République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Tébessa

Faculté des Sciences Exactes
et Sciences de la Nature et de la Vie

Département de l'informatique

Mémoire
Présenté en vue de l'obtention du diplôme de MAGISTER

Option : Intelligence artificielle et bases de données.

Par
Abdelbassette CHENNA

Spécification et vérification de code pour plate-forme embarquée

Date de soutenance : 12 Février 2015

Devant le jury

Mr. Abdelmajid ZIDANI	Prof. Univ. Batna	Président
Mr. Mohamed BENMOHAMMED	Prof. Univ. Constantine	Rapporteur
Mr. Samir ZIDAT	Prof Univ. Batna	Examinateur
Mr. Redha LAOUAR	Prof. Univ. Tebessa	Examinateur

ملخص

التحقق هو العملية التي تهدف إلى ضمان الأداء السليم لنظام يكون عادة آلي حرج، وهذا بضمان أن يحقق خصائص معينة . حاليا الأنظمة الآلية متواجدة في جميع المجالات: العمليات الصناعية، والسيارات، وإلكترونيات الطيران، والسكك الحديدية، الطاقة الذرية...الخ. إن وجود مثل هذه الأنظمة في برامج حرجة، إلى جانب تعقيدها المتزايد يجعل من الضروري التحقق آليا لضمان سلامة عملها . بالإضافة إلى ذلك، القيود الاقتصادية غالبا ما تتطلب إنجازا في أقصر الأوقات، الأمر الذي يزيد من الحاجة إلى طرق تحقق فعالة بأقل التكاليف. .

إن الطرق الدقيقة بما فيها طريقة "التحقق من النموذج"، وقد تم تطبيقها بنجاح في العديد من المجالات، من بينها الشبكات، وبروتوكولات الأمن وبروتوكولات الاتصالات السلكية واللاسلكية والتحكم الآلي. . والتي تسمح بإعطاء وصف لهيئة النظام المعني، ثم انطلاقا من هذه الهيئة يتم التحقق بطريق آلية من الخصائص المطلوبة قبل إنجاز النظام. إن خوارزميات "التحقق من النموذج" صممت خصيصا للتحقق التام من النظام. عندما يكون الرسم البياني للنظام ذو حجم معقول بمعنى يمكن للذاكرة المركزية استيعابه فإن طريقة التحقق من النموذج تكون جد فعالة، في حين أن غالبية الأنظمة يكون لها رسم بياني بحجم كبير جدا ففضاء الحالات الممثل للنظام هو عادة بدلالة عدد البرامج التي يتكون منها. هذا الارتفاع الواضح في حجم الرسم البياني للنظام معروفة باسم "الانفجارالاندمجي لفضاء الحالات". هذه الظاهرة و رغم العديد من سنوات البحث, مازالت تشكل العائق الرئيسي لطريقة "التحقق من النموذج".

Abstract

Verification is a process that aims to ensure the proper functioning of a system, usually automated and critical, ensuring that checks certain properties. Nowadays, the automated systems are omnipresent: industrial processes, automobile, avionics, railway, Atomic Energy ... The presence of such systems in critical applications, coupled to their complexity makes essential their verification automatically in order to guarantee the safety of their operation. Moreover, the economic constraints impose a short time of development, which makes increased the need for effective checking methods at reduced cost.

Formal methods, including the "Model-Checking", were successfully applied in several domains, like networks, security protocols, telecommunication protocols and automatic control. They can produce a description, called system specification, and then, from the specification, automatically check desired properties before implementing the system. The Model-Checking algorithms are usually designed for the total system verification. When the graph modeling the system is of reasonable size (it can be managed by the memory), the model-checking methods are very effective in the exploration of this graph and detecting errors. However, most real software systems have a states graph of very large size. Actually, the state space, representing all possible behaviors of a system or a complex protocol is usually exponential in the number of processes that contains. This important change in the size of the graph called combinatorial explosion of the size of the state space graph is still, after several years of work, the main obstacle of the automatic verification by model checking.

Résumé

La vérification est une procédure qui vise à assurer le bon fonctionnement d'un système, en général automatisé et critique, en garantissant qu'il vérifie certaines propriétés. De nos jours, les systèmes automatisés sont omniprésent : processus industriels, automobile, avionique, ferroviaire, énergie atomique... La présence de tels systèmes dans des applications critiques, couplée à leur complexité croissante, rend indispensable leur vérification de façon automatique afin de garantir la sûreté de leur fonctionnement. En plus, les contraintes économiques imposent souvent un temps de développement court, ce qui rend accru le besoin de méthodes de vérification efficaces et à coût réduit.

Les méthodes formelles, incluant la méthode du "Model-Checking", ont été appliquées avec succès dans plusieurs domaines, entre autres, réseaux, protocoles de sécurité, protocoles de télécommunication et contrôle automatique. Ils permettent de produire une description, dite spécification du système en question, puis, à partir de la spécification, vérifier automatiquement les propriétés désirées avant d'implémenter le système. Les algorithmes de Model-Checking sont, en général, conçus pour la vérification totale du système considéré. Lorsque le graphe modélisant le système est de taille raisonnable, c'est à dire qu'il peut être géré par la mémoire principale, les méthodes de model-checking sont très efficaces dans l'exploration de ce graphe et la détection des erreurs éventuelles. Cependant, la plupart des systèmes logiciels réels, possèdent des graphes d'états de très grande taille. En effet, l'espace d'états, représentant tous les comportements possibles d'un système ou d'un protocole complexe est habituellement exponentiel en le nombre de processus le comportant. Cette évolution importante dans la taille du graphe connue sous le nom d'explosion combinatoire de la taille de l'espace d'états, constitue encore, même après plusieurs années de travail, l'obstacle principal de la vérification automatique par model checking.

Table des matières

Partie 02

Partie 3

Introduction

A l'heure actuelle, les systèmes informatisés sont devenus omniprésents dans les domaines industriels et scientifiques les plus pointus (médecine, avionique, industrie automobile, secteur bancaire). Mais ils sont également présents dans les gestes les plus simples de notre vie quotidienne. Ces systèmes peuvent comporter des dysfonctionnements ou des erreurs de conception pouvant occasionner des dommages, dont les conséquences peuvent être anodines ou très graves.

Ces systèmes sont devenus de plus en plus complexes. Leur vérification automatique est, alors, devenue une obligation. L'automatisation de la vérification est soumise à plusieurs contraintes en ressources humaines (modélisation, interaction, correction) et matérielles (mémoire, temps de calcul, coût de développement...).

Aujourd'hui, grâce à diverses techniques de vérification, regroupées sous le terme de «méthodes formelles", il est possible, dans la majorité des cas, de vérifier automatiquement, et en respectant ces contraintes humaines et matérielles, si un système respecte ou non les propriétés voulues. Les méthodes formelles regroupent des techniques qui utilisent un formalisme et des outils mathématiques pour mener à bien la tâche de vérification. On commence par produire une description intermédiaire, dite *spécification*, du système que l'on souhaite développer, puis, préciser par des assertions *les propriétés* qu'il doit satisfaire. Cette spécification est basée sur un langage formel. Elle est utilisée comme référence pendant le développement du système et sert à vérifier que les propriétés seront satisfaites pour toute exécution de celui-ci.

Problématique

Les méthodes de vérification formelle comme le model checking, bien qu'efficaces et pratiques souffrent du problème connu de l'explosion de l'espace d'états, ce qui rend les ressources mémoire disponibles insuffisantes pour un stockage total et une vérification exhaustive.
On nomme explosion combinatoire de la taille de l'espace d'états le fait que le nombre des états du système augmente de façon exponentielle en fonction du nombre de ses composants (n transitions exécutables ⇒**n!** ordres d'exécution possibles et 2^n états)

Partie 1

Chapitre 1

Introduction

Chapitre 1

Introduction

Sommaire

1. Problématique de la vérification

Cette section introduit les définitions de base de la vérification. Il ne s'agit pas de définitions formelles et rigoureuses, mais il m'a paru nécessaire de fixer le sens des mots que j'utilise pour parler de ce thème. En effet, chaque communauté de recherche a sa propre culture, et par exemple, la signification du mot spécification n'est pas la même quand il s'agit de spécifier un type abstrait B, ou quand il s'agit d'énoncer une propriété temporelle que doit vérifier un circuit. Dans le cas de la spécification B, un point important est que la spécification doit être complète, c'est-à-dire décrire le comportement du programme dans tous ses cas d'utilisation. Dans le deuxième cas, il ne s'agit que d'une propriété particulière qui énonce une des caractéristiques du circuit. Pourtant, le processus de raffinement en B qui permet de générer un programme correct, peut-être considéré comme une méthode de vérification formelle, de même que la vérification d'une propriété temporelle d'un circuit. Tout en introduisant le vocabulaire, cette section présente ma vision de la vérification que ce soit de matériel ou de logiciel. Je ne m'étendrai pas sur l'intérêt de la vérification, tant il me parait évident que concevoir du matériel ou du logiciel correct est une nécessité, que ce soit sur le plan de la sécurité (e.g. logiciel ou matériel embarqué en avionique) ou de l'impact économique.

2. Définition

Pour définir le problème de la vérification, j'ai choisi la définition de Sommerville qui est simple, pragmatique et assez large pour s'appliquer aussi bien au cas du matériel que du logiciel.

3. Vérification :

La vérification est la réponse via la question

‘’Are we building the product right?’’.

Cette définition sous-entend trois éléments de base à la vérification :

- ✓ right" met en évidence une notion de référence qui énonce ce qui doit-être correct.
- ✓ building the product" énonce un processus de conception : une partie du produit est en cours de réalisation. Enfin
- ✓ Are we" met en évidence le besoin d'une méthode pour s'assurer que la réalisation en cours est correcte.

4. Spécification:

Une spécification est une propriété qui sert de référence pour la Vérification: c'est la propriété à vérifier pour le produit en cours de réalisation. Une spécification peut contenir des erreurs ou peut-être incomplète. Une même propriété peut servir de spécification à une étape donnée, et peut-être la réalisation qui doit vérifier une spécification pendant une autre étape du processus de vérification. Par exemple, la description d'un circuit au niveau

transfert de registres est la spécification du circuit vu aux niveaux portes logiques, mais c'est la réalisation du circuit vu au niveau instructions assembleur.

5. Test de logiciel :

Le test est la méthode de vérification la plus largement employée pour le logiciel : tout programmeur, de l'étudiant débutant à l'ingénieur confirmé, effectue des tests pour essayer de se convaincre que le logiciel qu'il est en train d'écrire réalise bien la fonction souhaitée. Ces tests sont généralement écrits à la main" en choisissant des valeurs pertinentes pour l'application considérée, et en vérifiant que ce qui est calculé pour ces valeurs correspond bien à ce qu'on attend. Des outils comme Junit pour java permettent de mécaniser la vérification du résultat du test, et facilitent ainsi les tests de non régression. De nombreux outils permettent de générer de façon automatique ou semi-automatique des jeux de test, en imposant certains critères, comme la couverture de chemins.

6. Simulation de matériel :

La simulation est la méthode de vérification de matériel la plus largement employée dans l'industrie. Des outils de CAO permettent de décrire des circuits dans un langage de description de matériel, VHDL et VERILOG étant parmi les plus utilisés, et de simuler leur fonctionnement. Cela signifie que l'on va exécuter un modèle du circuit, pour un certain nombre de pas de simulation (i.e.top d'horloge), et pour certaines valeurs d'entrée. Comme pour le test logiciel, des méthodes automatiques permettent de générer des jeux de test en assurant certains.
De façon synthétique, on peut dire que le test de logiciel et la simulation de matériel sont des méthodes de vérification qui valident le système pour un ensemble de valeurs et répondent donc à la question :

Un ensemble de valeurs d'entrée appliqué à la réalisation satisfait-il la Spécification ?

7. Vérification formelle :

La vérification formelle est la réponse à la question :

Est-ce que la réalisation satisfait la spécification quelles que soient les valeurs d'entrée ?

La vérification formelle introduit donc un quantificateur universel. Il apparait donc clairement que ce problème ne peut être résolu par des méthodes probabilistes, mais que cela nécessite au contraire d'utiliser des méthodes formelles basées sur des Mathématiques. Il s'agira de trouver un modèle mathématique permettant d'exprimer aussi bien la réalisation que la spécification, et fournissant des méthodes de preuve adaptées.

8. Vérification formelle complété ou incomplète :

La vérification formelle est *Complète* si la vérification est effectuée en explorant de façon exhaustive tous les états successifs possibles du système. La vérification est incomplète si la vérification ne considéré que des séquences d'état de longueur bornée. Pour la vérification

de circuits, cela signifie que l'on va vérifier la propriété pour un nombre de cycles borné ; pour la vérification de programmes, cela signifie que l'on borne le nombre de passages dans les boucles. Par exemple, le model checking ou l'induction en utilisant un démonstrateur de théorèmes sont des méthodes formelles complètes. Par contre, le *bounded model checking* est une méthode de vérification formelle incomplète.

9. Vérification formelle partielle ou totale des programmes :

La vérification partielle consiste à vérifier le programme en supposant que le programme termine, tandis que la vérification totale nécessite de montrer en plus que le programme termine. Notons que ce problème de terminaison n'a pas de sens dans la vérification de matériel puisqu'un circuit a pour vocation de fonctionner indéfiniment.

10. Procédure de décision:

Une procédure de décision est un algorithme qui termine en répondant oui ou non à un problème de décision (qui est par conséquent décidable). Les procédures de décision sont liées à une théorie, c'est-à-dire à une logique sous-jacente. Par exemple, un solveur SAT est une procédure de décision pourra théorie du calcul propositionnel qui est décidable et NP-complété. La résolution de contraintes sur domaines finis est une procédure de décision sur la théorie des entiers bornés.

11. Démonstrateur de théorèmes:

Un démonstrateur de théorèmes a pour vocation de traiter le cas des théories indécidables comme celle des entiers par exemple. Il s'agit en général d'un outil interactif (i.e. assistant de preuves) qui va utiliser des procédures de décision pour les théories décidables, et un principe de définition associé à des règles d'inférence pour dériver de nouvelles propriétés vraies à partir de propriétés déjà démontrées. Exécuter une preuve avec un démonstrateur de théorèmes peut nécessiter une intervention humaine. Par exemple, l'induction est une réglé d'inférence classique pour la théorie des entiers naturels, qui nécessite très souvent l'intervention de l'utilisateur pour indiquer sur quelle variable exécuter l'induction et quelle hypothèse d'induction utilisée.

Partie 1

Chapitre 2

Etat de l'art

Chapitre 2

Etat de l'art

Sommaire

1. Introduction

Avant d'être mis sur le marché ou à la disposition du plus grand nombre, un code est généralement soumis à de nombreux tests qui ont plusieurs objectifs. L'un d'eux est d'assurer que le code n'entraîne pas un comportement erratique et dysfonctionnel de la plate-forme sur laquelle il s'exécute. Dans le processus de déploiement traditionnel, le logiciel est remis au client final de manière plus ou moins sécurisée, dans une boite fermée, sur un cédérom, ou sur un autre support peu ou pas altérable par un tiers.

Cependant, dans les nouveaux schémas de déploiement, l'application arrive directement sur nos machines, sans que nous en aillions nécessairement conscience. Apparaît alors le problème de la sécurité de notre système : il se peut que le code que nous ayons chargé soit altéré ou bien encore qu'il s'agisse d'un virus, tel un code mobile se propageant à travers le réseau. Pour se prémunir de ce genre de danger, il est nécessaire de définir une politique de sécurité de la plate-forme et de mettre en place les mécanismes assurant cette politique de sécurité, et le processus de vérification.
De nombreux travaux sont en cours visant à montrer qu'un code suit bien une politique de sécurité donnée et ne met pas en danger la sécurité intrinsèque de la plate-forme d'exécution.

Le but n'est pas seulement de certifier qu'un programme effectue bien la tâche pour laquelle il a été conçu mais aussi d'assurer qu'il ne menace pas la sécurité de la plate-forme en effectuant des opérations illégales.

La vérification est nécessaire lorsque du code est chargé dans une plate-forme autre que la plate-forme de développement. Nous avons affaire à du code mobile. Il est nécessaire d'assurer la sécurité de la plate-forme, de ses données et des autres applications qui peuvent être présentes sur la plate-forme.
Différentes techniques de vérification sont aujourd'hui disponibles, La sécurité qui est apportée par ces nouvelles techniques peut prendre la forme d'une preuve formelle, d'une annotation de typage ou d'une autre forme de certificat ou d'annotation

2. Taxonomie des techniques de vérification et de validation

Plusieurs techniques de vérification sont applicables sur les systèmes informatiques. Elles peuvent être classifiées en deux grandes catégories. La première concerne les techniques semi-formelles, c'est-à-dire que, malgré une définition rigoureuse, il n'est pas possible de raisonner mathématiquement sur les propriétés de la spécification (consistance, correction, etc.), car ces méthodes reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise. La seconde regroupe les méthodes qui font appel à la logique et aux mathématiques pour effectuer cette vérification. Nous nous intéressons, dans les travaux présentés dans cette thèse, aux méthodes formelles.

2.1 Techniques de vérification et de validation semi-formelle

Usuellement, plusieurs techniques de vérification et de validation sont exploitées au sein de l'industrie du logiciel, notamment celles qui reposent sur les tests.

2.1.1 Vérification par tests

Un test est une technique de validation et de vérification par exécution d'une partie ou de la totalité du logiciel. En pratique, un test est effectué en deux phases : la construction des tests et l'exécution de ces tests. La construction des tests produit, manuellement ou automatiquement, une suite de tests composée d'un ensemble de cas de test. Un cas de test est un ensemble d'interactions avec le système décrivant les entrées transmises au système et les sorties attendues. En fonction de l'objectif des tests, ils peuvent être classés selon [01] en :

- Tests de conformité dont le but est de s'assurer de la conformité d'un système par rapport à son modèle (abstraction du système) vis-à-vis des exigences définies ;
- Tests de robustesse dont l'objectif est d'analyser le comportement du système dans un environnement dont les conditions ne sont pas prévues parla spécification ;
- Tests de performance dont l'objectif est de mesurer des paramètres de performance du système ;
- Tests d'interopérabilité dont le but est d'évaluer la capacité des différents composants du système à échanger des informations.

Un cas de test doit comporter les valeurs des paramètres et des résultats attendus ainsi que les critères pour décider si le résultat est conforme à la cible attendue. Ces tests doivent être les plus exhaustifs possibles pour couvrir toutes les exigences du système. Or, il n'est évidemment pas possible, dans le cas d'applications complexes où le nombre de cas possibles est en général exponentiel par rapport à la taille du système, d'effectuer un test complet car le nombre de cas à explorer est généralement grand et potentiellement infini.
Les tests peuvent également être classés en tests unitaire, d'intégration, fonctionnelle et de déploiement.

Test unitaire : il s'agit d'un test qui vérifie un module indépendamment du reste du système, afin de s'assurer qu'il répond à ses spécifications. Ces dernières sont écrites dans le cadre de

sa conception par raffinement des exigences fonctionnelles du système. Le test unitaire est considéré comme essentiel notamment dans les applications critiques. Lors de la modification du code, les tests unitaires doivent être rejoués pour vérifier qu'il n'y a pas eu de dysfonctionnement introduit par les modifications (test de non régression).

Test d'intégration : il s'agit d'un test qui se déroule après les tests unitaires, c'est-à-dire lorsque chaque module a été vérifié indépendamment. Les tests d'intégration ont pour but de vérifier que toutes les parties développées indépendamment fonctionnent bien ensemble de façon cohérente vis-à-vis des exigences écrites dans le cadre de la conception.

Test fonctionnel : il s'agit d'un test qui vérifie l'adéquation du système aux contraintes définies dans les spécifications. Les tests fonctionnels évaluent la réaction du logiciel par rapport aux données d'entrée.

Test de déploiement : il s'agit d'un test qui analyse le comportement du logiciel une fois mis en place dans l'environnement réel d'exploitation.

2.1.2 Vérification par relecture

La vérification par relecture consiste à relire les éléments qui constituent le système à vérifier par rapport aux documents des exigences. La relecture concerne la vérification de la correspondance entre ce qui est requis dans les documents et ce qui est effectivement réalisé dans le système. La relecture doit généralement être réalisée de manière indépendante, c'est-à-dire que les personnes qui font la relecture ne sont pas celles qui ont développé le système ou qui ont écrit les exigences.

2.2 Techniques de vérification formelle

Les méthodes de vérification formelle désignent un ensemble de méthodes fondées sur les mathématiques et la logique pour assurer qu'un système informatique est conforme à ses spécifications. Ces dernières doivent être décrites dans un langage défini mathématiquement (syntaxe et sémantique), nous en citons ici quelques-unes.

La vérification formelle des compilateurs ou des transformations de modèles(ou de programmes) est un domaine à part entière depuis le milieu des années 60 [02].L'ensemble de ces travaux sont répertoriés dans [03].

Les méthodes formelles ont connu une grande maturité cette dernière décade. Il est possible de nos jours de spécifier et vérifier complètement un logiciel par des techniques formelles et non sur une abstraction du système. En outre, les microprocesseurs sont également de plus en plus vérifiés formellement. Cependant, la vérification formelle du code source ne garantit pas à elle seule la correction de la compilation. Or, le compilateur est également un programme qui peut introduire des erreurs lors de la compilation en générant un code cible erroné à partir d'un programme source correct.

La vérification d'un compilateur consiste à s'assurer que ce dernier ne produise pas de code cible erroné à partir d'un programme source correct.

Plusieurs techniques de vérification formelle existent. Nous distinguons parmi elles quelques familles telles que : la vérification de modèles (model checking), l'analyse statique et le développement prouvé.

2.2.1 Génération automatique de tests et oracles

Il existe plusieurs méthodes de génération automatique de tests, citons par exemple [04]. Au lieu d'écrire lui-même manuellement les tests, l'utilisateur final fournit un générateur de tests qui exploite une spécification formelle du système et un oracle pour décider si le résultat de l'application des tests est correct par rapport au résultat attendu. Comme indiqué précédemment, les tests générés ne peuvent pas être exhaustifs sauf pour des systèmes très simples.

2.2.2 Vérification à l'exécution

La vérification à l'exécution [05] détermine lors de l'exécution d'un système si celui-ci satisfait un ensemble d'exigences. Ces dernières sont utilisées pour représenter les comportements désirés du système ou, au contraire, pour mettre en évidence ceux qui peuvent entraîner une défaillance logicielle. Ces exigences sont souvent exprimées par une spécification formelle. Une représentation abstraite est extraite de l'exécution du système pour être transmise à un oracle qui détermine si la propriété considérée est satisfaite. La détection d'erreurs par la technique ne doit pas influencer l'exécution du système vérifié (non intrusive).

2.2.3 Vérification de modèles

Pour déterminer si un système informatique satisfait une propriété, la vérification des modèles [06], ou model-checking en anglais, décide, via un algorithme, si un modèle (une abstraction) du système satisfait la formule logique représentant la propriété souhaitée. Il s'agit d'abord de construire une représentation symbolique ou réelle de l'espace des états possibles du système, ensuite de vérifier si la formule est satisfaite pour chaque état.

Quand le modèle représentant le système à vérifier est fini, l'exploration de son espace d'états peut être exhaustive. Toutefois, ce modèle se retrouve rapidement de taille importante, lorsque le système à vérifier est réaliste. Des techniques d'approximation ont été développées pour réduire cette explosion combinatoire des états. Par exemple, [07] propose d'extraire une représentation réduite du modèle. Cependant, le passage à l'échelle reste problématique.

De plus, la vérification de modèles repose sur une abstraction du programme à vérifier. Même si le modèle du programme est complètement vérifié, aucune garantie n'existe pour assurer que l'implantation dérivée du modèle soit également vérifiée. Il faut donc vérifier l'implantation par rapport au modèle.

2.2.4 Preuve de programmes

La preuve de programmes consiste à vérifier, automatiquement ou de manière assistée, que les formules logiques combinant les spécifications et les exécutions possibles du programme sont des théorèmes. Par exemple, la vérification déductive est fondée sur la logique de Hoare [08] et l'annotation du programme par les pré-conditions (exigences sur les arguments de fonctions), les post-conditions (garanties sur les résultats des fonctions) ainsi que les variants et invariants de boucles. Cette méthode vérifie alors que, pour chaque fonction, les pré-conditions impliquent les post-conditions, en s'appuyant sur les variants et

invariants. De plus, les post-conditions doivent être satisfaites pour chaque appel de fonction. Elle s'appuie sur des générateurs d'obligations de preuve (par exemple Why [09]), puis des outils de démonstration automatique, ou des assistants de preuve. Notons que le programme à analyser doit être complètement accessible (boîte blanche). Parmi les outils utilisant la preuve de programmes, et particulièrement la vérification déductive, nous citons, par exemple, FramaC [10] qui analyse des programmes C. Cependant, cette méthode requiert généralement de nombreuses annotations manuelles, notamment, lorsque le programme est de taille conséquente.

