Communications Systems with SDL*. Wiley, 2003.

[DPC+09] Philippe Dhaussy, Pierre Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation. In *MoDELS*, pages 438–452, 2009.

[DRB07] Philippe Dhaussy, Jean-Charles Roger, and Frédéric Boniol. Mise en oeuvre d'unités de preuve pour la vérification formelle de modèles. In *conférence IDM'07*, pages 101–116, 2007.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited : on branching versus linear time (preliminary report). In *POPL '83 : Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 127–140, New York, NY, USA, 1983. ACM.

[ERV96] Javier Esparza, Stefan Romer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, 1996.

[ESD97] E.A. Emerson, S. Jha, and D. Peled. Combining Partial-Order and Symmetry Reductions. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 19–34, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1) :110–121, 2005.

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In *CAV '96 : Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, London, UK, 1996. Springer-Verlag.

[GK97] Uwe Glasser and Rene Karges. Abstract State Machine Semantics of SDL. *Journal of Universal Computer Science*, 3, 1997.

[Gla99] R. J. van Glabbeek. The Linear Time – branching time spectrum I. In J. A. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 1999.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *SPIN '01 : Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 217–234, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[GP93] Patrice Godefroid and Didier Pirottin. Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract). In *CAV '93 : Proceedings of the 5th International Conference on Computer Aided Verification*, pages 438–449, London, UK, 1993. Springer-Verlag.

[GPS96] Patrice Godefroid, Doron Peled, and Mark G. Staskauskas. Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs. In *International Symposium on Software Testing and Analysis*, pages 261–269, 1996.

[GS90] Susanne Graf and Bernhard Steffen. Compositional Minimization of Finite State Systems. In *Computer Aided Verification*, pages 186–196, 1990.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking : History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[HCR01] Frédéric Herbreteau, Franck Cassez, and Olivier Roux. Application of Partial-Order Methods to Reactive Programswith Event Memorization. *Real-Time Syst.*, 20(3) :287–316, 2001.

[HMS96] Message Sequence Chart (MSC). In *ITU-T Recommendation Z.120*, Geneva, 1996.

[Hoa69] C. A. R. Hoare. *An axiomatic basis for computer programming*, volume 12. ACM, New York, NY, USA, 1969.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Hol97] G.J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.

[Hol98] G.J. Holzmann. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des.*, 13(3) :289–307, 1998.

[HP94] G.J. Holzmann and Doron Peled. An Improvement in Formal Verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[KGC04] Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *Formal Methods and Software Engineering : 6th International Conference on Formal Engineering Methods*, pages 224–238. Springer, 2004.

[KLM+98] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static Partial Order Reduction. In *TACAS '98 : Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

[KM00] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation with Partial Order Reductions for Asynchronous Communicating Systems. In *TACAS*, pages 266–282, 2000.

[Koz82] Dexter Kozen. Results on the Propositional μ-Calculus. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 348–359, London, UK, 1982. Springer-Verlag.

[KP92] Shmuel Katz and Doron Peled. *Defining conditional independence using collapses*, volume 101. Elsevier Science Publishers Ltd., Essex, UK, 1992.

[Maz86] A Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets : applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[Min99] Marius Minea. Partial Order Reduction for Model Checking of Timed Automata. In *CONCUR*, pages 431–446, 1999.

[MMY02] Eric Mercer, Chris J. Myers, and Tomohiro Yoneda. Modular Synthesis of Timed Circuits using Partial Order Reduction. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en oeuvre*. PhD thesis, Université Joseph Fourier, Grenoble, 1992.

[MP92a] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[MP92b] K. L. Mcmillan and D. K. Probst. A Technique of State Space Search Based on Unfolding. In *Formal Methods in System Design*, pages 45–65, 1992.

[NH83] Rocco De Nicola and Matthew C. B. Hennessy. Testing Equivalence for Processes. In Josep Díaz, editor, *Automata, Languages and Programming, 10th Colloquium*, volume 154 of *lncs*, pages 548–560, Barcelona, Spain, 18–22 July 1983. Springer-Verlag.

[Niv79] M. Nivat. Sur la synchronisation de processus. *Revue technique Thomson-CSF*, 11 :899–919, 1979.

[Obe99] Iulian Ober. Using GOAL observers to extends the geode simulator. Technical report, 1999.

[OMHG03] David Owen, Tim Menzies, Mats Heimdahl, and Jimin Gao. On the Advantages of Approximate vs. Complete Verification : Bigger Models, Faster, Less Memory, Usually Accurate. *Software Engineering Workshop, Annual IEEE/NASA Goddard*, 0 :75, 2003.

[Ove81] William T. Overman. *Verification of concurrent systems : function and timing*. PhD thesis, 1981.

[Pag96] Florence Pagani. Partial Orders and Verification of Real-Time systems. In *FTRTFT*, pages 327–346, 1996.

[PCDR02] Armelle Prigent, Franck Cassez, Philippe Dhaussy, and Olivier Roux. Extending the Translation from SDL to Promela. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 79–94, London, UK, 2002. Springer-Verlag.

[Pel93] Doron Peled. All from One, One for All : on Model Checking Using Representatives. In *CAV '93 : Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag.

[Pel98] Doron Peled. Ten Years of Partial Order Reduction. In *CAV*, pages 17–28, 1998.

[Pin93] Sophie Pinchinat. *Des bisimulations pour la sémantique des systèmes réactifs*. PhD thesis, Université Joseph Fourier, Grenoble, 1993.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *SFCS '77 : Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[Rog06] Jean-charles Roger. *Exploitation de contextes et d'observateurs pour la validation formelle de modèles*. PhD thesis, ENSIETA, December 2006.

[RV07] Arend Rensink and Walter Vogler. Fair testing. *Inf. Comput.*, 205(2) :125–198, 2007.

[SDL92] Specification and Description Language (SDL). In *ITU-T Recommendation Z.100*, Geneva, 1992.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation : Preliminary Edition*. PWS Publishing Co., Boston, MA, USA, 1996.

[TA04] Predrag Tosic and Gul Agha. Concurrency vs. Sequential Interleavings in 1-D Threshold Cellular Automata. In *Proc. IEEE - IPDPS 04 (APDCM Workshop), Santa Fe, New Mexico, USA, April 26-30*, 2004.

[Tar55] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2) :285–309, 1955.

[TGH95] Daniel Toggweiler, Jens Grabowski, and Dieter Hogrefe. Partial order simulation of SDL specification. In SDL'95 with MSC in CASE, editor, *Proceedings of the 7th SDL Forum*, Elservier, Amsterdam, September 1995.

[Tuo99] H. Tuominen. Embedding a Dialect of SDL in PROMELA. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 245–260, London, UK, 1999. Springer-Verlag.

[Tur37] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42 :230–265, 1937.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, London, UK, 1991. Springer-Verlag.

[Var98] Kimmo Varpaaniemi. On Stubborn Sets in the Verification of Linear Time Temporal Properties. *Lecture Notes in Computer Science*, 1420 :124+, 1998.

[Whi05] Jon Whittle. Specifying Precise Use Cases with Use Case Charts. In *MoDELS Satellite Events*, pages 290–301, 2005.

[Win86] Glynn Winskel. Event Structures. In *Advances in Petri Nets*, pages 325–392, 1986.

[Yus04] Nataliya Yustinova. *Abstractions and Static Analysis for Verifying Reactive Systems*. PhD thesis, Centrum vo or Wiskunde en Informatica, The Netherlands, November 2004.