2.2.5 Analyse statique

L'automatisation de l'étude des comportements possibles d'un programme, notamment dans le cadre de l'optimisation lors de la compilation ou de la vérification de propriétés, a fait naître des techniques particulières telles que l'analyse statique. L'analyse statique d'un programme permet de prédire les comportements ou les valeurs possibles à l'exécution, à partir du code source sans exécuter celui-ci, d'où le terme statique par rapport au test qui est de l'analyse dynamique. En effet, l'étude sémantique de la représentation syntaxique d'un programme permet d'inférer des propriétés survenant à l'exécution mais sans aucune exécution du programme.

L'étude des propriétés des programmes informatiques est soumise à deux limitations : d'une part, la sémantique d'un programme n'est en général pas calculable de manière finie et d'autre part, le théorème de Rice montre que :

"Toute propriété non triviale (ni toujours vraie, ni toujours fausse) sur la sémantique d'un langage de programmation est indécidable". L'interprétation abstraite permet dans certains cas de pallier ces limitations [11, 12, 13] grâce à la définition d'une méthode de calcul approchée par sur-approximation de la sémantique.

L'idée fondamentale de l'interprétation abstraite [11] est de s'appuyer sur les résultats des mathématiques discrètes pour décrire et calculer les propriétés de programmes issues de leur sémantique d'exécution. Les mathématiques interviennent d'une part dans les propriétés des structures de treillis complets ou d'ordre partiel, et leur préservation par une fonction monotone ou les connexions de Galois, et, d'autre part, dans les théorèmes de points fixes [14] ainsi que leurs versions constructives par Kleene.

L'analyse statique par interprétation abstraite [15] calcule une abstraction des propriétés du système réel dans un domaine abstrait. La correction de cette technique assure qu'une propriété satisfaite dans le domaine abstrait le sera également dans le système réel.

Plusieurs travaux fondés sur l'analyse statique ont vu le jour dans le domaine des applications critiques. Par exemple [16] appliquent le principe de l'analyse statique par interprétation abstraite sur des programmes C pour des systèmes critiques de niveau A en aéronautique. Cette technique est appliquée directement au programme source pour détecter les erreurs à l'exécution. Ces travaux ont montré la faisabilité de l'utilisation des méthodes formelles dans le processus de développement des systèmes critiques.

Cependant, il reste nécessaire de tenir compte de l'environnement à partir du quelle programme a été généré [17].

2.2.6 Validation de la traduction

Cette technique a été introduite par Pnueli, dans le cadre des projets européensSafeAir I et II sur le développement de systèmes sûrs en aéronautique, et exploitée en s'appuyant sur différentes techniques formelles : la vérification de modèles [18, 19, 20], l'interprétation abstraite [21], ou les algorithmes spécifiques prouvés dans un assistant de preuve [22]. Cette technique consiste à comparer l'origine et le résultat de la traduction en s'appuyant sur des méthodes adaptées à la propriété de correction souhaitée. Cette comparaison peut être syntaxique (par exemple, une classe Java est produite pour chaque classe Uml) ou sémantique (le résultat de l'exécution est le même pour la sémantique du modèle et du code). La validation de la traduction a été développée pour détecter les erreurs des générateurs de code. En pratique, elle complète le compilateur ou le générateur de code par un module qui vérifie les propriétés de correction par une analyse statique. Cette dernière s'assure de la correspondance du programme source et du code généré.

La validation de la traduction a été utilisée dans la vérification de plusieurs systèmes industriels mais a un coût de vérification élevée. Cependant, quand le module de validation échoue, l'utilisateur n'a généralement aucun diagnostic sur son système : l'erreur peut être liée au modèle, au compilateur, au générateur de code ou même à l'outil de vérification. À ceci s'ajoute la nécessité de vérifier le module de validation lui-même. Par contre, le générateur de code lui-même est une boîte noire qui n'est pas contrainte dans son implantation par la technique de vérification.

Dans le cadre de GeneAuto, pour éviter ce problème, nous avons choisi de nous intéresser principalement à la vérification du générateur lui-même et non à des instances de génération.

2.2.7 Code auto-certifié

Le principe du code auto-certifié, ou Proof Carrying Code [23], consiste à produire en plus du code généré la preuve, appelée certificat, que ce code est correct. Celle-ci sera vérifiée indépendamment lors de l'utilisation de ce code.

Cette technique a été appliquée aux systèmes nécessitant une garantie de sécurité, principalement dans le cas de code mobile.

La preuve de correction peut être produite par l'utilisateur ou par le générateur.

Cependant, la preuve fournie peut être incorrecte. Il s'agit ici, comme dans la validation de la traduction, d'une vérification lors de l'utilisation. De plus, il peut être plus coûteux de produire une preuve que de vérifier une propriété donnée. Le problème essentiel reste la possibilité d'un échec de vérification et de l'assistance à l'utilisateur dans ce cas. Dans notre cas, le générateur de code doit être une boîte blanche pour permettre la construction de la preuve conjointement à la génération du code.

2.2.8 Développement prouvé de compilateurs

La vérification est appliquée au compilateur lui-même. Il s'agit de spécifier formellement les exigences ainsi que l'implantation du compilateur et la preuve que celle-ci satisfait ces exigences. Cette technique est ancienne et a donné de nombreux résultats concluants. Nous pouvons citer les travaux initiaux de Milner et Weyhrauch [25] avec LCF et la bibliographie relativement récente réalisée par Dave [03].

Plusieurs travaux sur la vérification de compilateurs sont d'actualité. Nous citons par exemple, les travaux de [25, 26] et de [27] sur la vérification d'un mini-compilateur Java en utilisant l'assistant de preuve Isabelle [28].
De même, [29, 30] ont vérifié formellement un mini-compilateur C, en une seule passe en utilisant Isabelle.
Dans le cadre du développement d'un compilateur réaliste et applicable en industrie, nous citons les travaux récents de Leroy et al. [31].
Il s'agit d'un compilateur d'un sous-ensemble du langage C qui comporte les éléments nécessaires pour le développement d'applications embarquées critiques [32], et qui produit du binaire optimisé. Le compilateur a été développé directement avec l'assistant de preuve Coq. Pour simplifier la vérification du compilateur, celui-ci a été découpé en de nombreuses transformations élémentaires successives, chaque transformation est prouvée correcte en
Coq. Le développement du compilateur C en Coq a permis d'assurer la préservation, dans le code machine, d'une certaine observation de la sémantique d'exécution (les événements d'entrée-sortie) et par conséquence des propriétés préalablement vérifiées sur le code source qui correspondent à cette observation.
Cette approche est très prometteuse dans le domaine de la vérification formelle et particulièrement dans les applications critiques qui nécessitent plus d'assurance. Elle est en cours d'expérimentation au sein d'Airbus.
La plupart des approches de vérification formelle appliquées aux compilateurs s'appuient sur une définition de la sémantique des langages source et cible et sur la preuve de préservation de cette sémantique. Par exemple, Leroyet al. S'intéressent aux événements d'entrée/sortie avec l'environnement du programme et montrent que la cible en assembleur va effectuer les mêmes séquences d'entrée/sortie que le programme source en C.

Partie 2

Chapitre 3

Les automates

Chapitre 3
Les automates

Sommaire

Un algorithme ou un système informatique peut être vu comme et, représenté par, un automate se trouvant dans un état et effectuant des transitions pour passer dans d'autres états.

Dans ce chapitre, nous présentons divers concepts utilisés par la suite en essayant d'harmoniser au mieux diverses notions présentées dans des documents de références différents de manières différentes

1. Définition de base

Dans ce qui suit, nous supposons qu'il existe deux ensembles d'éléments distincts :

1) L'ensemble P des propositions qui seront généralement associées aux états.
2) L'ensemble A des actions qui seront généralement associées aux transitions entre états.

Nous distinguons l'ensemble L des actions visibles de l'action invisible τ ($\notin$L). Intuitivement, L'action τ permet de modéliser un changement d'état du système non marqué par une action visible par l'environnement.

$A =_{déf} L \cup \{\tau\}$.

Une horloge est une variable spéciale dont la valeur augmente automatiquement avec le passage du temps. Nous supposons dans ce qui suit que le domaine d'une horloge est R+.

Nous dénoterons dans ce qui suit

- p désigne une proposition faisant partie de P.
- a, b désignent des actions faisant partie de L.
- H comme étant un ensemble d'horloges.
- x, y comme étant des horloges de H.

2. Séquences et arborescences

Une trace ou séquence d'états (s_0, s_1, s_2,…) est une séquence finie ou infinie d'états labellés par un ensemble de propositions. La **figure 1.a** montre une trace où le premier état est labellé par p, le second par p et q, le troisième de nouveau par p,…

On trouve également des séquences similaires, appelée ici séquence d'actions, dans lesquelles on ne retrouve pas des propositions mais des actions. Une séquence d'actions est une séquence finie ou infinie (s_0, a_0, s_1) (s_1, a_1, s_2) … où s_i est un état et $a_i \in$ A.

La **figure 1.b** montre une séquence d'actions où l'action a est d'abord effectuée, puis les actions b, c,…

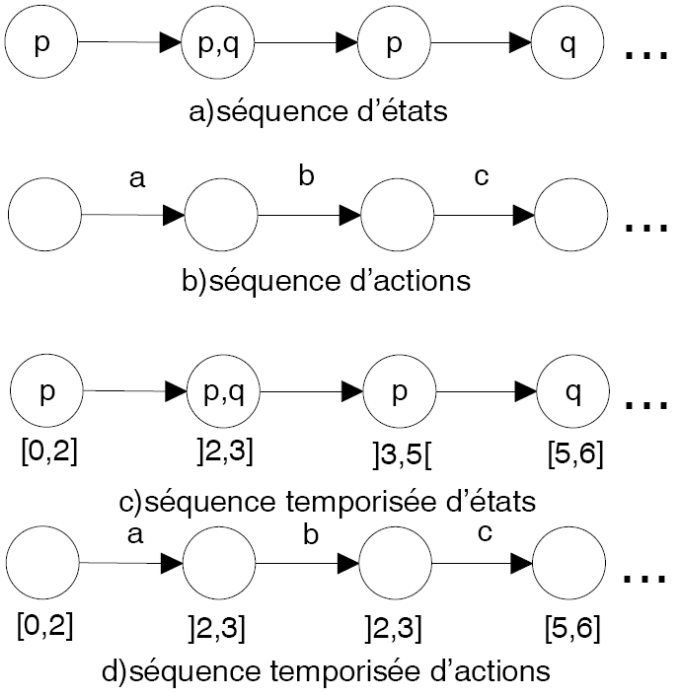


Figure 1

On peut rajouter à ces séquences une notion quantitative de temps durant lequel on reste dans les différents états, en associant à chaque état un intervalle. On obtient ainsi une séquence temporisée d'états et une séquence temporisée d'actions. Dans ce qui suit, nous supposons que le temps est isomorphe à R^+.

Un intervalle est un couple | a, b | où a, b $\in R^+$ et | $\in$ {[,]}. Pour un intervalle I, la fonction Min(I) donne la borne inférieure de I. De même, la fonction Max(I) donne la borne supérieure de l'intervalle I.

L'intervalle I – t est l'intervalle contenant l'ensemble des valeurs t' tel que $\exists t'' \in I$ avec $t' = t'' - t$ et $t' \geq 0$.

Une séquence d'intervalles est une suite finie ou infinie d'intervalles $I = I_0, I_1, I_2,\ldots$ qui partitionne R+, c'est à dire telle que

- pour tout intervalle I_i de I avec i > 0, I_{i-1} et I_i sont adjacents.
- toute valeur t $\in$ R+ appartient à un et un seul intervalle I_i.

Une séquence temporisée d'états est un couple (σ, I) constitué d'une trace σ et d'une séquence d'intervalles I. Le $i^{ème}$ intervalle de la séquence d'intervalle est associé au $i^{ème}$ état de la trace. Notons qu'il y a autant d'intervalles dans I qu'il y a d'états dans σ. La **figure 1.c** montre une séquence temporisée d'états dans laquelle on reste dans l'état initial, labellé par p, durant l'intervalle [0,2]. Ensuite, on passe dans l'état suivant, labellé par p et q, et on y reste durant l'intervalle]2,3],…

De même, une séquence temporisée d'actions est un couple (σ, I) constitué d'une séquence d'actions σ et d'une séquence d'intervalles I. Comme dans le cas d'une séquence temporisée d'états, le $i^{ème}$ intervalle de la séquence d'intervalle est associé au $i^{ème}$ état de la séquence d'actions. La **figure 1.d** montre une telle séquence où on reste dans l'état initial durant l'intervalle [0,2] avant d'effectuer l'action a et de se retrouver dans l'état suivant de la séquence. On reste dans cet état pendant l'intervalle]2,3], puis l'action b est effectuée,…

Certains ensembles de séquences présentées ci-avant peuvent être représentés sous forme d'arborescence.

Un arbre de branchement est un arbre ayant des branches finies ou infinies dont la racine et les nœuds sont labellés par des propositions. Une branche d'un tel arbre est une séquence d'états. La **figure 2.a** montre un tel arbre où partant de la racine, labellée par p, on peut transiter vers un fils gauche ou un fils droit. Si on décide de transiter vers le fils gauche, on se retrouve dans un nœud labellé par p et q à partir duquel on a le choix entre trois fils,…
Un arbre d'actions est un arbre ayant des branches finies ou infinies dont les arcs sont labellés par une action. Une branche d'un tel arbre représente une séquence d'actions. La **figure 2.b** représente un tel arbre où, partant de la racine, on peut au choix effectuer une action a ou une action b. Si on effectue l'action b, on se retrouve dans un nœud où on ne peut que refaire une action b,…

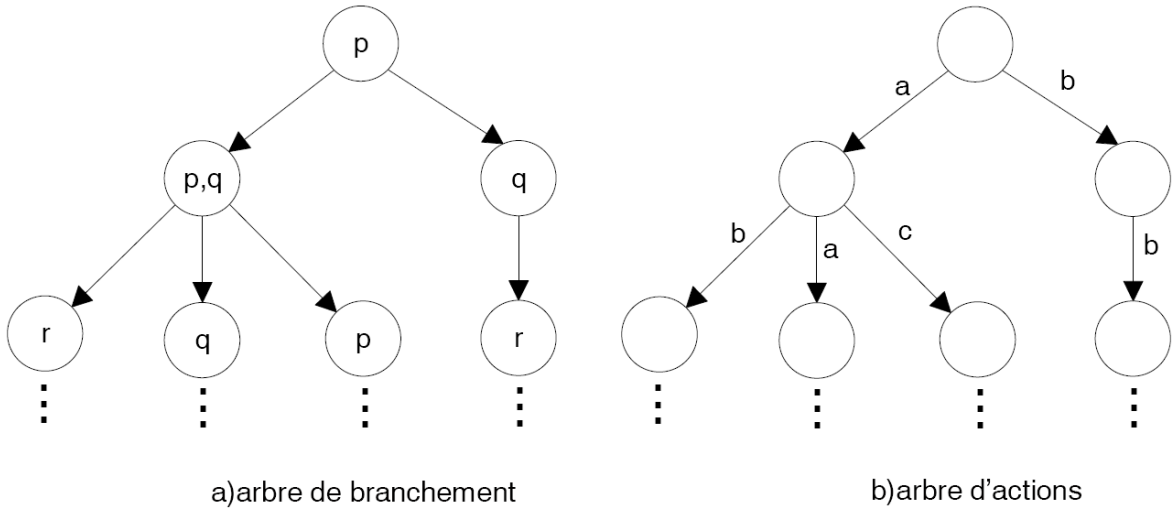


Figure 2

Les arbres temporisés rajoutent par rapport à ces deux types d'arbres une notion de temps quantitatif.
Un arbre temporisé est un arbre fini ou infini dans lequel un intervalle est associé à chaque nœud. Cet intervalle représente les instants pendant lesquels on peut se trouver dans le nœud.

Un endroit dans un arbre de branchement se caractérise donc par un nœud de l'arbre et un instant. Lorsqu'on est dans un nœud d'un arbre temporisé, à chaque instant t inclus dans l'intervalle I associé à ce nœud, il est possible de soit faire une transition vers un nœud fils, la borne inférieure de l'intervalle associé à ce fils devant être t, soit de vieillir jusqu'à un instant t' toujours inclus dans I. On peut également avoir des nœuds sans fils où à chaque instant, on ne peut que vieillir. Chaque parcours dans de tels arbres est soit une séquence temporisée de branchement, soit une séquence temporisée d'actions suivant que ce soient les nœuds de l'arbre qui sont labellés par des ensembles de propositions ou les transitions par des actions.

Notons qu'il existe également des séquences et des arbres, temporisé ou non, où on retrouve simultanément des propositions et des actions.

3. Les automates

Ayant défini A et P,
Un automate A peut être défini comme un 3-uple (S, δ, S_0) où

- S est l'ensemble des états.
- δ est la fonction de transition entre états.
- S_0 est l'ensemble des états initiaux (certaines définitions réduisent cet ensemble à un seul état).

Un automate peut définir un langage L(A) ou encore l'ensemble des comportements possibles d'un système. Suivant ce que l'automate modélise, on peut définir δ de diverses façons :

- δ : S x S : définit les changements d'états possibles.
- δ : S x A → S : définit un automate déterministe qui change d'état en avalant un symbole de A.
- δ: S x A → 2^S : détermine un automate non déterministe qui change d'état en avalant un symbole de A.

Ayant δ : S x A → S ou S x A → 2^S, on parle de système à transitions labellé (LTS pour labelled transition system).

De même, on peut rajouter divers éléments :

- μ : S → 2^P : une fonction qui associe à chaque état un ensemble de propositions.
- F ⊆ S : un ensemble d'états d'acceptation. Cet ensemble sert à définir le langage L(A) associé à l'automate.

Ainsi, pour définir un langage de mots finis, on peut définir un LTS A avec L(A) défini classiquement comme donné par [33] c'est à dire l'ensemble des strings avalés par A et aboutissant à un état de F.

Pour définir un langage de mots infinis (ω-mots), on peut définir L(A) comme l'ensemble des strings avalés en empruntant un chemin dans A partant d'un état de S_0 et traversant infiniment souvent un sommet de F.
Avec cette définition nous parlons d'automate de Büchi (et d'acceptation de Büchi)
Ainsi l'automate de Büchi représenté à la **figure 3** définit le langage $(a + b)^*ac^{\omega}$.

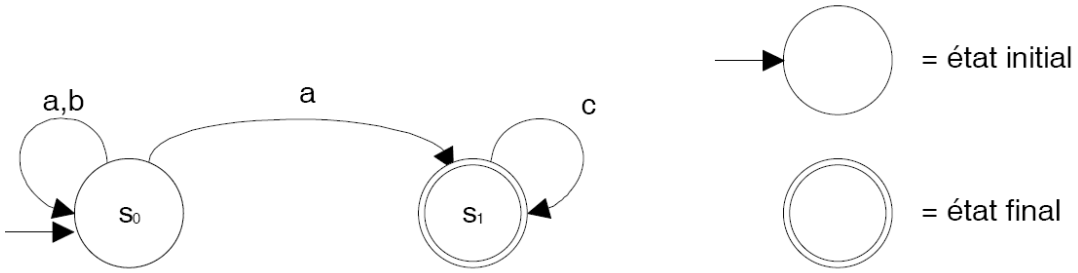


Figure 3:exemple d'automate de Büchi

Si le but est de décrire et d'analyser un système, on peut définir une structure de Kripke comme un 4-uple A = (S, δ, s_0, μ) avec
S un ensemble d'états finis
δ⊆ S x S
μ : S → 2P
s_0 est l'état initial.
Partant de s_0, seuls les chemins s_0, s_1, s_2,… infinis sont acceptés dans une structure de Kripke. Ceci peut être plus formellement spécifié en rajoutant aux structures de Kripke un ensemble d'états d'acceptation F et en introduisant l'acceptation de Büchi, avec F contenant toujours tous les états de S.

Un exemple de structure de Kripke est représenté par la **figure 4**, où les propositions associées à chaque état s sont notées à coté du nœud correspondant. Ces propositions peuvent représenter, par exemple, les propriétés des états de la structure de Kripke.

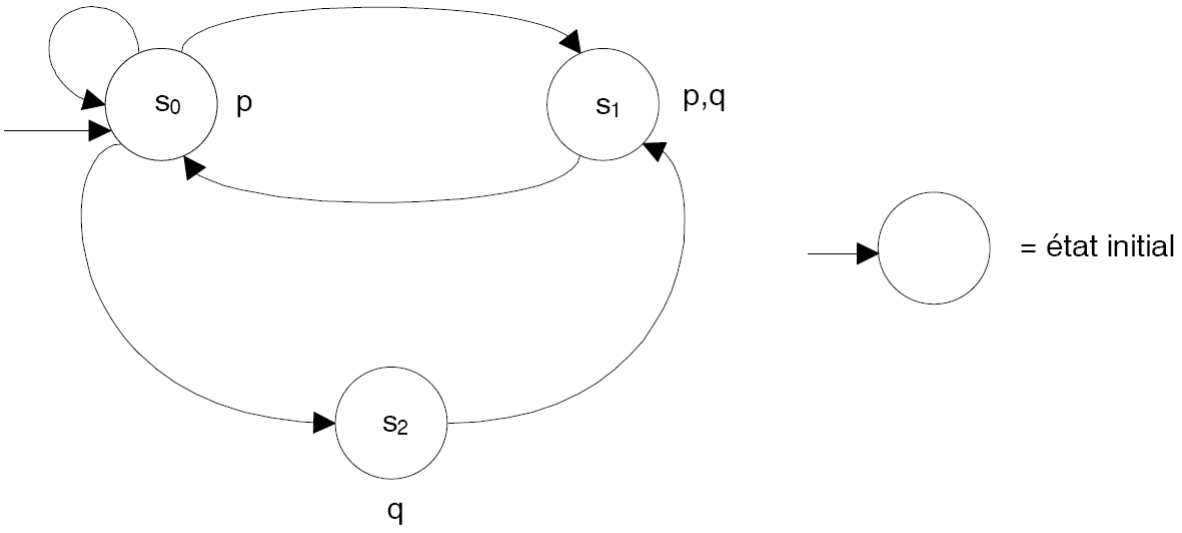


Figure 4:exemple de structure de Kripke

Un automate temporisé est un automate étendu en ajoutant des horloges et des contraintes sur ces dernières. Ces horloges permettent d'introduire une notion quantitative de temps. Cela permet de représenter des systèmes temporisés sous forme d'automate.
Nous allons présenter ici deux types d'automates temporisés, utilisés dans les chapitres suivant. Le premier, que nous appellerons structure temporisée, a été défini dans [35]. Le second, légèrement différent, que nous appellerons graphe temporisé, a été défini dans [34].

Pour ces deux automates temporisés, on définit une valuation v comme étant une fonction qui associe une valeur à chaque horloge d'un automate temporisé. v(x) donne la valeur donnée à x par v.

La valuation v + t est la valuation v' tel que pour tout x, v'(x) = v(x) + t.
La valuation v [x → 0], est la valuation v' telle que v'(x) = 0 et pour tout y ≠ x, v'(y) = v(y).
On peut généraliser cet opérateur pour un ensemble d'horloges H'. Dans ce cas, la valuation v [H' → 0] est la valuation v' telle que pour tout x ∈ H', v'(x) = 0 et pour tout x ∉ H', v'(x) = v(x).

Pour une structure ou un graphe temporisé, un état est un couple <s, v> où s est un sommet de l'automate et v est une valuation. Un état caractérise donc un "endroit" du système temporisé à un instant donné.
Si <s, v> est un état, on définit l'état <s, v> + t ≡ <s, v + t>

Structure temporisée [35]

Dans les structures temporisées, l'ensemble des contraintes possibles sur les horloges de H est défini par la grammaire:

$$\varphi \rightarrow x \leq c \mid c \leq x \mid \neg\varphi \mid \varphi \wedge \varphi \text{ où } c \in \angle.$$

Certaines contraintes telles que x = c, x < c, true, φ ∨ φ peuvent être définie comme des abréviations. (Elles sont équivalentes à des contraintes permises par la grammaire)

Une structure temporisée est un automate ayant des propriétés temporelles ; elle est définie par un 6-uples (S, s_{init}, μ, H, Δ, δ), où
- S est l'ensemble fini des sommets de la structure.
- $s_{init} \in$ S est l'état initial.
- μ est une fonction qui associe à chaque sommet de S un ensemble de propositions ⊆ P.
- H est l'ensemble des horloges utilisées par la structure.
- Δ associe à chaque sommet de S une contrainte sur les horloges, appelée invariant du sommet.
- $\delta \subseteq S \times S \times 2^H$ est un ensemble de transitions. Une transition entre états s et s' ∈ S est labellée par un ensemble d'horloges réinitialisées.

L'ensemble des états d'une structure temporisée est l'ensemble des états <s, v> où les valeurs données aux horloges par v satisfait Δ(s). L'état initial est l'état <s_{init}, v_0> où v_0 est la valuation donnant la valeur 0 à chaque horloge de H (on suppose que v_0 satisfait Δ(s_0)). A chaque instant t, la structure temporisée se trouve dans un état <s, v>. Toutes les horloges vieillissent à la même vitesse. Partant de <s, v>, si le système vieillit de t, les valeurs des

horloges seront données par v + t après vieillissement. Si on reste dans un sommet s pendant l'intervalle de temps (t, t'), les valeurs des horloges doivent satisfaire l'invariant de s durant tout cet intervalle. Si en t' la structure temporisée effectue une transition vers un sommet s', il faut que les valeurs des horloges satisfassent l'invariant de s'. L'état dans lequel on se trouve en t' dépend de l'intervalle (t, t') durant lequel on est resté en s. Si cet intervalle est ouvert à droite, alors on se trouve déjà en s', sinon on se trouve encore en s. Lors d'une transition, les horloges associées à cette transition sont réinitialisées à zéro. La manière dont on initialise les horloges dépend également du fait que l'intervalle est ouvert ou fermé à droite. Si l'intervalle est ouvert à droite, le changement de sommet est d'abord effectué, puis l'initialisation des horloges. Par contre, l'initialisation des horloges est faite avant le changement de sommet dans le cas où l'intervalle est fermé à droite. Il est à noter que, dans ce cas là, les horloges remises à zéro ne valent plus zéro lorsqu'on arrive dans l'état d'arrivée.
Si on a l'exécution (s, [0, 3])(s',]3, 4[)(s'', [4, 7])…
A l'instant 3, on se trouve toujours dans le sommet s et toutes les horloges ont la valeur 3.Comme l'intervalle [0, 3] est fermé à droite, on effectue d'abord la mise à zéro des horloges puis le changement de sommet. Au moment où on arrive en s', la valeur des horloges initialisées lors de la transition n'est plus zéro, mais juste plus grande. Par contre, lors de la transition à l'instant 4, le changement de sommet est effectué avant la remise à zéro des horloges. Lorsqu'on arrive dans s'', ces horloges valent bien dans ce cas zéro.
Nous voyons que dans ce modèle, une transition ($\in \delta$) d'une structure temporisée implique non seulement un changement d'états mais également un vieillissement. On ne peut donc pas au même moment se trouver dans deux états différents.

Dans un système réel, des événements survenant dans ce système provoquent des changements d'états du système. Ces événements ne sont en général pas instantanés et prennent un temps t. Bien que les transitions ne sont pas instantanées, ce temps t pris lors des changements d'états n'est pas directement spécifiable au niveau des transitions d'une structure temporisée, mais peut être modélisé, pour un événement a, par une transition représentant le début de a, une transition représentant la fin de a et un état représentant les instants pendant lesquels a est effectué.

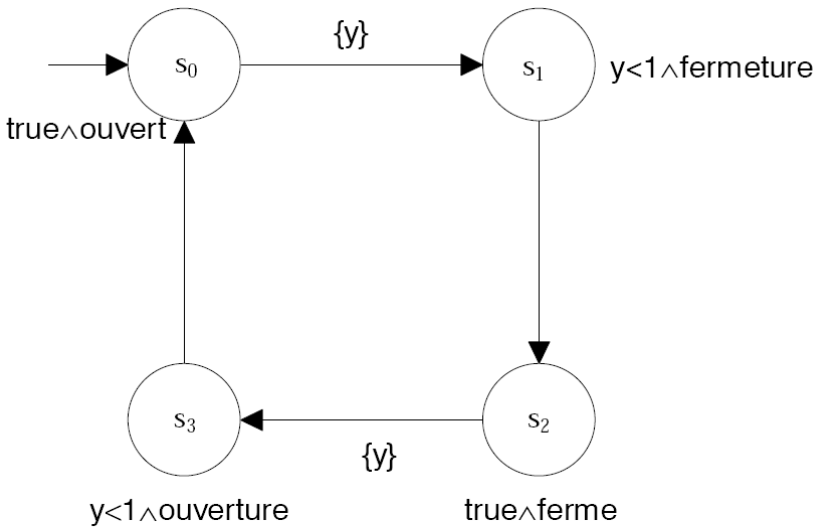


Figure 5: structure temporisée représentant le comportement d'une barrière d'un passage à niveau

La **figure 5** montre une structure temporisée représentant le comportement d'une barrière d'un passage à niveau. Le sommet s_0 représente l'état où la barrière est ouverte et aucun train n'est annoncé. La structure temporisée permet de rester dans ce sommet indéfiniment, ceci correspondant au cas où il n'y aurait jamais de train passant par le passage à niveau.

Lorsqu'un train est annoncé, la barrière commence à se fermer. Ceci correspond à effectuer une transition vers le sommet s_1. Comme la barrière met au plus une unité de temps pour être effectivement fermée, on se retrouve ensuite dans le sommet s_2, représentant l'état où la barrière est fermée, au plus une unité de temps plus tard. Pour une raison ou une autre, un train rentrant dans le passage à niveau peut ne jamais en ressortir (s'il tombe en panne par exemple). Il est donc possible de rester dans cet état indéfiniment et la barrière peut ne jamais se relever. Ceci est représenté dans la structure temporisée par le fait qu'il est possible de rester indéfiniment dans le sommet s_2.Lorsque le train sort du passage à niveau, la barrière commence à se relever. Ceci correspond à effectuer la transition vers le sommet s_3. De nouveau, la barrière mettra au plus une unité de temps pour être rouverte. Après être rentré dans le sommet s_3, on se retrouvera donc de nouveau dans le sommet s_0 après au plus une unité de temps.

Graphe temporisé [34]

Pour un graphe temporisé, Ψ(H) est l'ensemble des contraintes possibles sur les horloges de H, une contrainte étant définie par la grammaire :

$$\varphi \rightarrow \text{true} \mid x < c \mid x \leq c' \mid x - y < c'' \mid x - y \leq c'' \mid \neg\varphi \mid \varphi \wedge \varphi$$

Où $c \in \angle \setminus \{0\}$.
$c' \in \angle$.
$c'' \in 9$.

De nouveau, certaines contraintes comme $x = c$, $x > c$, true ou $\varphi \vee \varphi$ peuvent être définie comme des abréviations.

> La formule $x - y \; op \; c''$ ($op \in \{<, \leq\}$) permet d'avoir ici des contraintes sur les horloges légèrement différentes par rapport aux contraintes permises dans le cadre des structures temporisées.

Notons que par rapport à [34], on interdit des contraintes qui n'ont pas de sens, comme $x \leq -5$ ou $x < 0$, en limitant le domaine des constantes c dans les formules $x \; op \; c$.
Un graphe temporisé est un 5-uple $(S, s_{init}, H, \Delta, \delta)$ où
- S est l'ensemble fini des sommets du graphe.
- $s_{init} \in S$ est le sommet initial.
- H est l'ensemble de ses horloges.
- $\Delta : S \rightarrow \Psi(H)$ est une fonction qui associe à chaque sommet une condition appelée invariant du sommet. Contrairement aux structures temporisées, un invariant doit satisfaire la propriété de fermeture en arrière, c'est à dire $\forall s \in S, v \in V, t \in R^+$. $v + t$ satisfait $\Delta(s) \Rightarrow v$ satisfait $\Delta(s)$. Cela veut dire que si on peut se trouver dans un état s à un moment donné (caractérisé par une valuation) alors on pouvait s'y trouver également avant.
- $\delta \subseteq S \times A \times \psi(H) \times 2^H \times S$ est un ensemble d'arcs labellés par une action, une contrainte sur les horloges et un ensemble d'horloges.

Les graphes temporisés sont similaires aux structures temporisées, à quelques détails près. Une différence entre ces deux types d'automates, outre les invariants de sommets possibles, est que les transitions d'un graphe temporisé sont labellées par une action et une contrainte sur les horloges. Pour pouvoir effectuer une transition, les valeurs des horloges doivent ici, en plus de satisfaire l'invariant du sommet d'arrivée, satisfaire la condition de la transition. De plus, effectuer une transition dans un graphe temporisé est instantané, c'est à dire que si une transition est effectuée à un instant t, on se trouvera toujours à l'instant t après la transition. Lorsqu'on arrive dans l'état d'arrivée, les horloges devant être remises à zéro valent effectivement zéro. Contrairement à une structure temporisée, on peut donc se trouver ici dans plusieurs états différents à un instant donné.
La figure 6 représente le graphe temporisé d'un contrôleur de température. Celui-ci est relié à un capteur qui envoie au contrôleur un signal lorsque la température dépasse un certain seuil. A ce moment-là, le contrôleur active un système de refroidissement qui sera actif pendant deux unités de temps. Pendant cette période aucun signal ne peut venir du capteur. Si cela se produit tout de même, le système rentre dans un état d'erreur. Après les deux unités de temps, le contrôleur retombe dans son état initial et attend un nouveau signal. De plus, on considère que le capteur n'est pas fiable et peut tomber en panne, un processus de

refroidissement sera pour cela réenclenché automatiquement lorsqu'un signal s n'a plus été reçu depuis dix unités de temps.

Le graphe temporisé de la **figure 6** débute dans le sommet initial avec x et y valant zéro. L'envoi du signal par le capteur fait basculer le graphe temporisé dans le sommet s_1. Comme la garde de ce sommet est y = 0 et que y est remis à zéro lors de la transition, le processus de refroidissement est immédiatement enclenché et le graphe temporisé transite instantanément vers l'état s_2 où il reste pendant la durée du processus, c'est à dire deux unités de temps. Si un signal apparaît pendant ce temps, le graphe temporisé bascule vers le sommet s_3, pour spécifier une erreur. Après les deux secondes, le système revient au sommet initial s_0 et attend le prochain signal. La variable y permet de déterminer le temps écoulé depuis le dernier signal reçu. Lorsqu'y vaut 10, le système commence automatiquement un refroidissement.

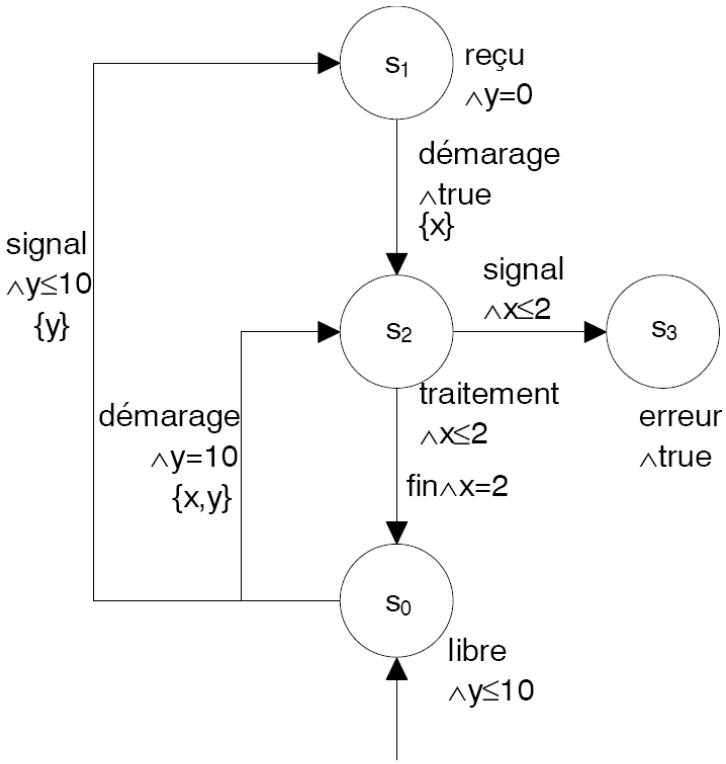
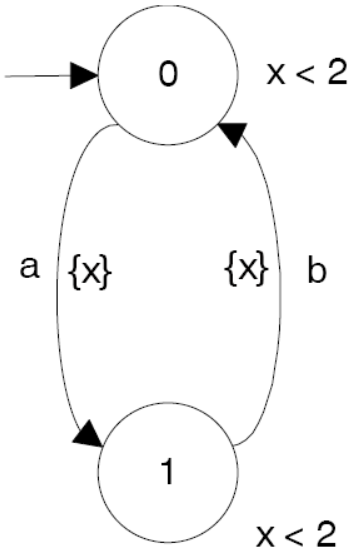


Figure 6:graphe temporisé représentant un contrôleur de température

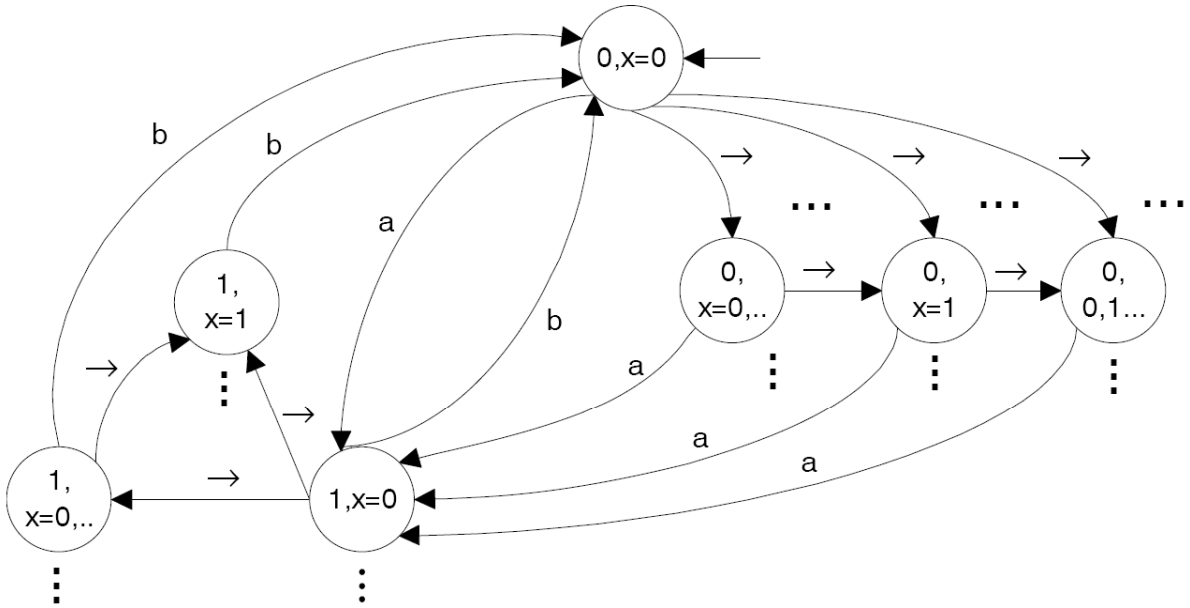
A partir d'un graphe temporisé G, nous pouvons construire un LTS, généralement infini vu la densité du temps, donnant les transitions et changements d'états possibles de G.
Nous donnons ici les idées principales de la construction d'un tel LTS pour un graphe temporisé (voir [34] pour la définition complète).
L'ensemble des états du LTS est l'ensemble des états <s, v> de G. Il existe deux types de transitions :

1) Les transitions effectuant une action a. Une telle transition existe entre deux états <s, v> et<s', v'> s'il existe un arc (s, a, φ, H', s') dans G tel que v satisfasse φ et v' = v [H' → 0].
2) Le vieillissement de toutes les horloges d'une même valeur. La nouvelle valuation doit toujours satisfaire l'invariant de s. Si le graphe temporisé peut se trouver dans un état s avec des valeurs d'horloges données par la valuation v et vieillir de t, nous aurons dans le LTS des transitions de l'état <s, v> vers les états <s, v + t'> avec t' ≤ t. Ceci étant justifié par la propriété de fermeture arrière des invariants de sommets. Partant d'un sommet s avec des valeurs d'horloges données par v, lors d'une transition dans une structure temporisée, vu la densité du temps et le fait qu'une transition n'est pas instantanée, on ne pourra pas déterminer les valeurs d'horloges après transition. Il n'est pour cela pas possible, pour une structure temporisée, de déterminer les transitions du premier type et donc de construire un LTS à partir de tels automates temporisés.

La **figure 7.b** montre une partie du LTS du graphe temporisé représenté à la **figure 7.a**. Les transitions labellée par → sont des transitions liées à un vieillissement.



a)graphe temporisé



b)LTS du graphe temporisé

Figrue 7

4. Traduction d'automates

L'analyse d'un système modélisé sous la forme d'un automate, suivant le type de propriétés que nous voulons regarder, peut se traduire par l'analyse de chaque exécution de l'automate de manière indépendante par rapport aux autres ou par l'analyse séparée des ensembles des exécutions possibles partant d'un état commun de l'automate, ces ensembles étant dans ce cas mis sous forme d'arbre.
Cela peut être fait, suivant les cas, en utilisant des séquences, des arbres, d'états ou d'action.

Pour un automate non temporisés :

- l'ensemble des traces ou des séquences d'actions de l'automate, suivant les cas, est l'ensemble des séquences d'états ou de transitions possibles décrit par l'automate.
  Chacune de ces traces ou séquences d'actions satisfait les conditions suivantes :
  - Elles commencent par un état initial de l'automate.
  - Dans le cas de traces, si un état s_1 précède un état s_2 dans une trace, alors il existe une transition de s_1 vers s_2 dans l'automate. Dans le cas de séquences d'actions, si une séquence contient la transition (s_1, a, s_2) alors il existe une transition de s_1 vers $s_2$labellée par l'action a dans l'automate.

- Les traces ou séquences d'actions satisfont les conditions d'acceptation de l'automate.

- L'ensemble des arbres de branchements ou d'actions de l'automate est un ensemble d'arbres dont les racines sont des états initiaux de l'automate et reprenant pour chaque états accédé, les transitions possibles et les états voisins accédés via ces transitions. De tels arbres sont donc une sorte de dépliage de l'automate.

Si, pour être représentées par un automate, les séquences d'états ou d'actions (éventuellement mises sous forme d'arbres) doivent être infinies et que ce n'est pas le cas, on s'arrange pour que ce le soit. Certains types d'automates ne permettant pas d'accepter des séquences finies, il faut alors construire si nécessaire de tels automates de telle manière que certaines de leurs séquences infinies puissent être associées de manière naturelle à une séquence finie. Par exemple, en construisant l'automate comme si les séquences finies étaient acceptées, puis en rajoutant un état " final" vers lequel on transite à la fin d'une séquence finie en effectuant si besoin est une action "finale". Lorsqu'on arrive dans cet état, on ne peut plus que faire des transitions vers l'état "final" lui-même en effectuant de nouveau si besoin est l'action "finale".

De cette manière, des séquences infinies de l'automate auraient comme préfixe les séquences finies désirées, suivie, suivant les cas, soit d'une infinité d'état " final", soit d'une infinité d'action "finale".

Dans le cas d'un automate temporisé, les exécutions peuvent être représentées par un ensemble de séquences ou d'arbres temporisés :

- L'ensemble des séquences temporisées de l'automate est l'ensemble contenant, suivant les cas, toutes les séquences temporisées de branchement ou d'actions satisfaisant les conditions suivantes :
  - Les séquences commencent dans un état initial de l'automate.
  - Pour tout sommet s_i et intervalle I_i d'une séquence temporisée, il faut pouvoir atteindre ce sommet dans l'automate temporisé et y rester pendant l'intervalle I_i en passant au paravent par les sommets s_0, s_1,…,s_{i-1} le précédant dans la séquence temporisée et en y restant durant les intervalles qui leur sont associés dans la séquence temporisée. De plus, dans le cadre d'une séquence temporisée d'actions, toutes les actions effectuées pour atteindre s_i dans l'automate doivent correspondre aux actions effectuées dans la séquence temporisée d'actions pour atteindre s_i.

- L'ensemble des arbres temporisés de l'automate est un ensemble d'arbres dont les racines sont des états initiaux de l'automate temporisé et reprenant pour chaque sommet s accédé à un instant t les transitions possibles et les états accédés via ces transitions ainsi que l'intervalle de temps durant lequel on peut rester dans s. On peut avoir deux types de transitions à partir de chaque nœud
  - Une transition représentant un changement de sommet dans le modèle. Pour une transition donnée de l'automate, les instants pendant lesquels on peut rester dans le sommet d'arrivée vont dépendre de l'instant où cette transition est effectuée. On va donc avoir en général dans l'arbre une infinité de transitions

pour une transition dans l'automate temporisé, une pour chaque instant où celle ci peut être effectuée.

- Une transition liée au fait qu'une transition de l'automate devienne ou cesse d'être disponible dans l'automate. Ceci correspond à un vieillissement jusqu'à un moment où les horloges de l'automate prennent des valeurs telles que les contraintes liées à la possibilité de faire une transition change de valeur. Pour un nœud s_i, la borne de droite de l'intervalle I_i sera le plus petit instant entre l'instant où l'invariant du sommet s_i dans l'automate devient faux et le premier instant où une transition devient disponible ou cesse de l'être. Il n'y a donc pas dans ce cas de changement de sommet.

Partie 2

Chapitre 4

Les logiques temporelles

Chapitre 4
Les logiques temporelles

Sommaire

On présente dans ce chapitre quelques logiques temporelles classiques qui ont été abordées dans le cadre de ce travail. Pour chaque logique, des explications sur les différents opérateurs sont données ainsi que des commentaires sur l'expressivité. Ces commentaires sont basés sur une réflexion personnelle.

1. Présentation des logiques temporelles

Les logiques temporelles que nous allons étudier peuvent être vues comme une extension des logiques propositionnelles du premier ordre. En effet, une logique propositionnelle s'applique à un ensemble de propositions dont les valeurs de vérité sont données par une interprétation (fonction qui va de l'ensemble des propositions possibles dans {Vrai, Faux}). Une formule de logique temporelle est généralement évaluée soit sur une trace (ce sont les logiques temporelles linéaires ou linear time temporal logic), soit sur un arbre de branchement (ce sont les logiques de branchement ou branching time temporal logic), un état donnant la valeur de vérité des propositions à un moment donné. La différence entre les traces et les arbres de branchement est que dans le premier cas, la formule est évaluée sur une des "exécutions" possibles du système tandis que dans le second cas, la formule peut raisonner sur toutes les évolutions futures possibles. Il existe également certaines logiques hybrides dans lesquelles se mélangent les aspects linéaires et de branchement. Nous ne parlerons pas des logiques de ce genre dans ce travail.
Il existe d'autres différences entre les logiques.

Certaines logiques temporelles, appelées logiques temporisées, introduisent une notion quantitative de temps dans leur sémantique. Par contre, dans les logiques non temporisées, bien que les valeurs de vérités des propositions varient en passant d'un état au suivant dans une trace ou un arbre de branchement, aucune notion quantitative de temps dans la sémantique n'est généralement présente. Cet ajout de temps quantitatif est intéressant pour effectuer de la vérification sur des systèmes temporisés.

La vérification de propriétés avec les logiques temporelles temporisées peut se faire en utilisant des automates temporisés. On peut considérer que le temps est discret (ex. : les valeurs du temps sont isomorphes à $\angle$) ou dense (ex. : les valeurs du temps sont isomorphes à 3 ou Θ). Ceci influence la logique considérée. Par exemple, avec un temps dense la notion de moment suivant n'a plus de sens et des opérateurs tels que O (état suivant) peut ne plus être relevant.

Une autre différence qui peut survenir entre les logiques temporelles est l' "orientation". Il y ales logiques orientées propositions : ce sont les logiques évaluées sur des arbres ou des séquences d'états labellés par des propositions. Il y a également les logiques orientées actions où ce ne sont pas les états qui sont labellés, mais les transitions avec des actions. Ceci permet d'analyser un système en regardant les séquences d'actions possibles et non les séquences d'états possibles du système et leurs propriétés associées. Il existe des logiques permettant de combiner des propriétés sur des états et sur des actions. De nouveau, nous ne parlerons pas des logiques de ce genre dans ce travail.

Dans le cadre de ce travail, les logiques temporelles sont utilisées pour spécifier des propriétés à vérifier sur un système. Les formules de logiques temporelles ayant une sémantique bien définie, il est possible de trouver des méthodes systématiques pour les vérifier sur un automate représentant un système. De l'expressivité de la logique utilisée va dépendre l'ensemble des propriétés vérifiables à l'aide de ces méthodes systématiques

définies. Dans le cadre du Model Checking, il est intéressant de pouvoir exprimer à l'aide de logiques temporelles certaines classes de propriétés. Ces propriétés sont les propriétés d'atteignabilité (reachability properties) et d'équité (fairness properties).

Les propriétés d'atteignabilité portent sur l'atteignabilité potentielle ou inévitable d'états ou d'actions. On peut aussi distinguer les propriétés de vivacité (liveness properties) et de sûreté (safety properties). Les premières expriment le fait qu'un état désirable sera atteint ou qu'une action désirable sera effectuée. Par exemple, le fait qu'une demande d'accès à une ressource partagée est toujours suivi de l'accès à cette ressource en un temps fini. Les propriétés de sûreté expriment le fait qu'un état indésirable ne sera jamais atteint ou qu'une action indésirable ne sera jamais effectuée. Par exemple, le fait qu'on ne se trouvera jamais dans une situation deadlock.

Les propriétés d'équité portent sur la répétition infinie de certains états ou actions. On distingue l'équité faible de l'équité forte. L'équité faible exprime le fait que si une propriété est continuellement vérifiée, alors une autre le sera infiniment souvent. Continuellement vérifiée ne veut pas dire que la propriété est invariante mais qu'à partir d'un moment donné, la propriété est toujours satisfaite (avant ce moment, certains états peuvent ne pas satisfaire la propriété). Il peut bien sûr ne pas y avoir de moments précédents. Par exemple, si un processus est continuellement prêt à être exécuté, alors il le sera infiniment souvent. Plusieurs actions ne pouvant être effectuées à la fois, demander qu'une action a continuellement satisfaite implique qu'une autre action b le soit infiniment souvent n'a pas de sens. On ne parle donc pas d'équité faible exprimée en termes d'actions.

L'équité forte exprime quant à elle le fait que si une propriété est infiniment souvent vérifiée ou une action infiniment souvent effectuée, alors une autre propriété ou action le sera également. Par exemple, si un processus demande l'accès à une ressource infiniment souvent, il l'obtiendra infiniment souvent.

2. Exemples de logiques

Nous avons mis en évidence trois caractéristiques majeures que peuvent avoir les différentes logiques temporelles :

- Les logiques peuvent être dites linéaires ou de branchement.
- Les logiques peuvent être temporisées ou non.
- Les logiques peuvent être orientées actions ou propositions.

Théoriquement, ces trois caractéristiques peuvent se mélanger de n'importe quelle manière, néanmoins des logiques de certains types semblent ne pas avoir été définies.

Linéaire/de branchement	**Temporisée /non Temporisée**	**Action/Proposition**	**Exemple**
linéaire	Non Temporisée	Action	TLA
linéaire	Non Temporisée	Proposition	LTL
linéaire	Temporisée	Action	
linéaire	Temporisée	Proposition	MITL
branchement	Non Temporisée	Action	ACTL
branchement	Non Temporisée	Proposition	CTL
branchement	Temporisée	Action	
branchement	Temporisée	Proposition	TCTL

Nous allons présenter ici des exemples pour les différents types de logiques. Cette liste d'exemples étant loin d'être exhaustive, le lecteur pourra se référer par exemple à [37] pour la présentation du µ-calcul et d'autres logiques.

Pour tout nœud, état ou sommet s d'un arbre ou d'une séquence, µ(s) donne l'ensemble des propositions labellant s dans ce qui suit.

2.1 La logique LTL ([36])

2.1.1 Description :

La logique LTL est un exemple de logique temporelle linéaire non temporisée (untimed linear temporal logic).
En voici la syntaxe:
Une formule logique ϕ est une formule LTL si elle peut être construite à partir de la grammaire suivante:

$\phi \rightarrow p \mid \neg p \mid TRUE \mid FALSE \mid \phi \wedge \phi \mid \phi \vee \phi \mid O\phi \mid \phi \, U \, \phi \mid \phi \succ \phi$

Nous introduisons maintenant la sémantique des opérateurs de LTL:
Une formule LTL est évaluée sur une séquence d'états infinie $\sigma = s_0 s_1 s_2 \ldots$

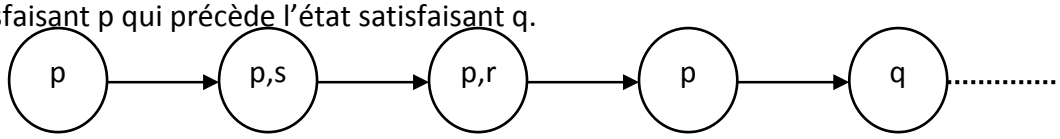
Pour tout σ nous avons $\sigma \models TRUE$ et $\sigma \not\models FALSE$.
$\sigma \models p$ ssi $p \in \mu(\sigma_0)$.
$\sigma \models \neg p$ ssi $\sigma \not\models p$.
$\sigma \models \phi_1 \wedge \phi_2$ ssi $\sigma \models \phi_1$ et $\sigma \models \phi_2$.
$\sigma \models \phi_1 \vee \phi_2$ ssi $\sigma \models \sigma_1$ ou $\sigma \models \phi_2$.
$\sigma \models O\phi$ ssi $\sigma_1 \models \phi$ avec $\sigma_1 = s_1 s_2 s_3 \ldots$
$\sigma \models \phi_1 U \phi_2$ ssi $s_j \models \phi_2$ pour un $j \geq 0$ et $s_k \models \phi_1$ pour tout k avec $0 \leq k < j$.
$\sigma \models \phi_1 \succ \phi_2$ ssi pour tout $i \geq 0$ tel que $s_i \not\models \phi_2$, il existe $0 \leq j < i$ tel que $s_j \models \sigma_1$

Notons que l'opérateur ≻ présenté ici et repris par Wolper dans [39] ne correspond pas à l'opérateur ≻ défini par Manna et Pnueli [37].
La négation d'une formule n'existe pas telle quelle en LTL, mais peut être définie pour chaque type de formule. Ainsi

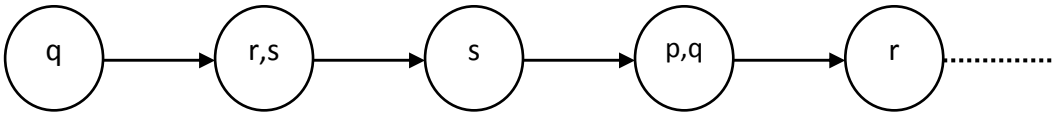
$\neg (\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$
$\neg (\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$
$\neg (O\phi_1) \equiv O\neg\phi_1$
$\neg (\phi_1 U \phi_2) \equiv \neg\phi_1 \succ \neg\phi_2$
$\neg (\phi_1 \succ \phi_2) \equiv \neg\phi_1 U \neg\phi_2$

La formule $\phi_1 U \phi 2$ demande qu'un état de la trace satisfasse la propriété ϕ_2 et que tous les états le précédant dans la trace satisfassent la propriété ϕ_1. Par exemple, la figure 8.a, où chaque état contient les propositions qu'il vérifie, représente une séquence satisfaisant la formule p U q. La figure 8.b satisfait également cette formule. Il n'est, en effet,

pas demandé dans la sémantique de LTL qu'il y ait obligatoirement au moins un état satisfaisant p qui précède l'état satisfaisant q.



a)Une séquence d'états où p est satisfait jusqu'à ce que q le soit



b) q est directement satisfait

Figure 8.séquence satisfaisant pUq

Une formule du type $\phi_1 \succ \phi_2$ demande que dans une trace tous les états, et plus particulièrement le premier, ne satisfaisant pas ϕ_2 soient précédés d'un état satisfaisant ϕ_1.
Une séquence d'états satisfaisant $\phi_1 \succ \phi_2$ ne doit donc pas nécessairement contenir d'états satisfaisant ϕ_1, si tous les états de la séquence satisfont ϕ_2.
L'opérateur O permet de spécifier une propriété devant être satisfaite par l'état successeur d'un état donné. La figure 9 montre une séquence d'états satisfaisant true*U* (p ∧O (¬q)).
Cette formule demande qu'il y ait dans la trace un état satisfaisant p dont le successeur ne satisfait pas q.

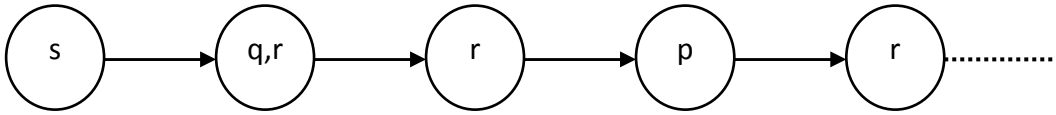


Figure 9. Séquence d'état satisfaisant true*U* (p ∧ O(¬q))

2.1.2 Expressivité :

LTL étant une logique linéaire, il n'est pas possible de différencier l'atteignabilité potentielle de l'atteignabilité inévitable, c'est à dire qu'un état ayant un seul état suivant dans une trace infinie, cela revient au même de demander s'il est possible ou inévitable d'atteindre dans une trace un état ayant une propriété donnée. Dans le cadre de vérifications de certaines propriétés simples, cela ne pose pas de problème. Pour un automate donné, savoir s'il est possible d'atteindre un état satisfaisant une certaine propriété p ne peut être exprimé directement en LTL. Mais on peut vérifier si ¬p est invariant, et de cette évaluation peut être déduite la valeur de vérité de la propriété désirée. Par contre, la propriété selon laquelle il est possible d'atteindre un état dans l'automate à partir duquel tout les états accessibles satisfont p ne peut être exprimée en LTL et on ne peut trouver de subterfuge pour déterminer la valeur de vérité de cette propriété.
Les propriétés de sûreté s'expriment sans problème. Exprimer le fait qu'on n'atteigne jamais un état ne satisfaisant pas p s'exprime sous la forme d'invariance avec la formule

¬(true U (¬p)). Les propriétés d'atteignabilité bornée, c'est à dire le fait qu'on puisse atteindre un état avant c unités de temps, ne peuvent en principe pas être exprimées en LTL puisque cette logique n'est pas temporisée. Grâce à l'opérateur O, on peut néanmoins simuler un passage de temps discret. Si on se donne comme convention qu'on se trouve dans un état différent à chaque unité de temps alors, en prenant une séquence d'états, le premier état est l'état dans lequel on se trouve à l'instant un, le deuxième état est l'état dans lequel on se trouve à l'instant deux, etc. On peut de cette manière exprimer le fait qu'il est inévitable d'atteindre un état satisfaisant la propriété p avant deux unités de temps avec la formule p ∨Op∨OOp. Il est évident que cette manière de procéder génère des formules très lourdes. Cela peut être évité en se définissant des notations appropriées. La possibilité de pouvoir exprimer des propriétés temporelles avec un temps discret peut être intéressante pour exprimer des propriétés de systèmes où une action n'est effectuée que lors d'un tick d'une horloge contenue dans le système.
Les propriétés d'équités, comme les propriétés de sûreté, n'ont pas de mal à être exprimée en LTL. La propriété d'équité faible selon laquelle une propriété p continuellement vraie implique qu'une autre propriété q l'est infiniment souvent s'exprime en LTL comme
true U ¬(true U ¬p) → ¬(true U ¬(true U q)) et la propriété d'équité forte selon laquelle le fait que p soit vérifié infiniment souvent implique que q l'est aussi s'exprime en LTL comme¬(true U ¬(trueU p)) → ¬(true U ¬(true U q)), l'implication étant définie classiquement.

2.2 La logique CTL ([35])

2.2.1 Description :

La logique CTL est un exemple de logique temporelle de branchement non temporisée (untimed branching time logic).
Sa syntaxe se définit comme suit:
Une formule ϕ est une formule CTL si elle peut être construite à partir de la grammaire suivante:

$$\phi \rightarrow p \mid \neg\phi \mid \phi \wedge \phi \mid \exists O\phi \mid \forall O\phi \mid \phi \exists U \phi \mid \phi \forall U \phi O$$

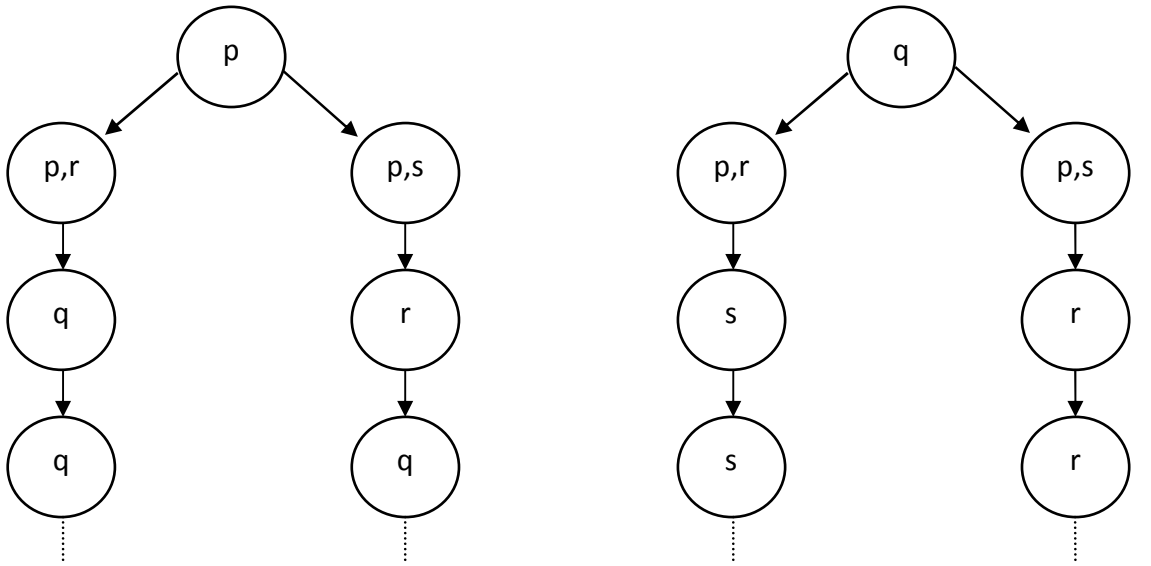
Nous introduisons maintenant la sémantique des opérateurs de CTL:
Une formule CTL est évaluée par rapport à un arbre de branchement n'ayant pas de branches finies.
Ayant un arbre de branchement de racine s_0
$s_0 \models p$ ssi $p \in \mu(s_0)$.
$s_0 \models \neg\phi$ssi$s_0 \not\models \phi$.
$s_0 \models \phi_1 \wedge \phi_2ssis_0 \models \phi_1$ et $s_0 \models \phi_2$.
$s_0 \models \exists O\phi$ssi s' $\models \phi$ pour un état s' successeur de s_0.
$s_0 \models \forall O\phi$ssi s' $\models \phi$ pour tout état s' successeur de s_0.
$s_0 \models \phi_1 \exists U \phi_2$ssi pour une branche de l'arbre de branchement (s_0, s_1, s_2,...), $\exists i \geq 0$ tel que

$$s_i \models \phi_2 \text{ et } \forall\, 0 \leq j < i,\ s_j \models \phi_1.$$

$s_0 \models \phi_1 \forall U \phi_2$ssi pour toute branche de l'arbre de branchement (s_0, s_1, s_2,...), $\exists i \geq 0$ tel que

$$s_i \models \phi_2 \text{ et } \forall\, 0 \leq j < i,\ s_j \models \phi_1.$$

L'idée des opérateurs ∃O et ∀O est la même que pour l'opérateur LTL O. Une formule CTL étant évaluée sur des arbres, un état peut avoir plusieurs successeurs. Dans le cas de l'opérateur ∃O, au moins un successeur doit satisfaire une propriété donnée tandis que dans le cas de ∀O, ce sont tous les successeurs.
L'opérateur ∃U demande qu'on puisse atteindre, de l'état racine de l'arbre, un état satisfaisant une propriété donnée avec les états intermédiaires satisfaisant une seconde propriété.



a) Les états d'une branche satisfont p **b)** q est directement satisfait jusqu'à ce que q soit satisfait

Figure 10. Arbre satisfaisant p ∃U q

L'arbre de branchement représenté à la figure 10.a satisfait la formule p ∃*U* q. Partant de l'état racine, la branche gauche permet d'atteindre un état satisfaisant q avec tous les états intermédiaires, y compris l'état racine, satisfaisant p. Par contre cet arbre ne satisfait pas la formule p ∀*U* q. Pour cela, il faut que toutes les branches de l'arbre permettent d'atteindre un état satisfaisant q, avec les états intermédiaires satisfaisant p. Or, pour atteindre un état satisfaisant q par la branche droite de l'arbre, on passe par un état ne satisfaisant que r (et donc pas p).
L'arbre de la figure 10.b, quant à lui, satisfait également la formule p ∃*U* q. En effet, rien n'oblige dans la sémantique de CTL de transiter par au moins un état satisfaisant p et l'état racine peut directement satisfaire q. Pour cette même raison, cet arbre satisfait également la formule p ∀*U* q.

2.2.2 Expressivité :

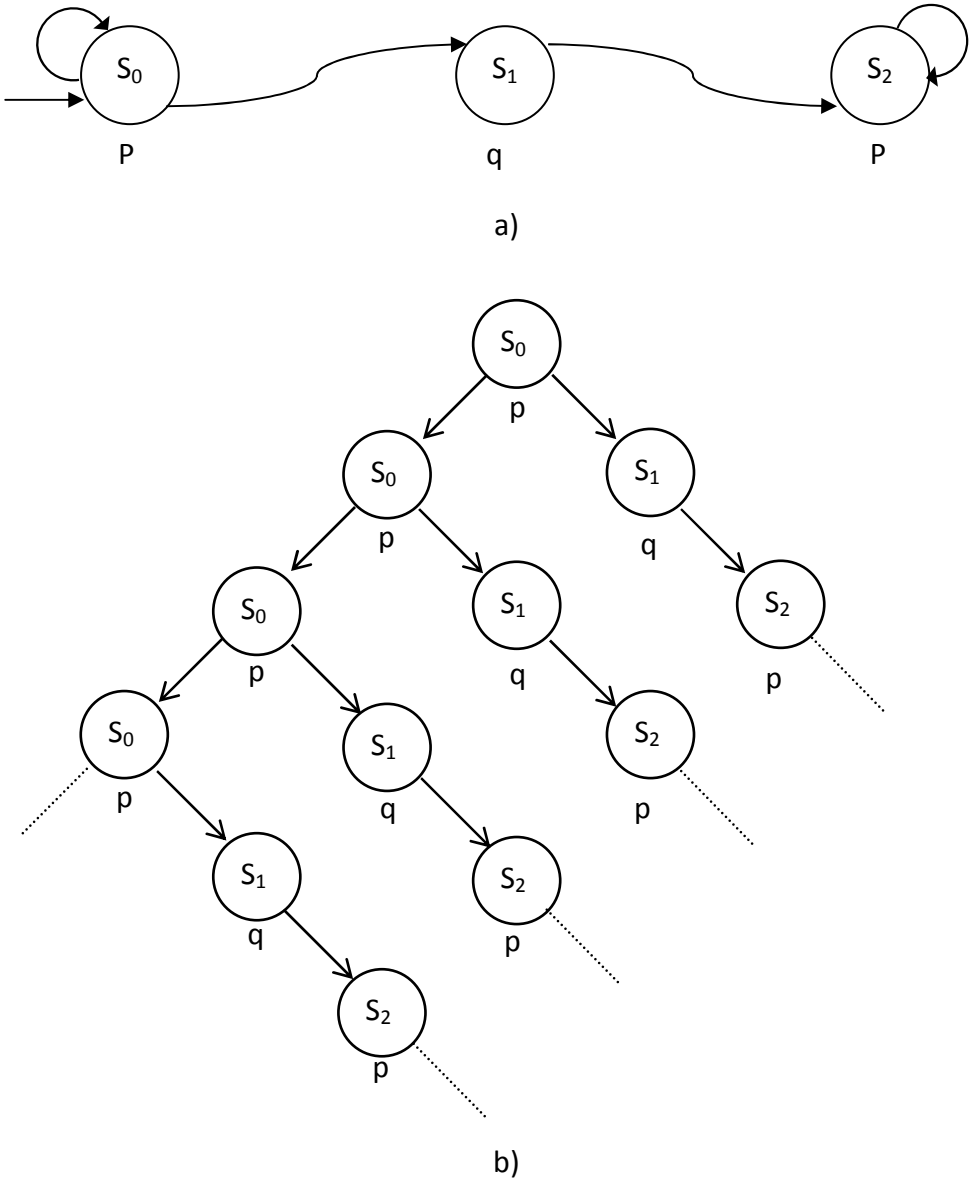


Figure 11

Les opérateurs ∀*U* et ∃*U* permettent, contrairement à LTL, de pouvoir différencier l'atteignabilité potentielle de l'atteignabilité inévitable. Par contre, il est dit dans [38] que les propriétés d'équité faible ne sont pas exprimables en CTL. En effet, la propriété exprimant le fait qu'il est inévitable d'atteindre un état à partir duquel une propriété p est toujours vérifiée n'est pas exprimable en CTL. On pourrait penser pouvoir exprimer cette propriété grâce à la formule true∀*U*¬(true∃*U*¬p). Cette formule spécifie le fait qu'il est inévitable d'atteindre un état à partir duquel tous les états accessibles satisfont p.
Si on regarde la structure de Kripke représentée par la figure 11.a, on peut remarquer que quelle que soit l'exécution de l'automate, on atteint inévitablement lors de l'exécution un état à partir duquel tous les états traversés ensuite sont labellés par p. Cependant, si on

déploie ce modèle en un arbre de branchement (représenté à la figure 11.b) et qu'on évalue la formule sur celui-ci, l'évaluation aboutit à false. En effet, chaque état de la branche de gauche a deux états successeurs, l'un satisfaisant p, l'autre q. La formule ne représente donc pas la propriété voulue. Pour spécifier des propriétés d'équité faible pour des systèmes non temporisés, on utilisera plutôt une logique linéaire comme LTL. Par contre, la simplicité d'un algorithme de Model Checking CTL (présenté au chapitre suivant) rend cette logique attrayante d'un point de vue pratique.
Il n'y a par contre pas de problème à exprimer le fait qu'une propriété p est infiniment souvent vérifiée. Ceci est fait grâce à la formule ¬(true∃*U*¬(true∀*U* p)). Il est donc tout à fait possible d'exprimer une propriété d'équité forte.
De la même manière qu'en LTL, on peut simuler un temps discret en posant certaines conditions sur les arbres de branchement. Grâce aux opérateurs ∀O et ∃O, il est maintenant possible de spécifier le fait qu'il est possible d'atteindre un état satisfaisant p avant deux unités de temps avec la formule p ∨∃Op∨∃O∃Op. Pour spécifier que la propriété est inévitable, on utilise l'opérateur ∀O.

On vient de voir deux types de logiques temporelles : LTL et CTL. Toutefois, il existe d'autres logiques temporelles comme TLA, CTL*, PLT. . .. L'aspect important de la logique temporelle est son aspect expressif, c'est à dire sa capacité de décrire différentes classes de propriétés.
Évidement la logique CTL est plus puissante que la LTL puisque cette dernière étend la CTL. Néanmoins, la logique linéaire est utilisable et bien simple pour exprimer des comportements (actions) qui ne nécessitent pas des branchements.
Cependant, « le but de la logique temporelle est de proposer des formalismes de haut niveau permettant d'analyser de façon simple une classe déterminée de propriétés ».

Partie 3

Chapitre 5

Le model CHECKING

Chapitre 5

Le model CHECKING

Sommaire

Introduction

Plusieurs méthodes formelles basées sur les sémantiques des programmes sont possibles : la génération automatique de tests, la preuve et le model checking. La génération automatique de tests nécessite d'envisager tous les cas possibles ce qui n'est pas aisé. La preuve consiste à prouver des propriétés automatiquement d'après une description du système, un ensemble d'axiomes et un ensemble de règles d'inférence. Cependant, le temps et les ressources nécessaires pour que les propriétés soient vérifiées peut dépasser des temps acceptables ou les ressources disponibles.

Le model checking semble alors être la méthode la plus efficace notamment grâce à des techniques d'abstractions permettant de pallier aux problèmes de temps et de disponibilité des ressources tout en garantissant une vérification complété par l'énumération exhaustive des états accessibles pendant l'exécution du système. Néanmoins, la principale difficulté dans cette approche est de mettre en place une technique de réduction efficace de l'espace d'état, qui explose automatiquement, tout en garantissant la vérification du système.

Pour vous expliquer notre travail, on commence par exposer le contexte scientifique dans lequel on a effectué nos recherches, notamment avec une présentation du model checking, implémenté au sein de deux outils très utilisé à savoir SimGrid et SPIN

1. Le model checking

Le model checking [46] désigne une famille de techniques de vérification automatique de Systèmes dynamiques. Il s'agit de vérifier si un modèle donné, le système lui-même ou une abstraction, satisfait une spécification, telles que l'absence d'interblocage (deadlock), par exemple.
Pour cela, il faut analyser tous les cas d'exécutions possibles du système. A l'issue de cette Vérification, si une propriété n'est pas satisfaite, un contre-exemple est fourni permettant ainsi d'identifier un cas d'exécution non conforme et ainsi localiser et corriger la source de l'erreur.
Ce contre-exemple peut être obtenu dès l'instant ou la propriété à vérifier n'est pas satisfaite durant l'exécution, ce qui permet une vérification plus rapide qu'avec les techniques traditionnelles, Au-delà du fait que celle-ci est automatique.

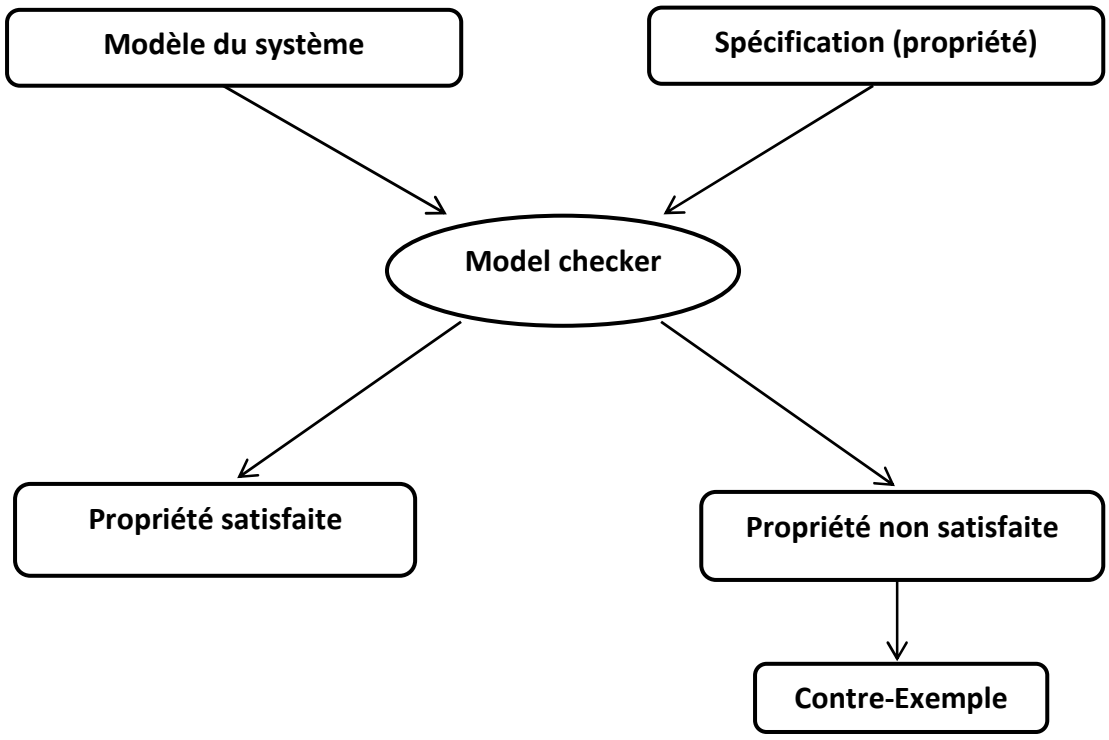


FIGURE 1 – Principe du modèle checking

1.1 Système de transitions et espace d'état

Pour vérifier toutes les exécutions possibles du système étudié, on représente son modèle sous la forme d'un système de transitions à états, ou chaque état correspond à un état du système à chaque pas d'exécution et chaque transition est une évolution possible d'un état donné vers un autre, selon l'action exécutée par le système. Ce système de transition peut être représenté par un graphe orienté ou automate, une machine de Turing ou bien un réseau de Pétri.

1.1.1 Définition (Système de transitions a états)

Un système de transitions S est un couple :

S = (Q, I, T, ->) tel que :
- Q est l'ensemble des états
- I est l'ensemble des états initiaux tel que $I \subseteq Q$
- T est l'ensemble des transitions/actions
- $-> \subseteq Q \times T \times Q$ est la relation de transition.

Une action/transition $t \in T$ est dite exécutable si et seulement si il existe un état $q \in Q$ et un état $q' \in Q$ tels que $q \xrightarrow{t} q'$.
Une exécution de S est une séquence infinie $\sigma = q_0 q_1 \ldots$ d'états $q_i \in Q$ tels que $q_0 \in I$ et pour tout $i \in \mathbb{N}$, il existe $(q_i, T_i, q_{i+1}) \in ->$ pour un certain $T_i \in T$
Un état $q \in Q$ est dit accessible si $q_i = q$ pour une certaine exécution $\sigma = q_0 q_1 \ldots$ de S et un $i \in \mathbb{N}$.
De plus, dans le cas des applications distribuées, nous travaillons avec des systèmes non déterministes, i.e., pour un état du système à un instant donné pendant l'exécution, il peut exister plusieurs transitions exécutables à partir de cet état. Formellement, cela signifie qu'il peut y avoir une transition $t \in T$ et trois états $q, q'q'' \in Q$ tels que $q \xrightarrow{t} q'$et $q \xrightarrow{t} q''$. On peut donc aller arbitrairement à q' *ou* q'' à partir de q. De plus, l'exécution est séquentielle, i.e., chaque processus ne peut exécuter qu'une seule action à la fois.

L'ensemble des états étudiés correspond alors à l'espace d'état. L'espace d'état est l'ensemble des états du système de transitions accessibles à partir d'un état initial en suivant les chemins du système de transitions. L'espace d'état a une structure plus simple que le système de transitions complet puisqu'on ne garde que les états en n'omettant la structure du graphe.

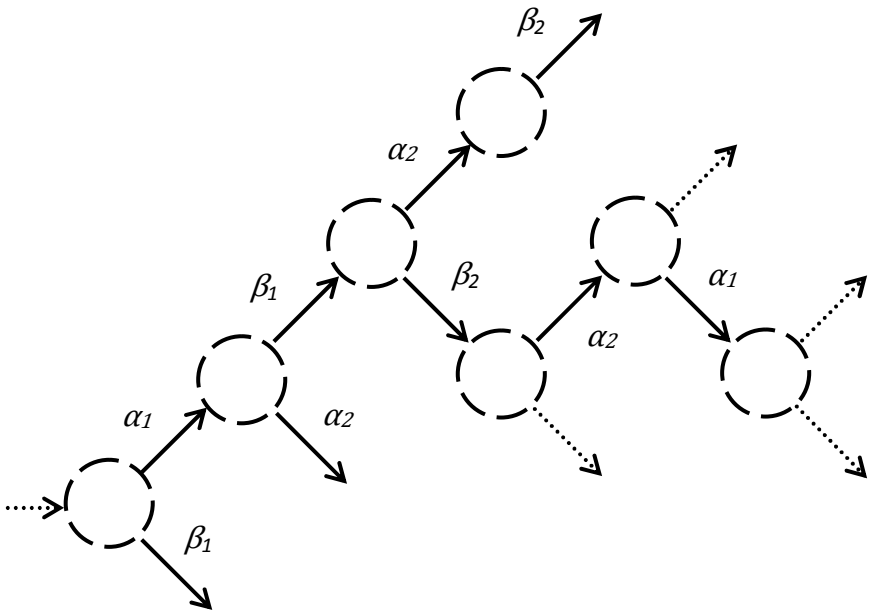


FIGURE2 – Exploration de l'espace d'état

Selon le type de programme étudié et surtout la spécification du modèle adoptée, la construction de l'espace d'état peut se faire soit dynamiquement, i.e., au fur et à mesure de l'exécution, soit à priori, i.e., que celui-ci est connu avant le début de l'exécution. Nous verrons par la suite comment cette caractéristique joue un rôle important dans l'implémentation du model checker. Dans notre cas, nous nous intéressons aux programmes en C qui seront exécutés à travers SimGrid, l'espace d'état sera donc construit dynamiquement pendant son exploration, en même temps que l'on effectue la vérification d'une propriété.

1.2Propriétés de sûreté et propriétés de vivacité

Parmi les propriétés vérifiées par model checking, plusieurs catégories sont possibles : les propriétés d'accessibilité (ex : « il existe un état accessible où x vaut 0 »), les propriétés d'invariance ou sûreté (ex : « durant toute l'exécution, x est différent de 0 ») ou bien les propriétés de vivacité (ex : « l'application se terminera finalement »).

Actuellement le model checker de SimGrid permet la vérification de propriétés de sûreté. L'objectif est d'intégrer la vérification d'une autre catégorie de propriétés : les propriétés de vivacité.

1.2.1 Propriété de sûreté (safety property)

Une propriété de sûreté permet d'exprimer le fait que « quelque chose de mauvais n'arrivera jamais ». Ainsi, l'invariance est un cas particulier de sûreté. Par exemple, « deux processus ne peuvent être au même moment en section critique », « le système n'a pas de deadlock », « une variable n'est jamais égale à 0 », etc. Ce type de propriété doit être vérifié et satisfait à chaque étape de l'exécution du système. Dans le cas où la propriété n'est pas satisfaite, on peut obtenir un contre-exemple fini.
Pour leur vérification, il est possible d'utiliser la logique temporelle mais également de simples assertions. La deuxième approche est actuellement implémentée dans le model checker de SimGrid.

1.2.2Propriété de vivacité (Liveness property)

Une propriété de vivacité exprime le fait que « quelque choses de bien arrivera finalement ». On garantit donc qu'une propriété sera vraie à partir d'un certain état de l'exécution du système. Ce type de propriété va donc être évalué sur un ensemble d'états représentant un chemin d'exécution du système. Par exemple, « l'application se terminera finalement », « si un client envoie une requête, il recevra finalement une réponse » ou « chaque processus sera finalement en section critique » (exclusion mutuelle), etc. Dans ce cas, nous travaillons donc sur un temps infini contrairement à une propriété de sûreté.

Pour formuler ce type de propriété, il est nécessaire d'utiliser un langage plus riche permettant d'exprimer le temps infini, comme la logique. Les logiques sont des langages de spécification formels, i.e., ils ont une définition mathématique précise ce qui permet d'exprimer les propriétés sans ambigüité mais cela permet surtout la simulation et la vérification automatique du système. Pour les propriétés de vivacité, on utilise des logiques temporelles [41] telles que LTL (Linear Temporal Logic), qui permet de considérer une propriété sur le long d'un chemin d'exécution grâce aux connecteurs et non plus sur un seul état, ou bien, la logique CTL (Computation Tree Logic[42]) dite *branchante*, qui permet de prendre en compte les entrelacements des futurs possibles à un point donné de l'exécution contrairement à LTL qui est linéaire. Ceci permet de quantifier sur les futurs possibles.
Pour une première approche dans SimGrid, le choix est porté de ne travailler que sur l'expression et la vérification de propriétés temporelles formulées en LTL.

1.2.2.1 Logique Temporelle Linéaire (LTL)

La Logique Temporelle Linéaire (LTL) est définie par :

- Un ensemble de propositions atomiques AP ou variables de propositions, i.e., des expressions booléennes portant sur des variables, des constantes et des prédicats.
- Des connecteurs logiques tels que : AND, OR, NOT, ⇒
- Des modalités :
  - **X** : *neXt* (demain) :**X***p* signifie que *p* est vrai dans l'état suivant le long de l'exécution.

- **F** : Finally (un jour) : **F***p* signifie que *p* est vrai plus tard au moins dans un état de l'exécution.
- **G** : Globally (toujours) : **G***p* signifie que *p* est vrai dans toute l'exécution.
- **U** : Until (jusqu'à) : *p***U***q* signifie que *p* est toujours vrai jusqu'à un état où *q* est vrai.

Exemple1. La propriété d'invariance « durant toute l'exécution, x est différent de 0 » se traduit en logique LTL par **G**¬(x=0).

Exemple 2. **G**(*p*→**F***p*) signifie « pendant toute l'exécution, si *p* est vrai à un état donnée, alors *q* sera vrai plus tard au moins dans un état suivant »

Chaque proposition atomique ainsi que la composition booléenne de plusieurs propositions atomiques avec des connecteurs est une formule interprétée sur le comportement du système, i.e., sur une séquence d'états représentant une exécution du système.

1.2.2.2Structure de Kripke

L'utilisation de logique temporelle introduit alors une nouvelle représentation du système sous forme de structure de Kripke [43]. La structure de Kripke est dérivée du système de transitions et modifiée avec les informations utiles au model checking. De plus, on supprime l'ensemble *T* des transitions/actions car on ne s'intéresse qu'à des propriétés sur les suites d'états visités.

Définition (Structure de Kripke). Soit *AP* un ensemble de propositions atomiques. Une structure de Kripke est un quadruplet *M=(Q, I, R, L)* tel que :

- *Q* est l'ensemble des états
- *I* est l'ensemble des états initiaux tel que $I \subseteq Q$
- $R \subseteq QxQ$ est une relation de transition qui vérifie : $\forall q \in Q, \exists q' \in Q$ tel que $(q, q') \in R$
- $L : Q \rightarrow 2^{AP}$ est une fonction d'étiquetage qui définit pour chaque état $q \in Q$ l'ensemble *L(q)* de toutes les propositions atomiques qui sont vraies dans *q*

Pour vérifier tous les chemins d'exécutions possibles du système, il est utile de déplier cette structure afin d'obtenir un arbre infini dont la racine est l'état initial de la structure, et chaque nœud de l'arbre a pour successeurs ceux obtenus par la relation de transition *R*. Ainsi, on visualise plus facilement les exécutions possibles du système.
Les formules LTL sur une structure de Kripke sont de la forme Af où f est une formule LTL de chemin et A l'unique quantificateur autorisé en début de formule, permettant de quantifier tous les chemins d'exécution du système. La formule est donc satisfaite si tous les chemins de la structure de Kripke dépliée satisfont f.

1.2.2.3 Automate de Büchi

Pour vérifier les propriétés de vivacités formulées en terme de Logique Temporelle Linéaire, on utilise un automate de Büchi [44], qui est un automate étendu pour travailler sur des mots infinis.

Définition (Automate de Büchi).

Un automate de Büchi est un quintuplet B = (Σ, Q,→,q_0, F) tel que :

- Σ est un alphabet fini
- Q est l'ensemble des états
- →∈ Q x Σ x Q est la relation de transition entre les états
- q_0∈ Q est l'état initial
- $F \subseteq Q$ est l'ensemble des états finaux dits acceptants

Un mot w est une suite de lette $l_1 \dots l_n$, finie ou infinie, appartenant à un alphabet fini Σ.

Un mot fini est accepté par un automate s'il est possible d'aller de l'état initial $q0 \in Q$ à un état final $q_f \in F$ de l'automate en prenant une transition étiquetée par l_1 (première lettre du mot) puis une transition étiquetée parl_2 (seconde lettre du mot), etc.

Un mot infini est accepté par un automate de Büchi s'il part de l'état initial q_0, respecte la relation de transition et *passe infiniment souvent par un état acceptant* $q_f \in F$.

L'ensemble des mots acceptés par un automate *A* est noté $L(A)$.

Exemple 1. Cet automate de Büchi non déterministe reconnaît les mots infinis contenant un nombre fini de **a**.

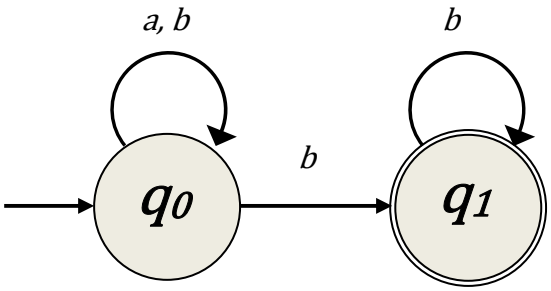


FIGURE3 – Exemple d'automate de Büchi

Pour déterminer si la propriété est satisfaite sur l'ensemble des chemins d'exécutions possibles du système, on représente la négation de la formule LTL. Si l'évolution du système amène à une acceptation de l'automate, cela signifie que la négation de la propriété est reconnue et donc par conséquent qu'il existe un chemin d'exécution pour lequel la propriété n'est pas satisfaite par le système.

Exemple 2. Soit la formule LTL ϕ= $\boldsymbol{GF}p \rightarrow \boldsymbol{G}(q \rightarrow \boldsymbol{F}r)$. L'automate de Büchi A_ϕ est :

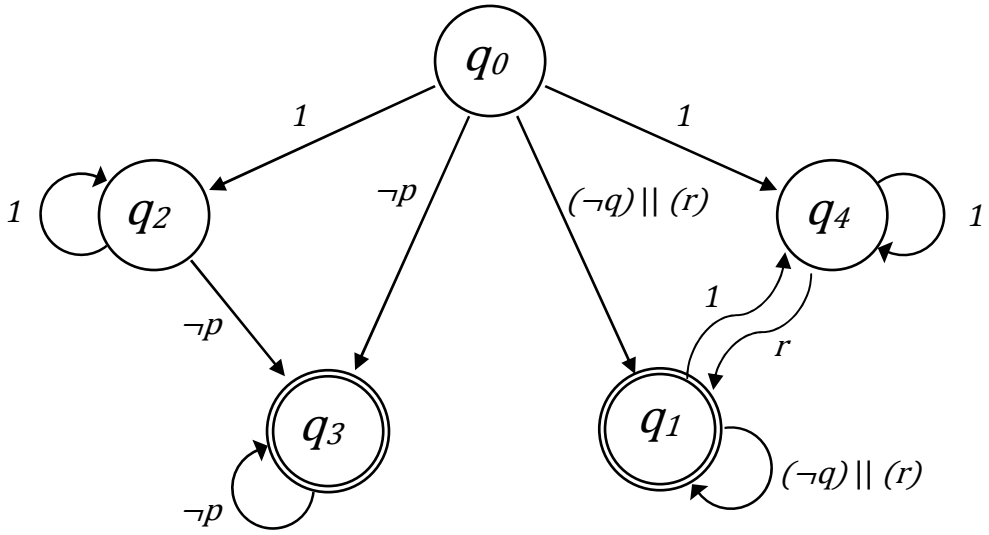


FIGURE 4 – Automate de Büchi A_ϕ *avec* ϕ = **GF***p*→**G**(*q*→**F***r*)

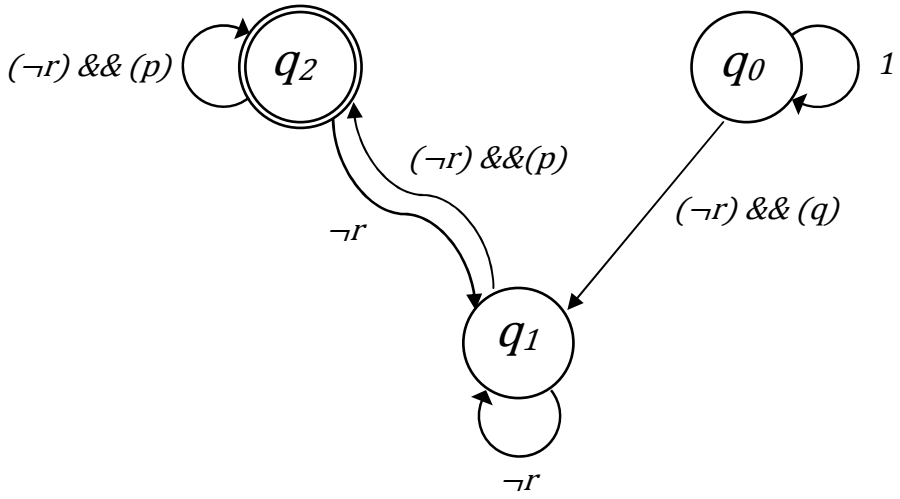


FIGURE 5 – Automate de Büchi $A_{\neg\phi}$ *avec* ϕ= **GF***p*→**G**(*q*→**F***r*)

Cet exemple met en avant plusieurs caractéristiques spécifiques aux automates de Büchi. Les transitions étiquetés par un 1 sont des transitions toujours exécutables, quelque soit les valeurs des propositions atomiques. Toutefois, il n'est pas obligatoire de les franchir. L'automate est donc non-déterministe. Il est possible de transformer ce dernier en automate de Rabin [45] déterministe. Cependant, le pouvoir de reconnaissance des automates déterministes est moindre dans les cas des mots infinis, c'est pourquoi on conserve le non-déterminisme éventuel.

Si l'on observe les états finaux dits acceptants, contrairement aux automates classiques, il est possible de sortir de ces états pour obtenir un cycle d'acceptation, condition d'acceptation d'un automate de Büchi. Ainsi, $q_0q_4q_1q_4q_4q_1$… est un cycle d'acceptation dans l'exemple 2.

2. le modèle checker SPIN : (Simple Promela INterpreter)

Le modèle checker SPIN [68] a été conçu pour vérifier et valider des systèmes, il permet de décrire un modèle grâce a un langage de haut niveau appelé PROMELA
PROMELA est un langage impératif qui ressemble au langage C agrémenté de quelques instructions pour la concurrence. Le modèle PRPMELA peut ensuite être analysé avec Spin par :

- *Simulation* : le modèle est exécuté pas à pas, ce qui permet de se familiariser avec son comportement.
- *Vérification* : les états du modèle sont explorés exhaustivement pour vérifier que le modèle satisfait des propriétés (par exemple, exclusion mutuelle) spécifies en LTL (Linear Temporal Logic).

2.1 PROMELA

Un programme Promela est une liste de déclarations de *types, constantes, variables* (globales) *et processus*, suivie d'un point d'entrée du programme (init).

Déclaration de types : Les types de base de Promela sont : *bit* ou *bool*, *byte*, *short* et *int*;
ces types n'ont pas besoin d'être déclarés. Le seul type que l'utilisateur peut être déclaré est le type énumérée *mtype* pour lequel on peut spécifiér ses valeurs (constantes symboliques).
Exemple : *mtype* = {ack, nak, err } ;

Déclaration de constantes : Promela permet les macro-définition à la C. Il est donc possible de déclarer des constantes comme en C avec *#define*, mais aussi des macros plus compliquée

Déclaration de variables globales : les variables scalaires ou tableau de Promela sont déclarées comme en C : on indique le type, le nom de la variable et optionnellement sa valeur initiale.

Bool b1 = false, b2 = false ;

bit k = 0;

bit porteouverte[3];

Une classe spéciale de variables sont les canaux, qui représentent des queues FIFO d'une longueur constante et dont les éléments (appelés messages) sont des tuples

(typés) de valeurs scalaires ou vecteur de Promela. Les canaux sont déclarés en utilisant le mot clé *chan* suivi du nom du canal et optionnellement de sa longueur et du type des messages qui circulent. Par exemple :

Chan Ouvreporte = [0] of {byte, bit},
Transfert = [2] of {bit, short, chan}

Ouvreporte est un canal synchrone, car sa longueur est 0, ce qui correspond à un rendez-vous ; sur ce canal circulent des messages ayant une partie byte et une partie bit. Transfert est un canal asynchrone, car il peut stocker au plus deux messages.

Déclaration de processus : La formule la plus simple de déclaration de processus est :

proctype nom (paramètres_formels)
{ instructions }

Où les paramètres formels sont déclarés comme en C mais séparés par des point-virgules.

Un processus peut être lancé de deux manières. La manière implicite, qui peut s'appliquer que si la liste de paramètres formels est vide, consiste à préfixer *proctype* par le mot clé *active*. Il est même possible de lancer un nombre fixe de copies du processus en utilisant la manière implicite.
Par exemple :
active proctype ascenseur () { /* un processus `ascensseur' lancé */
...
}
active [3] proctype porte () { /* trois processus `porte' lancés */
...
}
La manière explicite, consiste à utiliser l'instruction *run* dans le point d'entrée du modèle ou dans un autre processus :
run nom (paramètres_actuels)

Le corps des processus est une liste de déclaration de variables (scalaires, tableau ou canal) suivie d'une liste d'instructions séparés par point-virgule. Le ; est séparateur d'instructions et non fin d'instruction comme en C!

Instructions : Promela s'inspire peu de la syntaxe du C pour les instructions. Les points communs avec le C sont : l'affectation "=" et le test d'égalité de valeurs "==", l'utilisation des accolades pour délimiter les blocs d'instructions, la syntaxe des opérations sur les bits et les entiers (++, --, etc.), l'utilisation des étiquettes sur les lignes et l'instruction goto, l'utilisation des expressions comme instructions.
Dans ce dernier cas, une expression peut être utilisée comme une instruction si elle ne fait pas des effets de bord ; alors elle est exécutable quand sa valeur devient vraie

(par le changement des valeurs des variables partagées). Par exemple, dans le code suivant :
1: a = b+1;
2: (a == b)
L'exécution reste bloquée à la ligne 2 tant que a et b ont des valeurs différentes (la valeur d'une de ces variables peut être changée par des processus parallèles).
L'instruction vide est **skip**.
L'affectation, l'instruction vide, l'expression et le lancement d'un processus sont des instructions atomiques (indivisibles, exécutées sans entrelacement avec les autres processus)
en Promela. Les autres instructions ne le sont pas. Pour rendre atomique un bloc d'instructions, on doit utiliser l'instruction "atomic". Par exemple :

```
atomic {
(state==1);  state = state+1
}
```

La plus simple instruction de sélection est l'instruction "if", qui s'approche plus au niveau de la syntaxe du *switch* de C, avec une sémantique différente. En effet, contrairement au *switch*, le test des différents cas est fait de façon non-déterministe (aléatoire) et si aucune branche ne peut être sélectionnée, l'exécution est bloquée en attente de l'activation d'une sélection. La syntaxe générale est :

```
if
        :: sequence_d_instructions_1
        :: ...
        :: sequence_d_instructions_N
fi
```

L'instruction d'itération **do** a une syntaxe similaire au if, sauf que après l'exécution de l'instruction sélectionnée le contrôle passe au début de l'instruction do jusque à l'exécution d'une instruction "break". Par exemple, le code suivant aura une exécution infinie :

```
proctype ascenseur () {
        show byte etage = 1; /* show: faire apparaître la valeur à la simulation */
        do
        :: (etage != 3) -> etage++
        :: (etage != 1) -> etage--
        od
}
```

<u>*Point d'entrée du modèle*</u> :
L'exécution du système commence par la section *init* qui est exécutée en parallèle avec les éventuels processus actifs. Le corps de la section *init* est le même que celui des processus : une liste de déclarations de variables locales et une liste d'instructions.

Par exemple :

```
init {
        run porte(1);  run porte(2);  run porte(3);
        run ascenseur()
}
```

Propriétés LTL : Les opérateurs LTL disponibles sont ceux listés dans l'interface : les opérateurs booléens (l'implication est notée→), opérateur globalement (notation []), l'opérateur dans le futur (notation <>) et l'opérateur *until* (notation U).
Les opérandes doivent être des propositions. Pour définir des propositions, utilisez des macros C dans la partie *Symboles Définition*. Vous pouvez ici faire référence aux variables globales du modèle, aux variables locales des processus (en préfixant par le nom du processus, par exemple *porte[2].etage*), ou à une étiquette dans le corps d'un processus (par exemple *porte[2]@l1*).
Il faut sauvegarder les formules écrites dans des fichiers à extension *.ltl* (ce n'est pas le même fichier que la spécification Promela, dont l'extension consacrée est *.pml*).

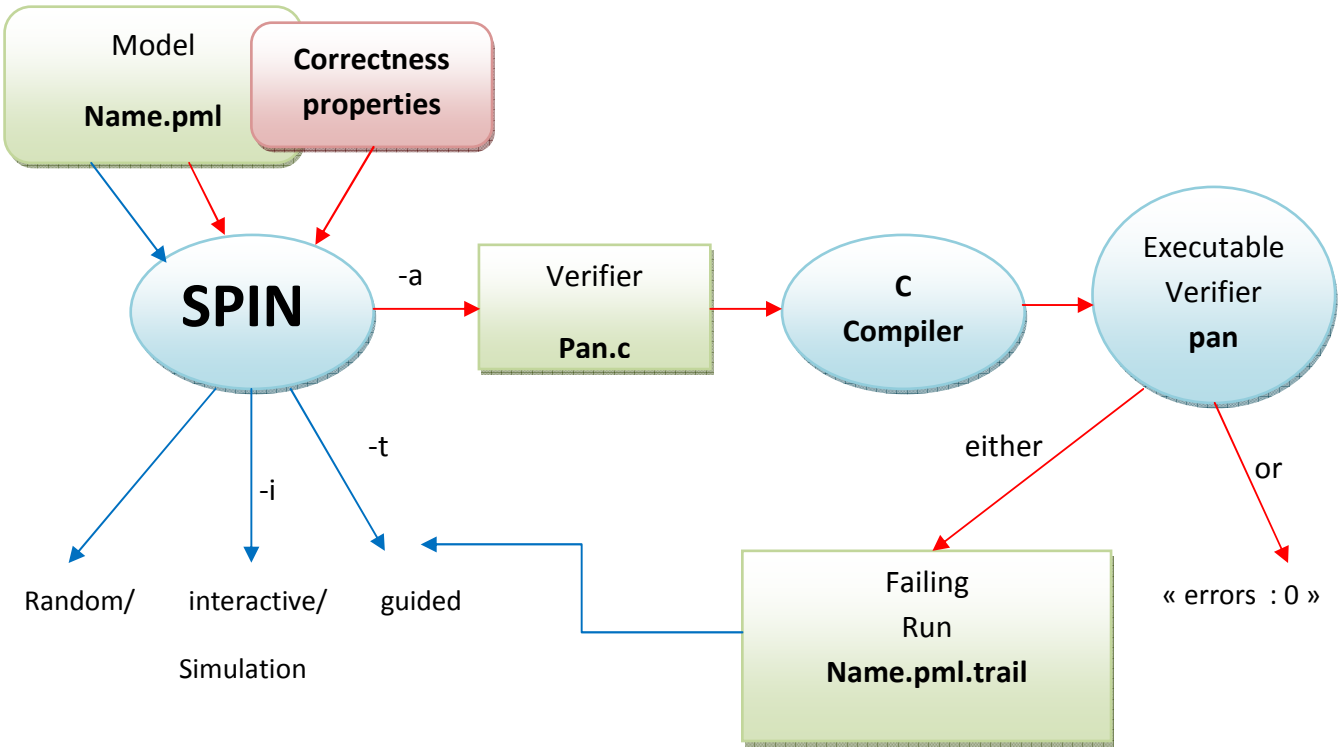
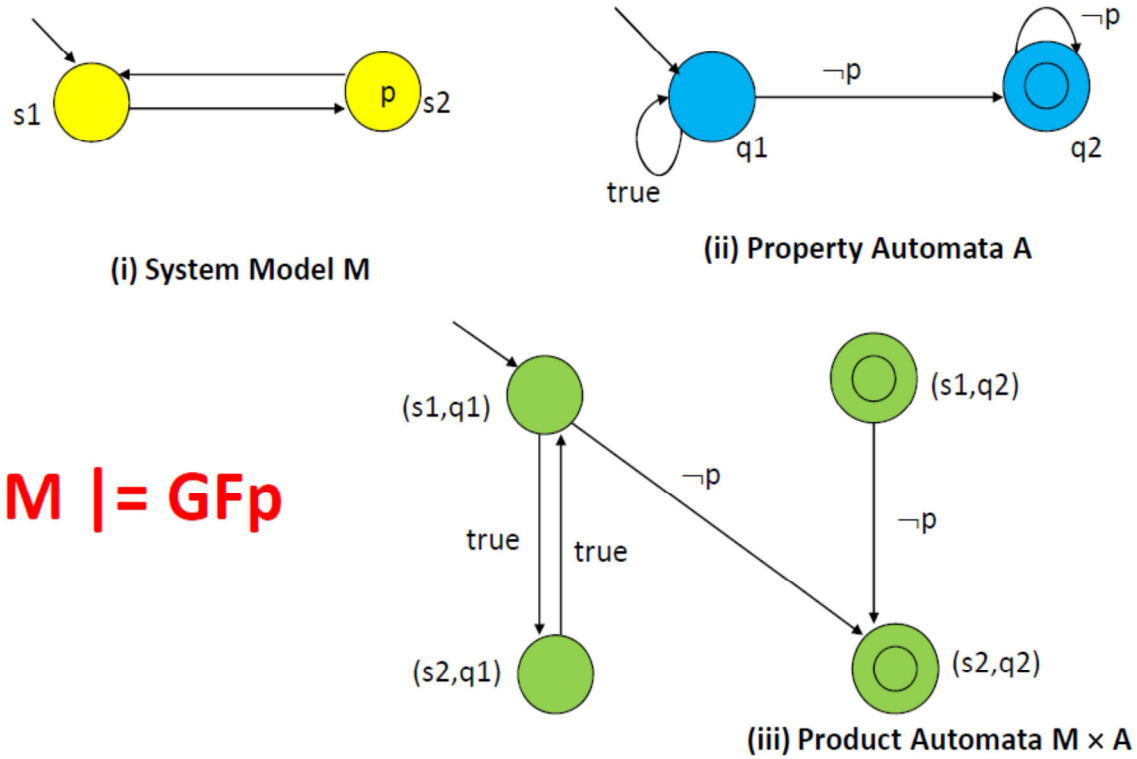


FIGURE 6 Le model checker SPIN

2.2 Vérification de propriétés :

Pour la vérification des propriétés du système, et a la différence du model checker SimGrid ; le parcours va s'exécuter sur le produit synchronisé du modèle du système et de l'automate de Büchi de la négation de la propriété. Cependant, SPIN adopte les deux types de parcours de base ; et en profondeur et en largeur. Cette caractéristique permet de sélectionner, selon le type de système à vérifier, le type de parcours a appliquer ce qui permet de donner des performances meilleures en terme de ressources.

Nous montrons dans les schémas ci-dessous la façon de construire le système d'état à explorer :



Principe :

Pour déterminer si la propriété est satisfaite sur l'ensemble des chemins d'exécutions possibles du système, on représente la négation de la formule LTL. Si l'évolution du système mène à une acceptation de l'automate, cela signifie que la négation de la propriété est reconnue et donc par conséquent qu'il existe un chemin d'exécution pour lequel la propriété n'est pas satisfaite par le système.

Partie 3

Chapitre 6

ETUDE DU DÉVELOPPEMENT DU MODEL CHECKER SimGrid

Chapitre 06

ETUDE DU DÉVELOPPEMENT DU MODEL CHECKER SimGrid

Sommaire

Introduction

Un model checker a été développé au sein de SimGrid et permet de vérifier des propriétés de sureté sur des programmes C. Au-delà d'un développement vers la vérification de propriétés de vivacité, l'objectif à long-terme est de permettre en même temps la simulation et la vérification par model checking d'applications.

Dans ce chapitre nous allons étudier les deux algorithmes de parcours d'espace d'état, implémenté par le model checker de SimGrid, afin de mieux comprendre le principe de la vérification de propriétés. Ainsi que les deux approches stateless et stateful; leurs apport et leurs cout en terme de ressources

Cette étude nous a permet d'avoir une idée sur la taille du système à parcourir pour vérifier un certain type de propriété

1. Principe du model checker de SimGrid

1.1 Stateful model checking pour les propriétés de sûreté

Pour vérifier les propriétés sûreté, le model checker de SimGrid utilise un algorithme de parcours en profondeur (Depth First Search) permettant une présentation explicite des états en travaillant sur des mots finis ou infinis. Grace à cette approche, il est possible de vérifier un modèle sans nécessairement construit l'ensemble de ses états.

Algorithme 1 Parcours en profondeur (DFS)

```
DFS (Model M, state s)
  Set_visited(s) ; /* pour ne pas le considérer plusieurs fois, optionnel */
  foreach s' in successors(s) do
        If not_visited(s') then
          DFS (M, s');
        end if
  end foreach
```

Le principe avec cet algorithme est de partir de l'état initial du système et d'étudier de façon récursive l'ensemble des successeurs possible pour chaque nouvel état. Le système que l'on étudier étant non-déterministe, il est possible à un état donné de l'exécution d'avoir plusieurs transitions exécutables et donc plusieurs successeurs. Chacune de ces exécutions pouvant mener à des états différents, il est nécessaire d'étudier chaque chemin possible. Pour cela, on conserve donc sous la forme d'une pile l'ensemble des états qui ont été créé durant l'exécution mais qui n'ont pas encore été évalués. Ainsi, chaque fois qu'un nouvel état est créé, il est empilé puis dépilé pour être vérifier .Ceci permet donc d'exécuter une vérification complète sur un chemin d'exécution possible, puis lorsqu'il n'y a plus de successeurs possibles, on revient en arrière sur le dernier état créé qui n'a pas encore été vérifier et on avance de nouveau à partir de celui-ci jusqu'à la fin d'un autre chemin.
Initialement, le model checking est effectué de façon stateless, i.e., la représentation de chaque état visité n'était pas conservée. Pour effectuer le backtracking pour le parcours en profondeur, le programme est relancé à partir de l'état initial et on rejoue les actions associées aux états encore présents dans la pile, qui représentent le chemin d'exécution avant d'arriver au point de backtracking. Pour cela, une méthode permet de sauvegarder la représentation complète de l'état initial du système avant son exécution, puis lorsqu'il faut exécuter un retour arrière pour analyser un nouveau chemin d'exécution, on restaure le système dans son état initial et on rejoue chaque transition qui a mené à l'état précédent le niveau de backtracking.

En plus d'une vérification stateless, il est très intéressant de travailler sur une version de model checking stateful en premier pour les propriétés de sûreté. Cette approche consiste à conserver la représentation complète de chaque état visité. Pour cela, on utilise la méthode citée précédemment qui sauvegarde l'état complet du système à chaque étape de l'exécution. Ainsi, au lieu de faire un backtracking à partir de l'état initial du système, il est possible de revenir à un instant donné précis de l'exécution. De plus, ceci permet de ne pas

revisiter les états précédents le point de backtracking, ce qui peut être non négligeable sur le temps de vérification, selon le type d'actions.

Grâce à la sauvegarde de la représentation complète de chaque étape, on peut ajouter l'identification des états déjà visités pour une première optimisation de l'algorithme. Pour cela, nous conservons le hash de la représentation complète de chaque état visité, grâce à une fonction de hashage déjà implémentée dans SimGrid. Ainsi, pour déterminer si un état a déjà été visité, on calcule son hash et on compare avec les hash déjà calculés. Toutefois, un risque de collision subsiste à partir d'un certain nombre d'états. De plus, la probabilité d'obtenir des hash identiques, sans une analyse plus détaillée des représentations, est trop faible pour rendre cette approche suffisante à elle-seule pour réduire significativement le nombre d'états visités.

1.2 Vérification des propriétés de vivacité

Pour vérifier une propriété de vivacité, il faut représenter la négation de la formule LTL en automate de Büchi puis effectuer un parcours en profondeur sur le produit cartésien du modèle du système et de cet automate. Durant ce parcours, on va rechercher des cycles d'acceptation, i.e., une séquence d'états partant de l'état initial et passant un nombre infini de fois par un état acceptant.

1.2.1 Construction de l'automate de Büchi

Pour la construction et l'implémentation de cet automate au sein de SimGrid, il est possible d'utiliser le logiciel LTL2BA [1], développé par D. Odoux et P. Gastin. A partir d'une formule LTL, il génère une représentation graphique de l'automate de Büchi, représentant la formule, mais surtout, une description de l'automate en langage PROMELA (PROtocol Meta LAnguage). Il s'agit d'un langage de spécification de systèmes asynchrones

On peut également identifier les propositions atomiques qui composent la formule, nécessaires pour l'évaluation de l'automate ensuite. Ces propositions vont correspondre à des variables globales du programme que l'on souhaite vérifier, il faut donc pouvoir accéder à leur valeur dans l'algorithme de vérification. Plusieurs solutions peuvent être envisagées : l'utilisateur saisit avec la formule à vérifier des primitives permettant l'accès aux variables ou bien on y accède automatiquement grâce à la librairie DWARF [46] utilisée dans l'implémentation de débuggers.
Dans le cadre de ce travail, le choix a porté sur la première solution pour se concentrer sur le développement du model checker et la vérification de nouvelles propriétés. Toutefois, l'objectif de SimGrid est d'être utilisable à la fois par des utilisateurs non expérimentés et expérimentés dans le domaine. La seconde solution devrait donc être implémentée par la suite, permettant une utilisation simple et rapide du model checker en ne fournissant que le programme à vérifier et la propriété.

1. http ://www.lsv.ens-cachan.fr/~gastin/ltl2ba/

1.2.2 Algorithme de parcours en profondeur avec détection de cycle

Pour la vérification des propriétés de vivacité, le principe est toujours un parcours en profondeur. Cependant, dans ce cas, le parcours va s'exécuter sur le produit cartésien du modèle du système et de l'automate de Büchi de la négation de la propriété. Ainsi, chaque état sera, dans ce cas, une paire d'états composée d'un état du modèle et d'un état de l'automate. Une paire est initiale si l'état du système et l'état de l'automate qui la composent sont respectivement des états initiaux du modèle du système et de l'automate de Büchi. Une paire est dite d'acceptation (finale) si son état de l'automate de Büchi est un état d'acceptation.

Pour leur évolution, le principe est d'avancer d'un pas du coté du système en franchissant une transition exécutable puis, on regarde si cette exécution permet d'évoluer du coté de l'automate.
La nouvelle paire d'état sera alors l'état successeur du modèle obtenu après exécution d'une transition et l'état dans lequel se situe l'automate suite à cette exécution.

Algorithm 2 Double-DFS

```
/*Initialisation */
        stack = empty_stack() ;
        visited = empty_set() ;
        reached = nil ;
        foreach p0 in initial_pairs do
                push(p0, stack);
                set_visited (p' , false, visited);
                DDFS(false);
        end foreach

DDFS (Boolean search_cycle)
        p = top(stack); /* accède au sommer de la pile sans dépiler */
        foreach p' in successors (p) do
        if (search_cycle and (p' == reached)) then
                report acceptance cycle; /* contre-exemple fouri */
                exit ; /* possibilté de poursuivre pour détecter tous les cycles*/
        end if
        if ((p' ; search_cycle) not in visited) then
                push (p', stack);
                set_visited (p', search_cycle, visited);
                DDFS (search_cycle);
                if not search_cycle and p' is accepting then
                        reached = p;
                        DDFS(true);
                end if
            end if
        end foreach
        pop(stack);
```

Dans le cas de la vérification des propriétés de vivacité, nous avons vu précédemment qu'il faut déterminer s'il existe un cycle d'acceptation, i.e., un chemin d'exécution partant de l'état initial et passant un nombre infini de fois par un état acceptant.
On doit donc durant notre parcours, exécuter une détection de cycle dés l'instant où l'on tombe sur une paire d'acceptation (paire dont l'état de l'automate de Büchi est acceptant). Il s'agit donc d'un double-parcours en profondeur (Algorithme 2) [47]. Dés qu'une paire d'acceptation est rencontrée, on active la détection de cycle par un simple booléen tout en continuant le parcours classique sur les successeurs de cette paire. Une fois que l'on arrive à la fin du chemin d'exécution, on retourne au niveau de la paire d'acceptation et on vérifie s'il existe une suite de paire d'états qui mène à la même paire d'acceptation.

Tout comme pour les propriétés de sureté, cet algorithme a été développé en version stateless et stateful. Dans le cas d'une vérification stateless, il est tout de même nécessaire de conserver les paires d'acceptation rencontrées pour déterminer si elles sont rencontrées plusieurs fois de suite, et donc s'il existe un cycle d'acceptation. Pour l'approche stateful, la difficulté vient du fait que nous évoluons parallèlement au niveau du système et au niveau de l'automate de Büchi de la négation de la propriété. De plus, il faut exécuter une transition du coté du système pour connaitre les successeurs possibles et avancer éventuellement dans l'automate. Sachant qu'il peut y avoir plusieurs transitions exécutables à partir d'un même état, pour évaluer et créer tous ses successeurs, il faut exécuter chacune des transitions en revenant entre chaque exécution à l'état source. Il faut donc exécuter une sauvegarde de l'état du système mais également l'état dans lequel se trouve l'automate de Büchi au bon endroit avant et après l'exécution d'une transition et les créations de nouveaux états.

Partie 3

Chapitre 7

L'EXPLOSION COMBINATOIRE DE L'ESPACE D'ÉTAT ET LA VERIFICATION RANDOMISÉE

CHAPITRE 7
L'EXPLOSION COMBINATOIRE ET LA VERIFICATION RANDOMISEE

Sommaire

Les algorithmes de parcours en profondeur présenté précédemment permet la construction explicite du modèle et donc de l'ensemble de ses états. De plus, dans le cas des propriétés de vivacité, nous travaillons sur le produit cartésien du modèle du système et de l'automate de Büchi de la négation de la propriété. Enfin, l'indéterminisme éventuel de l'automate mais également du système impliquent rapidement une augmentation exponentielle du nombre d'états et donc des ressources mémoires utilisées (n transitions exécutables ⇒ n! ordres d'exécution possibles et 2^n états). On parle alors d'explosion combinatoire [48] de l'espace d'état.
Depuis plusieurs années, ce problème a donné lieu à de nombreux travaux présentant plusieurs techniques de réduction ou de gestion en exploitant différentes informations [49], parmi lesquelles la réduction par ordre-partiel retenue pour le model checker de SimGrid.

On va commencer ce dernier chapitre par un état de l'art des méthodes de limitation de l'explosion combinatoire de l'espace d'état (avec plus de détails pour la méthode de Réduction dynamique par ordre partiel (DPOR) adoptée par SimGrid). Ces approches présentent toute des inconvénients ; celles qui gardent en mémoire des valeurs compressées des états au lieu de toute la description de l'état, et celles qui ne considèrent qu'une partie de l'espace d'état. Le gain en espace de stockage se fait au détriment d'un risque d'omission de certains états dans l'exploration.

Contribution
Nous introduisons par la suite la méthode de vérification randomisée, inspirée de la méthode de la marche aléatoire (RW).
Nous proposons un schéma générique des algorithmes d'exploration et nous précisons ses différentes fonctions et paramètres. Nous proposons aussi un algorithme d'exploration probabiliste plus adaptés qui réduit considérablement la redondance et améliore la couverture et l'atteignabilité des états, ainsi que les critères d'évaluation de cet algorithme

Limitation de l'explosion combinatoire de l'espace d'état

1. Approches possibles

1.1 Réduction du nombre d'états

Pour réduire le nombre d'états à évaluer pendant la vérification, il est possible d'exécuter une réduction basée sur les informations de l'état, notamment en déterminant les états identiques et donc en n'évaluant qu'un seul d'entre eux. Cette réduction peut être faite de façon statique dans le cas où l'on connait l'ensemble des états du modèle avant la vérification ou bien de manière dynamique, au fur et à mesure que l'on construit le modèle. La réduction par symétrie [38] ou la canonicalisation [50] sont des exemples de ce type de réduction. Dans le cas de SimGrid, une simple comparaison des hash des représentations complètes des états a été implémentée pour la vérification des propriétés de vivacité. Cependant, la probabilité d'obtenir des états identiques, i.e., avec le même hash, durant une exécution est trop faible pour rendre cette approche suffisante à elle seule pour pallier au problème de l'explosion combinatoire de l'espace d'état. Il faut donc envisager l'implémentation d'autres techniques d'identification des états identiques ou bien ajouter une autre approche de réduction.

Il est également possible d'exploiter les transitions pour réduire le nombre d'états, en utilisant les principes d'indépendance et d'invisibilité des transitions. Ceci permet d'éliminer plusieurs états à la fois en éliminant des parties des chemins d'exécutions contenant des transitions "inutiles" à la vérification de la propriété. La réduction par ordre partiel [51] [52], le regroupement de transitions [53] ou l'analyse simultanée d'accessibilité [56] sont des exemples de techniques de réduction sur les transitions. Actuellement, le model checker de SimGrid implémente une technique de réduction par ordre partiel [54] [55] avec l'évaluation des transitions dépendantes pour la vérification de propriétés de sureté. Nous avons donc choisi de poursuivre dans l'étude de cette approche pour la vérification des propriétés de vivacité en intégrant le principe d'invisibilité des transitions, présenté dans la Section 2.2.

1.2 Réduction de l'espace mémoire de chaque état

En négligeant dans une certaine mesure le temps de vérification, il est possible d'utiliser des techniques de réduction des ressources mémoires utilisées par le model checker permettant de travailler sur des espaces d'état plus grands. Pour cela, on peut utiliser la compression des représentations des états sauvegardées [57] [58].

Au lieu de sauvegarder tous les états visités avec leur représentation, on peut aussi exécuter un compromis entre la réduction des ressources mémoire et le temps de vérification en ne conservant que certains des états visités. La sélection de ces états peut être faite selon certaines heuristiques [59] ou bien, il est possible d'effacer des sauvegardes des états visités lorsque les ressources mémoires deviennent trop faibles [60]. Ainsi certains états déjà visités pourront l'être plusieurs fois mais en comparaison avec la mise en place d'un système de compression/décompression des sauvegardes, ceci peut s'avérer plus rapide tout en étant moins exigeant en ressources mémoire.

1.3 Utilisation d'environnements parallèles ou distribués

Une autre approche pour gérer le nombre d'états est d'utiliser simplement le calcul distribué ou parallèle avec plusieurs processeurs [61]. Pour cela on peut effectuer le model checking à travers plusieurs machines réseau. L'espace d'état est alors divisé en plusieurs parties réparties sur les différentes machines qui échangent des messages pour informer les autres machines de l'espace d'état. Toutefois, dans le cas de la vérification de propriétés de vivacité, cette approche est difficile à implémenter avec un algorithme de parcours en profondeur inadapté aux environnements distribués. Il faut donc dans ce cas utiliser d'autres algorithmes de vérification. [62][63]

On peut également utiliser plus simplement des machines multiprocesseurs avec mémoire partagée [64]. Cependant, dans le cas de SimGrid, le model chocking s'effectue directement sur l'ordinateur de l'utilisateur. Donc il n'est pas possible de s'appuyer sur cette unique approche si SimGrid est exécuté sur une machine uni-processeur.

1.4 Vérification aléatoire partielle et heuristiques

L'ultime approche pour la réduction de l'espace d'état est l'utilisation de techniques aléatoires et d'heuristiques [65] [66]. Ces techniques ne parcourent qu'une partie de l'espace d'état. Elles sont donc adaptées à la détection d'erreur mais pas la preuve de justesse. Par exemple, on peut visiter uniquement les états correspondants à certaines heuristiques, ou bien choisir, de ne parcourir que certains chemins d'exécutions pris aléatoirement.

2. Réduction dynamique par ordre partiel (DPOR)

Selon la modélisation et le type de système que l'on vérifie, la réduction de l'espace d'état peut être faite statiquement ou dynamiquement. Une réduction statique signifie que l'on réduit l'espace d'état avant la vérification. Ceci induit donc de connaitre l'ensemble des états du modèle dés le début. On peut alors appliquer une abstraction d'état ou une abstraction de comportement. Cette approche est la plus implémentée dans la communauté [67], notamment dans SPIN qui exploite une modélisation en langage PROMELA. Cependant, ceci est plus complexe pour les programmes C.

Inversement, une réduction dynamique signifie que l'on réduit l'espace d'état pendant la vérification, pendant que l'on construit l'espace d'état. Pour cela, on peut travailler sur une représentation efficace de l'espace d'état en déterminant dynamiquement à chaque état un sous ensemble de transition basé sur les informations des transitions, en particulier l'indépendance et l'invisibilité, et les états déjà visités.

Parmi les approches possibles présentées précédemment, dans le cas de SimGrid, plusieurs approches de réduction dynamique du nombre d'états sont adoptées. Selon la réduction effectuée, on peut travailler uniquement sur les états ou les transitions, ou bien, coupler les deux pour obtenir une réduction plus juste.

Réduction		**Sûreté**	**Vivacité**
∅		DFS	Double-DFS
Sur les transitions	Indépendance	DPOR de SimGrid	x
	Indépendance + Invisibilité	Ample set (plus d'états)	Ample set (cycle gratuit)
Sur les états	Comparaison de hash d'états	Conservation du hash des états visités	
	Symétries applicatives	Techniquement difficile	
	Canonicalisation		
Sur les états et les transitions	Indépendance + Invisibilité + Étiquetage des états	Ample set + état sain	Ample set + coloration des états

Sur le tableau ci-dessus, nous présentons les différentes approches possibles pour effectuer une réduction dynamique de l'espace d'état dans SimGrid, en exploitant la notion d'ordre-partiel ou l'identification des états identiques.

Comme expliqué précédemment, une première approche sur la réduction du nombre d'état en conservant les états déjà visités, a été implémentée grâce au stateful model checking.

Toutefois, face à une consommation des ressources mémoires trop importante sans optimisation, il est plus intéressant de s'orienter vers une autre approche de réduction du nombre d'état, déjà utilisée dans SimGrid, basée sur les transitions et non plus les états.

Actuellement le model checker de SimGrid applique une réduction dynamique par ordre-partiel pour la vérification des propriétés de sûreté [55]. Dans le cas d'une exploration explicite de l'espace d'état, la réduction par ordre partiel est une technique d'expansion d'un échantillon représentatif de toutes les transitions exécutables par un état. Ainsi, au lieu de vérifier l'ensemble des successeurs possibles d'un état, on ne conserve qu'un sous-ensemble *Ample(s)*. Dans le cas des propriétés de sûreté, cette réduction est basée sur la commutativité des transitions. Pour la vérification des propriétés de vivacité, nous verrons pourquoi cette unique condition d'indépendance n'est pas toujours correcte pour la réduction et pourquoi il faut ajouter la notion d'invisibilité (Section 2.2) à la sélection du sous-ensemble des transitions à exécuter pour chaque état.

2.1 Indépendance

Définition (Indépendance des transitions). Soit T, l'ensemble des transitions et S, l'ensemble des états. Soit *Enabled(s)* $\subseteq$ T, l'ensemble des transitions exécutables à l'état s $\in$ S.
Une relation d'indépendance I $\subseteq$ T $\times$ T est une relation symétrique et antiréflexive telle que, $\forall$s $\in$ S:

- si (α,β) $\in$ *Enabled(s)*, alors $\alpha \in$ Enabled(β(s)) et
- si (α,β) $\in$ *Enabled(s)*, alors α (β(s)) = β(α(s))

Si (α, β) $\in$ I, alors α et β sont indépendantes.

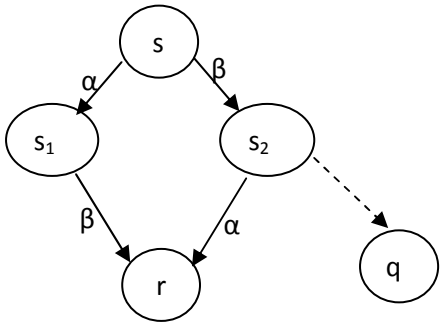


Figure 9- Transition α et β indépendantes

La première condition dans la relation d'indépendance signifie que l'exécution de l'une des transitions ne doit pas rendre l'autre transition non exécutable. La seconde condition signifie que l'on exécute les transitions dans un ordre ou dans l'autre, cela doit mener au même état.

Dans le cas de SimGrid, l'approche est dynamique, en découvrant au fur et à mesure l'espace d'état, donc cette élimination ne peut pas être réalisée ainsi. L'approche implémentée est une adaptation de réduction dynamique par ordre partiel proposée par Godefroid [54], appliquée à la vérification de propriétés de sûreté formulées en assertions et à la détection de deadlock. Le principe est de lancer une exécution sur un chemin aléatoire, puis, lorsqu'on arrive à la fin de ce chemin, on analyse les transitions dépendantes/ indépendantes en remontant état par état le long du chemin. Chaque transition correspond à un processus qui exécute une action et chaque processus s'exécute de façon séquentielle. Pour un état donné, il peut donc y avoir plusieurs processus pouvant exécuter une unique action chacun. On commence donc avec l'avant dernier état s_0 du chemin et on regarde si la transition exécutée par un état s précédent s' est dépendante avec la transition exécutée par s'. Si on rencontre deux transitions dépendantes, i.e., l'exécution d'un processus p_1 pour un état s a mené à l'exécution d'un autre processus p_2 pour l'état s_0 suivant s, on intègre l'exécution du processus p_2 dans s ce qui mènera obligatoirement vers un autre résultat d'exécution. Si l'état s_0 n'a pas lui-même d'autres transitions intégrées par analyse de dépendance avec l'exécution d'un autre état suivant ce dernier, on supprime alors cet état et on exécute une nouvelle analyse de dépendance sur l'état précédent s_0 (l'avant-avant-dernier du chemin). A l'inverse, si cet état possède d'autres transitions intégrées et donc exécutables, on arrête l'analyse de dépendance en effectuant un backtracking jusqu'à celui-ci et on exécute alors l'une de ses transitions exécutables pour avancer vers un nouveau résultat d'exécution et le vérifier. Donc à chaque fin de chemin d'exécution, on regarde les transitions dépendantes appartenant à celui-ci et on reporte selon les résultats certaines transitions vers d'autres états. Quand la vérification est terminée et qu'il n'y pas plus d'états dans la pile à étudier, tous les deadlocks et les assertions fausses ont été détectées avec garantie.

Grâce à cette relation, certaines branches d'exécution peuvent être volontairement omises pour la vérification, sachant que leur comportement est "équivalent" et mène au même état dans l'exécution du système. Sous certaines conditions, cette relation est suffisante pour réduire correctement l'espace d'état, en particulier, en l'absence de boucle dans l'exécution du système. Cependant, on peut observer plusieurs cas où l'élimination d'une branche peut mener à un résultat incorrect, en particulier si l'on travaille avec des propriétés LTL.

En effet, dans le cas d'une vérification d'une formule LTL, quand on exécute une transition, on se place du côté de l'automate de Büchi de la propriété pour regarder l'impact sur son évolution, en particulier si l'on peut avancer vers un autre état de l'automate. Ainsi, les états intermédiaires s_1 et s_2 obtenus selon l'ordre d'exécution des transitions dites indépendantes peuvent influencer l'évolution dans la vérification de la propriété. La propriété peut donc être sensible à l'ordre d'exécution des transitions même si celles-ci mènent à un état identique dans l'évolution du système. De même, s_1 et s_2 peuvent avoir des successeurs q en plus de r qui peuvent être éliminés si on ne conserve qu'une branche d'exécution, alors qu'ils n'ont pas été vérifies.

Enfin, ceci peut mener à un report constant de certaines transitions s_i bien que certaines ne seront peut-être jamais vérifiées avant la fin de l'exécution du système, alors qu'elles peuvent être significatives par rapport à la propriété. Il faut donc intégrer de nouvelles

propriétés sur les transitions afin de vérifier tous les états et transitions pertinents par rapport à la propriété tout en réduisant l'espace d'état.

2.2 Invisibilité

Définition (Transition invisible)

Soit S l'ensemble des états, T l'ensemble des transitions et AP l'ensemble des propositions atomiques de la propriété LTL.
Soit L : S → 2AP la fonction qui étiquette chaque état avec un sous-ensemble AP' ⊆ AP, correspondant aux propositions atomiques vraies dans cet état.

Une transition α ϵ T est invisible selon AP' ⊆ AP si, ∀ (s; s') ϵ S, tel que s' = α(s), L(s) ⋂ AP' = L(s') ⋂ AP'.
Donc, une transition est invisible si son exécution, à partir de n'importe quel état du système, ne modifie pas la valeur des propositions atomiques dans AP'.

2.3 Ample set

2.3.1 Conditions

A partir de ces deux caractérisations des transitions, il est possible ensuite de calculer le sous-ensemble de transitions Ample(s) de chaque état, dans le cas de vérification de propriétés de vivacité formulées en LTL. Plus particulièrement, nous nous sommes intéressés à la vérification de formules LTL_{-X}, i.e., des formules LTL sans la modalité **X.**

On ne s'intéresse qu'à cette catégorie de propriétés car elles sont invariantes en cas de bégaiements dans l'exécution, ce qui rend la commutativité (indépendance) et l'invisibilité suffisantes pour déterminer correctement un sous ensemble de transitions à évaluer à chaque état.

Par exemple, la formule LTL ϕ = a **U** b est invariante sous bégaiement car dans n'importe quelle séquence d'exécution qui la vérifie, comme (a¬b, a¬b,¬ab, ab, ab, ab; …), on peut répéter n'importe quels états et obtenir une séquence, comme (a¬b, a¬b, a¬b,¬ab,¬ab, ab, ab, ab, …), qui vérifiera toujours la propriété. A l'inverse, la formule LTL ψ= a Λ **X**b n'est pas invariante sous bégaiement car elle vérifie la séquence d'états (a¬b, ¬ab, ab,…) mais pas la séquence (a¬b, a¬b, ¬ab, ab,…).

En appliquant les propriétés d'indépendance et d'invisibilité des transitions, on peut donc déterminer le sous-ensemble de transitions Ample(s) ⊆ Enabled(s) pour chaque état s, sous les conditions suivantes [46] :

C0 : Enabled(s) =∅ , Ample(s) = ∅
C1 : pour tout chemin d'exécution du modèle sans réduction, une transition α qui est dépendante avec une transition β ϵAmple(s) ne peut pas être exécutée avant une transition γ ϵ Ample(s).
C2 : si Enabled(s) ≠ Ample(s), alors, ∀ α ϵ Ample(s), α est invisible
C3 : un cycle n'est pas acceptable s'il contient un état pour lequel une transition α est activée mais jamais incluse dans Ample(s) pour tout état s appartenant au cycle

La première condition (**C0**) indique simplement que si un état a au moins un successeur sans réduction, il doit en avoir un même avec la réduction. Ceci permet de garantir que,

même si une réduction totale est possible sur un état, une transition sera tout de même conservée pour poursuivre le model checking.

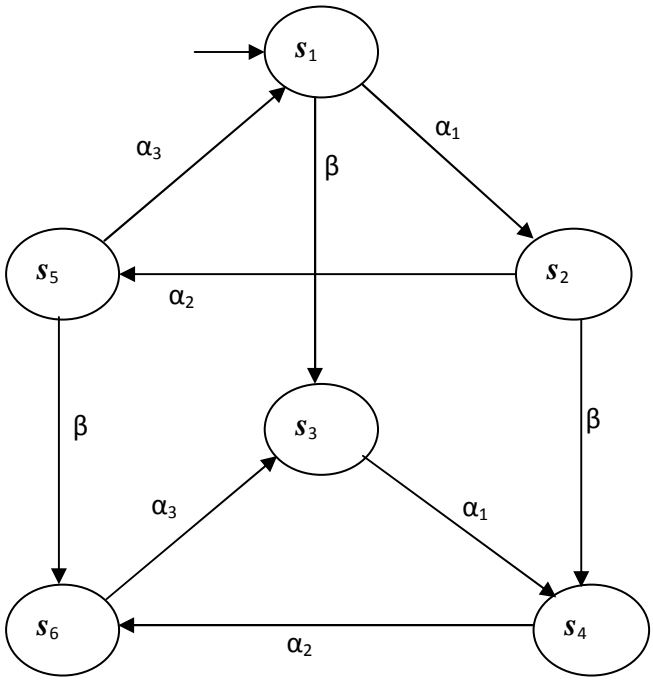
La seconde condition (**C1**) ne s'applique que sur un chemin complet d'exécution et surtout sur l'ensemble du système. Sachant que l'on travaille dynamiquement et qu'on construit au fur et à mesure l'espace d'état tout en le réduisant, il faut transformer cette condition pour calculer Ample(s) en se basant uniquement sur un état s et non sur tout un chemin d'exécution. Cette condition peut alors être reformulée :

C1': si Enabled(s) ≠ Ample(s), alors, ∀α ∈ Enabled(s)\Ample(s) et ∀β ∈ Ample(s), α et β sont indépendantes.

Les conditions **C1** et **C2** ne sont cependant pas suffisantes pour garantir que l'espace d'état réduit est équivalent sous bégaiement à l'espace d'état complet, et donc que la vérification sur l'espace d'état réduit est toute aussi correcte que sur l'espace d'état complet.

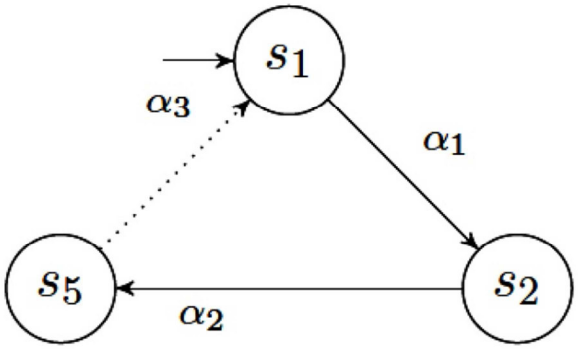
En effet, il est possible que certaines transitions soient reportées sin die, et donc finalement définitivement supprimées, en présence d'un cycle dans le système, alors qu'elles interviennent dans l'évolution de la vérification de la propriété au niveau de l'automate de Büchi.

Exemple. Soit l'espace d'état suivant :



On suppose que β est indépendante des transitions α_1, α_2 et α_3 et que α_1, α_2 et α_3 sont interdépendantes. On suppose qu'il existe une proposition atomique p, qui passe de vraie à fausse en exécutant β; donc β est visible.

Si l'on part de l'état s_1, on peut sélectionner Ample(s_1) = {α_1} et donc reporter l'exécution de β vers un état suivant, ce qui respecte les conditions C1 et C2. On génère alors l'état $s_2 = \alpha_1(s_1)$. Ensuite, on sélectionne Ample(s_2) = {α_2}, en reportant à nouveau l'exécution de β vers un nouvel état suivant, et on génère l'état $s_5 = \alpha_2(s_2)$. A l'état s_5, on sélectionne Ample(s_5) = {α_3} ce qui crée un cycle (s_1; s_2; s_5) et β est définitivement ignorée.



Bien que respectant parfaitement les conditions C0, C1 et C2 à chaque étape, la réduction obtenue ne contient aucune séquence d'états ou *p* change de valeur alors qu'il existait initialement une transition β qui le faisait. Le résultat obtenu sur la réduction sera donc différent de celui obtenu sur l'espace d'état initial et donc potentiellement incorrect.
La quatrième condition (C3) permet donc de prévenir ce type de situation. Cependant, elle ne peut pas être vérifiée pendant une réduction dynamique. On peut donc la reformuler ainsi, pour la rendre applicable à une vérification dynamique :

C3' : tout cycle doit contenir un état s tel que Ample(s) = Enabled(s)

Ces conditions permettent donc d'éviter une réduction trop restrictive pour les propriétés de vivacité. Dans le cas d'une application aux propriétés de sûreté, ceci implique une réduction moins forte qu'avec la simple utilisation de dépendance entre transitions. Il sera donc nécessaire d'adapter la réduction au type de propriété que l'on veut vérifier. L'idéal serait bien sûr de pouvoir déterminer automatiquement la réduction la plus adaptée selon le système que l'on veut vérifier et la propriété à vérifier.

2.3.2 Calcul dynamique

A partir de ces quatre conditions, on peut donc sélectionner un sous-ensemble de transitions à vérifié pour chaque état tout en respectant la pertinence de la propriété. Cependant, le calcul de ce sous-ensemble, appliqué à la majorité des model checker existants, nécessite en réalité une analyse supplémentaire à chaque fois, notamment pour vérifier la troisième condition(C2). En effet, cette condition fait intervenir la notion d'invisibilité sur les transitions, qui caractérise une transition comme invisible, si son exécution, à partir de n'importe quel état, ne modifie pas la valeur des propositions atomiques de la propriété LTL. Il faut donc, soit connaitre l'ensemble des états, ce qui n'est pas supposé être possible dans le cas d'une réduction dynamique, puisqu'on crée les états au fur et à mesure de la vérification, soit effectuer une analyse spéciale sur la transition et ses informations pour pouvoir déterminer si elle est invisible, quelque soit l'état à partir duquel elle est exécutée et sans connaitre l'ensemble des états. Dans le cas de SimGrid, ceci est d'autant plus difficile que l'on ne peut connaitre l'action d'une transition qu'en l'exécutant. Une analyse spécifique sur les transitions est donc difficilement envisageable.

Une première piste envisagée consiste à exploiter les informations collectées pendant la vérification d'un premier chemin d'exécution aléatoire. D'après la définition d'invisibilité d'une transition, il faut que son exécution ne modifie pas les valeurs des propositions atomiques pour tous les états à partir desquels elle est exécutée. Donc s'il existe un état pour lequel la transition modifie ces valeurs, on peut immédiatement affirmer que la transition est visible. On pourra alors avoir une première liste des transitions visibles sur l'ensemble de l'exécution. De plus, lors de la première exécution, même en choisissant aléatoirement la transition à exécuter à chaque état s parmi les transitions exécutables Enabled(s), on commence la sélection du sous-ensemble de transitions Ample(s) pour chaque état s. Or, si pour cette première exécution, certaines transitions sont visibles, la condition C2 n'est plus respectée pour Ample(s). Dés lors, on peut transformer la condition de façon bien plus restrictive mais parfaitement applicable à une réduction dynamique respectant la propriété, telle que :

C2' : Si $\exists \alpha \in$ Enabled(s), tel que α est visible, alors Ample(s) = Enabled(s)

Critiques

Cependant, cette approche est un peu trop restrictive pour être assez efficace dans la réduction de l'espace d'état. Si le premier chemin d'exécution aléatoire choisi ne contient que des transitions visibles, aucune réduction ne sera faite. De plus, ceci gère les transitions dont on peut affirmer dès la première exécution qu'elles sont visibles mais nous ne pouvons toujours pas déterminer correctement si les autres peuvent être reportées par dépendance et surtout si l'on peut éliminer immédiatement certains états.

3. La vérification randomisée

Dans la classe des méthodes randomisées, la vérification est assurée par des algorithmes d'exploration aléatoires et semi aléatoires permettant de réaliser une bonne exploration de l'espace d'états du système. Une bonne exploration signifie l'atteinte et la vérification d'un grand nombre d'états du graphe sans dépasser les ressources mémoire disponibles. Les algorithmes randomisés ont vite pris place au monde informatique vu leur simplicité et rapidité. Ils ont pu dépasser les algorithmes déterministes dans divers problèmes de recherche [73]. En effet, si le graphe d'états du système à vérifier est très grand, l'algorithme déterministe l'explore d'une façon ordonnée et garde les états explorés en mémoire. Dès épuisement de celle-ci, soit il abandonne l'exploration, soit il reste bloquée dans un problème de swapping. Il échoue à explorer tous les états du système et en particulier à atteindre les états situés, selon l'ordre qu'il respecte, au delà de la taille mémoire disponible. Un algorithme randomisé donne à ces états une chance plus importante d'être atteints et vérifiés via une exploration mieux diffusée, c'est à dire qui ne respecte pas un ordre particulier et donne aux états une probabilité non nulle d'être atteint. Un tel algorithme utilise la mémoire disponible, et vu l'insuffisance de celle-ci, il n'y garde qu'un sous ensemble des états explorés. D'où, la couverture totale de l'espace d'états n'est pas garantie. Néanmoins, au lieu d'abandonner l'exploration par manque de ressources et ne retourner aucune réponse quant à la satisfiabilité de la propriété en question par le système, le résultat de la vérification est donné approximativement, avec une probabilité que l'on peut contrôler.

3.1 La marche aléatoire (Random Walk)

Une marche aléatoire sur un graphe G est un cas particulier du processus stochastique appelé "chaîne de Markov à temps discret", avec un espace d'états M des transitions de probabilité homogènes et uniformes. L'algorithme démarre de l'état initial du graphe, et à chaque étape, il choisit aléatoirement selon une distribution (de transition) uniforme un successeur de l'état courant et le visite. Ce choix est indépendant du chemin précédemment parcouru, ce qui est caractéristique d'une chaîne de Markov.
L'algorithme se termine lorsqu'il n'y a plus de possibilité de choisir un successeur (deadlock) ou, dans le cas d'exploration en boucles, lorsqu'on atteint un nombre maximal d'étapes fixé par l'utilisateur.

Définition 3.1.1 Une marche de longueur n sur un graphe G est une suite de états $v_0, v_1, ..., v_n$ tel que $v_{i+1} \in Succ(v_i)$ pour i = 0, ..., n − 1. La marche est dite aléatoire ssi chaque v_{i+1} est tiré aléatoirement et uniformément parmi les successeurs de v_i

Du point de vue théorique, une caractéristique très importante de la marche aléatoire, et de tout algorithme d'exploration A appliqué au graphe G, est *le temps de couverture*.

Définition 3.1.2 Le temps de couverture d'un algorithme A sur un graphe G est défini comme étant le nombre moyen d'étapes nécessaires à l'algorithme A pour visiter tous les états de G

Inconvénient de la marche aléatoire

Rappelons que la marche aléatoire (RW) n'utilise aucune mémoire. A chaque étape, elle ne garde qu'un seul état et ne préserve aucune information sur les états visités précédemment. On remarque dans ce cas que la probabilité d'exploration des différents états du graphe est

loin d'être équilibrée, du fait que certains états sont beaucoup plus souvent visités que d'autres, pour diverses raisons :

- ✓ Si le graphe contient beaucoup de points morts ou beaucoup de retours arrière alors les états de petite profondeur sont plus souvent visités.
- ✓ Si la marche aléatoire atteint une composante fermée, elle continue à visiter seulement les éléments de cette composante.

3.2 Randomisation de la vérification

La majorité des méthodes randomisées de vérification utilisent la marche aléatoire (random walk : RW), comme schéma d'exploration. L'avantage principal de RW est qu'il ne nécessite, quasiment, aucune ressource mémoire puisqu'il ne garde que l'état courant au cours de l'exploration. Son inconvénient majeur est l'exploration redondante des mêmes états vu l'absence de toute information sur la partie déjà explorée. Pour l'optimiser, différentes approches proposent de le paralléliser, de le combiner avec des méthodes exhaustives d'exploration locale [71], ou de le doter de ressources mémoire qui lui permettent de garder une partie des états visités. En outre, lorsque des informations supplémentaires sur le système exploré sont disponibles, certaines améliorations visent à guider le RW pour arriver plus rapidement à détecter les erreurs du système.
L'objectif est de montrer que l'approche d'exploration randomisée peut apporter des performances importantes par rapport aux algorithmes déterministes d'exploration : elle permet d'augmenter de manière significative le nombre d'états explorés et d'améliorer leur atteignabilité

3.3 Algorithme proposé

3.3.1 Schéma général des algorithmes d'exploration

Un algorithme d'exploration quelconque peut être présenté par le schéma générique de la figure 10. Dans ce schéma, *P* représente les paramètres d'entrée de l'algorithme, par exemple, la propriété à vérifier ϕ, le nombre des exécutions parallèles initiales dans le cas d'une marche aléatoire parallèle, etc. Ce dernier paramètre, entre autre, peut être modifié durant l'exécution de l'algorithme selon les ressources disponibles et les besoins de l'exploration. *I* contient des informations globales sur la structure du graphe G, par exemple, le nombre moyen de successeurs des états, le nombre moyen de boucles, de deadlock,... etc. Notons que ce type d'informations est généralement collecté à la volée et utilisé pour guider et optimiser l'exploration.

//Structures de données :

P : **Paramètres de l'algorithme (statique) ;**
I : **Informations sur le graphe (dynamique) ;**
V : **Ensemble des états gradés en mémoire ;**
v : **état courant;**

//Initialisation :

$V \leftarrow \{v_0\}$
$v \leftarrow \{v_0\}$
// Corps de l'algorithme :

Tant que (***non*** condition d'arrêt) **faire**
v ← sélectionner (*V.P.I*) ;
visiter(v) ;
(V.I)← actualisé (V.*v*.P.I);
fait

Figure10 *Schéma général des algorithmes d'exploration*

Un algorithme respectant cette forme générale, peut être vu comme une marche aléatoire simple dans un graphe de dimension N, où N est la taille de la mémoire.
Une exécution de l'algorithme randomisé A sur le graphe original G correspond à une marche aléatoire simple sur le graphe, de dimension N, G^N défini comme suit : Un état de G^N est un N-uplet $V = (v_0, v_1, ..., v_N)$, où chaque v_i est un état de G stocké en mémoire ou un emplacement vide noté 0. Il y a une transition $V = (v_0, v_1, ..., v_N) \rightarrow V' = (v'_0, v'_1, ..., v'_N)$ dans le graphe G^N ssi $\exists$ j, $1 \leq j \leq N$, tel que $v'_j \in Succ(v_j)$ et $v_i = v'_i$, $\forall i \neq j$, $1 \leq i \leq N$. Pour N = 1, on se retrouve dans le cas d'une marche aléatoire simple dans le graphe G. Sinon, l'algorithme consiste en une marche aléatoire multidimensionnelle.
Selon le schéma précédent, un algorithme d'exploration est complètement défini en spécifiant *la condition d'arrêt* et les deux *fonctions sélectionner* et *actualiser*. Avec ces trois fonctions, on peut définir plusieurs variantes, incluant les algorithmes déterministes et probabilistes décrits dans la littérature, ainsi que ceux que nous allons proposer.

3.3.2 Une sélection randomisée

En se basant sur le schéma général présenté précédemment, plusieurs algorithmes d'exploration randomisés peuvent être proposés en combinant les méthodes de sélection randomisées aux stratégies de remplacement. L'algorithme que nous proposant est conçu en jouant sur la sélection seulement et non pas sur l'actualisation dans la mémoire des états stockés. Ci-dessous, nous énumérons sommairement quelques méthodes possibles pour une sélection randomisée :

1. Choisir uniformément un fils du dernier état visité et s'arrêter si l'état courant n'a plus de successeurs (deadlock), ce qui correspond à l'algorithme RW.
2. Choisir uniformément un état dans le sous ensemble V des états visités, puis choisir uniformément l'un de ses successeurs. Ce choix permet de diffuser l'exploration dans toutes les directions sans favoriser l'exploration en profondeur comme c'est le cas des algorithmes basés sur RW. Il donne à tous les états visités une même probabilité d'avoir un successeur exploré à l'étape suivante.

3. Une feuille f est un état n'ayant aucun successeurs explorés $Succ(f) \cap V = \emptyset$. Un état interne *i* est un état ayant des successeurs déjà explorés $Succ(f) \cap V \neq \emptyset$. Choisir aléatoirement un fils d'une feuille ou d'un état interne. La décision entre feuille et état interne s'effectue selon une probabilité prédéfinie donnée en paramètre. Cette méthode de sélection permet d'orienter l'exploration en profondeur ou en largeur selon le taux désiré.

4. Choisir aléatoirement un fils de l'état *n* de *V* qui a le plus de fils non visités ($Succ(n) \setminus V = Max_{v \in V} Succ(v) \setminus V$) ou qui a le moins de fils visités ($Succ(n) \cap V = Min_{v \in V}\ Succ(v) \cap V$). Ces méthodes de sélection utilisent les informations disponibles pour mieux distribuer l'exploration sur les états du graphe.

5. Choisir aléatoirement un petit fils d'un état visité, sélectionné selon l'une des façons décrite ci-dessus. Il s'agit d'appliquer la fonction $succ()$ plusieurs fois consécutives et retenir dans *V* l'état résultant sans retenir ses prédécesseurs, ce qui permet d'atteindre de grandes profondeurs dans les graphes explorés.

6. Garder à chaque instant deux états courants ou plus, et choisir les successeurs selon l'une des méthodes précédentes, ce qui amène à des "Algorithmes pseudo parallèles".

3.3.3 Stratégie de replacement

La fonction d'actualisation se contentera d'ajouter chaque état, nouvellement exploré, à l'ensemble *V* jusqu'à épuisement de l'espace de stockage. Dans ce cas, la mémoire est vidée et l'algorithme est relancé à partir de l'état initial. Ceci est répété un certain nombre *R* de fois. Il s'agit d'une stratégie de remplacement aussi simple et efficace

Modélisation

Un algorithme d'exploration qui entre dans le schéma général est défini principalement par trois paramètres : une condition d'arrêt de l'exploration, une fonction de sélection de l'état suivant à explorer, à partir d'un état ou d'un ensemble d'états déjà explorés et stockés en mémoire, et une fonction d'actualisation qui définie la stratégie de remplacement des états gardés en mémoire.

Contraintes :

On cherche à maximiser la portion du graphe explorée et à minimiser le temps consommé dans cette exploration, sous la contrainte de l'espace mémoire limité.

Nous avons, donc, considéré dans notre modélisation plusieurs critères pour évaluer une exploration, notamment le nombre moyen d'états couverts et le temps de couverture moyen, ainsi que la probabilité d'atteignabilité. En ce qui concerne la taille, limitée, de la mémoire utilisée, elle est incluse explicitement comme paramètre principal dans notre modèle général. Ce paramètre permet de prendre en considération la contrainte posée par les ressources mémoire limitées et de contrôler à chaque instant l'état de la mémoire, ce qui garantit qu'elle ne soit pas saturée et évite le problème de swapping.

3.3.4 Algorithme proposé : Depth Oriented Random Search

Conformément au schéma général présenté précédemment, cet algorithme explore l'espace d'états et garde les états visités en mémoire jusqu'à l'épuisement de celle-ci. Notons que le fait que DORS soit doté de mémoire, contrairement au RW, lui permet de distinguer entre les états visités et non visités et évite beaucoup d'explorations redondantes

La fonction de sélection choisie, à chaque étape, uniformément, un successeur du dernier état visité v. Cependant, l'exploration ainsi conduite risque de se bloquer dans deux cas : en arrivant à une feuille (succ(v) = ∞) ou dans une boucle. Notons par *D* l'ensemble des états feuilles visités et par W l'ensemble d'exploration courante qui consiste en l'ensemble des états visités depuis le dernier blocage (dernière atteinte d'une feuille ou détection d'une boucle) jusqu'à l'instant courant. Dans le cas où succ(v) ≠ ∞, mais que le successeur sélectionné de v se retrouve dans W, une boucle, dite engendrée par v, est alors détectée. Elle est en fait contenue dans W.

Dans le cas de blocage (v est une feuille ou v engendre une boucle), l'exploration est dite être en un point fermé, autrement elle est dite en un point ouvert. En arrivant à un point fermé, l'exploration change de comportement : le état courant *v* est resélectionné uniformément dans *V* \ *D* et l'exploration se poursuit en tirant aléatoirement, et uniformément, un état de succ(v).

Il convient de noter le double avantage du choix de v parmi *V**D*, dans le cas de blocage. D'une part, ceci permet d'éviter, l'inconvénient du RW qui, après chaque blocage, se trouve réinitialisé à partir de l'état initial, entraînant ainsi une exploration redondante des états de petite profondeur. D'autre part, il donne plus de chance à une exploration encore plus profonde.

```
V : ensemble des états en mémoire ;
W, D : ensembles des états ;
N : taille maximale de V ;
v : état courant ;
i : entier ;


V ← {v0} ;
W, D ← { } ;
v ← v0 ;
i ← 0 ;

Tant que (i ≤ N) faire
        Si (Succ(v) = Ø ou v ∈ W) Alors
                Si (Succ(v) = Ø) Alors
                        D ← D U {v} ;
                Fin Si
                v ← choisir uniformément un état dans V ;
                W ← {v} ;
        Sinon
                v ← choisir uniformément un état dans Succ(v) ;
                Si (v ∉ V) Alors
                        Vérifier (v) ;
                        V ← V U {v} ;
                        W ← W U {v} ;
                        i ← i+1 ;
                Fin Si
        Fin Si
Fait
```

Figure 11 *Algorithme randomisé orienté profondeur*

Bien que notre algorithme possède une fonction de sélection semblable à celle de DORS, proposé dans [74], il diffère de celui-ci dans plusieurs aspects. DORS est un algorithme randomisé du type Las Vegas. Il ne garde en mémoire que les états "non clos" (i.e. ceux qui ont au moins un successeur non visité). Le cas de saturation de la mémoire n'est pas étudié et la répétition de DORS n'est pas considérée. Notre algorithme est ressource-dépendant, incluant explicitement la taille de la mémoire comme paramètre contrairement à DORS.

Critères d'évaluation

En général, les performances des algorithmes d'exploration destinés à la vérification se mesurent par deux critères principaux : la couverture et l'atteignabilité.

a. La couverture

Ce critère exprime la capacité de l'algorithme à explorer l'espace d'états. Une bonne couverture reflète moins d'explorations redondantes. Si l'algorithme *A* arrive à couvrir *k* état distincts dans le graphe considéré *G*, en *n* étapes de son exécution, alors le taux de redondance est donnée par :

$$Tr = \frac{n-k}{n}$$

En effet, un algorithme d'exploration, à chaque étape de son exécution, peut soit visiter un, et un seul, nouvel état, soit répéter la visite d'un état déjà visité. Dans le premier cas, et le temps *n* et le nombre d'état couverts *k* sont incrémentés de un. Dans le deuxième cas, le temps est incrémenté mais non pas le nombre d'état couverts, ce qui augmente la redondance.

Notre étude de la couverture peut se faire de plusieurs façons :

- ✓ *Le temps moyen de couverture* $T_{A,G}(k)$: c'est le nombre moyen d'étapes *n* nécessaires à un algorithme donné *A* qui commence de l'état initial pour couvrir un nombre *k* d'états dans le graphe *G*. Autrement c'est le temps moyen (en secondes) nécessaire pour couvrir un certain pourcentage du graphe *G* par l'algorithme *A*.
- ✓ *Le taux de couverture* $T_c(A, G) = \frac{k}{|M|}$

  C'est le rapport du nombre moyen des états *k* visités par un algorithme donné *A*, au nombre total des états du graphe *G* (i.e. $|M|$).

- ✓ le nombre moyen d'état couverts *k* : dans le cas de graphes de très grande taille, le nombre des états atteignables peut être inconnu. On considère alors le nombre moyen d'état couverts *k*, qui sera mesuré en fonction du temps, au lieu de considérer le taux de couverture $T_c(A, G)$.

b. L'atteignabilité

Ce critère reflète la possibilité d'atteindre les états du graphe *G* par l'algorithme *A* considéré, en particulier, l'atteinte des états présentant une erreur. On obtient des informations sur l'atteignabilité à travers les probabilités suivantes :

- ✓ *La probabilité d'atteignabilité* : étant donné un graphe *G* et un algorithme d'exploration randomisé *A*, la suite ordonnée $\underline{v}_k = (v_0, v_1, ..., v_k)$ des états distincts de *G*, visités par *A* après *n* étapes est une variable aléatoire dont la probabilité sera notée $P_{A,G}(v_k, n)$. L'apparition d'un état donné *v* dans cette suite, autrement dit l'atteignabilité de *v* en *n* étapes, est une variable aléatoire dont la probabilité ;

  $P_{A,G}(v_k, n) = \sum_{v_k | v \in v_k} P_{A,G}(v_k, n)$ diffère d'un état a un autre

- ✓ La probabilité minimale d'atteignabilité : elle consiste en le minimum sur les états du graphe *G* des probabilités décrites ci haut :
  $\pi_{min}(A, G, n) = min\ P_{A,G}(v, n)$

- ✓ Le temps de détection : le principe est d'arrêter l'algorithme d'exploration *A* puis le relancer à plusieurs reprises. Etant donné un état cible *v*, ayant une probabilité d'atteignabilité π_v, le critère utilisé dans ce cas est le calcul du nombre de répétitions *R* nécessaires à l'algorithme *A* pour détecter cet état.

Conclusion et perspectives

Le développement et l'étude des systèmes embarqués, toujours plus complexes et plus grands, constituent un enjeu majeur dans le monde de l'informatique. Face à des contraintes et des exigences fortes, il est nécessaire de développer des outils permettant leur évaluation afin de renforcer leur développement et donc leur sûreté. Plusieurs approches sont possibles selon l'étude à réaliser, comme la vérification. Pour cela, il est possible d'utiliser la simulation ou bien la preuve automatique de théorèmes. Cependant, dans le cas de la simulation, on ne pourra analyser qu'un comportement d'exécution du système parmi tous les comportements possibles, et la preuve automatique reste un problème non-décidable en général, soumis à une limitation des ressources mémoires et un temps de vérification pouvant être infini.

Nous nous sommes penchés sur la méthode du model checking. Grâce à cette méthode, il est possible de vérifier l'ensemble des exécutions possibles d'un système dans un temps fini, avec en résultat un contre-exemple en cas d'échec. Les travaux de recherche sur lesujet sont nombreux et mettent en avant plusieurs problématiques encore en cours d'étude etde résolution aujourd'hui.

Notre travail pour ce mémoire était une étude approfondie de la technique du Model checking pour permettre la vérification de nouvelle propriétés : les propriétés de vivacités, ainsi que son intégration dans différents outils de vérification. Face à des systèmes non-déterministes ainsi que des automates de Büchi également potentiellement non-déterministes, le nombre d'états visités pendant la vérification augmente de façon exponentielle menant rapidement à une explosion combinatoire de l'espace d'état. Durant la dernière partie de notre travail, nous avons abordé l'approche de la vérification randomisée et on se basant sur un schéma générique qui englobe plusieurs variantes d'algorithmes d'exploration préexistants, nous avons proposé un algorithme randomisé orienté profondeur. La taille limitée de la mémoire utilisée a été introduite explicitement comme paramètre principale dans le modèle afin d'obtenir un algorithme ressources dépendants. On peut ainsi contrôler à tout moment de l'exploration l'état de la mémoire pour qu'elle ne soit pas dépassée et afin d'éviter le stagnation de l'exploration a cause du swapping.

Enfin, nous avons traîter, analytiquement l'approche de vérification randomisée, ainsi que l'algorithme proposé, une des nombreuses pistes de travail dans ce sens et de travailler sur un algorithme d'exploration randomisé orienté en largeur, et un algorithme d'exploration flexible paramétré par un taux de mixage de l'exploration en profondeur/en largeur guidé par un facteur caractéristique du graphe.

Bibliographie

[01]Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company,1990.

[02]John McCarthyand James Painter.*Correctness of a compiler forarithmetic expressions*.In Symposium in Applied Mathematics, pages33–41, 1967.

[03]Maulik A. Dave. *Compiler verification* : a bibliography. ACM SIGSOFTSoftware Engineering Notes, 28(6) :2–2, November 2003.

[04]Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and CésarViho. *Using on-the-fly verification techniques for the générationof test suites*. In Computer Aided Verification, pages 348–359. LectureNotes in Computer Science, 1996.

[05]Séverine Colin and Leonardo Mariani.*Run-time verification*. InModel-Based Testing of Reactive Systems, pages 525–555, 2004.

[06]Edmund M. Clarke, OrnaGrumberg, and DoronPeled.*Model checking*.MIT Press, 1999

[07]Klaus Havelund and Thomas Pressburger.*Model checking javaprograms using java pathfinder*. International Journal on Software Tools for Technology Transfer (STTT), 2(4) :366–381, 2000.

[08]Charles Antony Richard Hoare.*An axiomatic basis for computerprogramming*. Communications of The ACM, 12(10) :576–580, 1969

[09]Jean-Christophe Filliâtre and Claude Marché. *The why/krakatoa/-caduceusplateform for deductive program verification*. In CAV,pages 173–177, 2007.

[10]FramaC. http ://frama-c.cea.fr/

[11]Patrick Cousot and RadhiaCousot.*Abstract interpretation* : a unifiedlattice model for static analysis of programs by construction orapproximation of fixpoints. In Symposium on Principles of ProgrammingLanguages (POPL).

[12]Patrick Cousot. *Semantic foundations of program*. In S. S. Muchnickand N. D. Jones, editors, Program Flow Analysis : Theory andApplications, chapter 10, pages 303–342. Prentice-Hall, Inc., EnglewoodCliffs , New Jersey, 1981.

[13]Patrick Cousot and RadhiaCousot.*Abstractinterpretationframeworks*.Journal of Logic and Computation, 2(4), 1992.

[14]Alfred Tarski. *A lattice-theoretical fixpoint theorem and its applications*.In Pacific Journal of Mathematics, volume 5

[15]Patrick Cousot and RadhiaCousot.*Systematic design of programanalysis frameworks*.In Symposium on Principles of ProgrammingLanguages (POPL).

[16]Bruno Blanchet, Patrick Cousot, RadhiaCousot, Jérôme Feret,Laurent Mauborgne, Antoine Miné, David Monniaux, and XavierRival. *Design and implementation of a special-purpose static programanalyzer for safety-criticalreal-time embedded software*. InThe Essence of Computation

[17]Patrick Cousot. *Integrating physical system in the static analysis ofembedded control software*. In APLAS, pages 135–138, 2005.

[18]Amir Pnueli, Michael Siegel, and Eli Singerman.*Translation validation*.In Tools and Algorithms for Construction and Analysis of Systems,TACAS 98, 1384 :151–166, 1998.

[19]George C. Necula. *Translation validator for optimizing compilers*.SIGPLAN Not., 35(5) :83–94, 2000.

[20]Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg.VOC :A *translation validator for optimizing compilers*.In ENTCS, ElsevierScience, 2002.

[21]Xavier Rival. *Symbolic transfer function-based approaches to certifiedcompilation*. SIGPLAN Not., 39(1) :1–13, 2004.

[22]Jean Baptiste Tristan and Xavier Leroy.*Formal Verification of translationValidators- A Case Study on Instruction Scheduling Optimizations*.In POPL'08, 35th Symposium on Principles Of ProgrammingLanguages, January 2008.

[23]George C. Necula and Peter Lee.*Research on proof-carrying codefor untrusted-code security*. In IEEE Symposium on Security and Privacy,page 204. IEEE Computer Society, 1997.

[24]Robin Milner and R. Weyhrauch.*Proving compiler correctness in amechanised logic*.Machine Intelligence, (7), 1972.

[25]Gerwinklein and Tobias Nipkow. *Verifiedbytecodeverifiers*. TCS,298 :583–626, 2003.

[26]Gerwinklein and Tobias Nipkow. *A machine-checked model fora java-like language, virtual machine and compiler*.ACM Trans.Program. Lang. Syst., 28(4) :619–695, 2006.

[27]Martin Strecker. *Formal verification of a java compiler in isabelle*.In LNCS, editor, In Proc. Conference in Automated Deduction (CADE),volume 2392, pages 63–77. Springer-Verlag, 2002.

[28]Lawrence C Paulson.*The Foundation of a Generic Theorem Prover*.Journal of AutomatedReasoning, 5, 1989.

[29]Martin Strecker. *Compiler verification for c0.Technical report*, UniversitéPaul Sabatier, Toulouse, April 2005.

[30]Dirk Leinenbach, Wolfgag Paul, and Elena Petrova. *Towards theformal verification of a c0 compiler* : Code generation and correctness.In International Conference on Software Engineering and FormalMethods (SEFM'05), pages 2–12. IEEE Computer Society Press, 2005.

[31]Sandrine Blazy, ZaynahDargaye, and Xavier Leroy. *Formal verificationof a c compiler front-end*. In In Proceedings of Formal Methods,2006 (FM 2006), volume 4085/2006, pages 460–475. Springer-Verlag,2006.

[32]Sandrine Blazy and Xavier Leroy. *Mechanized semantics for theclight subset of the c language*.Journal of AutomatedReasoning,43(3) :263–288, 2009.

[33]Alfred Aho, Ravi Sethi, Jeffrey Ullman : *Compilateur* : Principe, technique etoutil, InterEdition

[34]Sergio Yovine, *Méthodes et outils pour la vérification symbolique de systèmestemporisés*, thèse de doctorat, Institut national polytechnique de Grenoble, 1993

[35]Rajeev Alur :*Techniques for Automatic verification of real-time systems*, thèse dedoctorat, Stanford University, 1991.

[36]A.Pnueli, *The temporal logic of programs*, proceeding of the eighteenth symposium on foundations of computer science, Providence, RI, November 1977

[37]Zohar Manna, Amir Pnueli, *The temporal logic of reactive and concurent system*, Springler-Verlag, 1992

[38]E.M.Clarke, E.A.Emerson, A.P.Sistla, *Automatic verification of finite-stateconcurent system using temporal logic specifications*, 1996.

[39]Pierre Wolper Lecture 4: *Temporal logic and their relation to automata*, transparentducour the algorithmic verification of reactive system, 1998.

[40]E. Bremont, M. Daoudou, S. Ravelomanana

[41]G. Necula, P. Lee, *Proof-Carrying Code*, in 24th ACM SIGPLAN-SIGACTSymposium on Principles of Programming Languages, pp. 106-119, Paris, France,1997.

[42]A. W. Appel, E. W. Felten, Z. Shao, *Scaling Proof-Carrying Code to ProductionCompilers and Security Policies*, Technical Report YALEU/DCS/TR-1182, Yaleuniversity, USA, January 1999.

[43]F. B. Schneider,*Enforceable Security Policies*, Department of Computer Science,Cornell University, USA, 1999.

[44]R. Whabe, S. Lucco, T. E. Anderson, S. L. Graham, *Efficient Software-Based FaultIsolation*, In the 14th ACM Symposium on Operating Systems Principles(SOSP'93), pp. 203-216, December 1993.

[45]U. Erlingsson, F. B. Schneider, *SASI Enforcement of Security Policies : ARetrospective,* In Proceedings of the 1999 New Security Paradigms Workshop,Caledon Hills, Ontario, Canada, September 22 - 24, 1999.

[46]E.M Clarke, O. Grumberg, and D. Peled.*Model Checking*.2000.

[47]F.Kroger.*Temporal Logic of Programs*,volume 8 of EATCS Monographs on Theoretical Computer Science.Springer-Verlag, 1987.

[48]E.M.Clarke and E.A. Emerson.*Design and synthesis of synchronization skeletons usingbranching time temporal logic*.In Lecture notes in Computer Science, volume 131. Springer,1982.

[49] S.A Kripke.*Semantical considerations on modal logic.*1963.

[50]LeslieLamport,*The temporal logic of action*, revised version of SRC research report 79, 1994.

[51]J.R. Buchi.*On a decision method in restricted second order arithmetic*. In Proc. International Congress on Logic, Method, and Philosophy of Science, pages 1 { 12, 1962.

[52] M. Rabin.*Decidability of second-order theories and automata on infinitetrees*.InTrans. Amer. Math. Soc, volume 141, pages 1-35. 1969.

[53] R.P. Kurshan, V. Levin, and H. Yenigun.*Compressing transitions for model checking*.InProc. of Computer Aided Veri_cation (CAV 2002). Springer, 2002.

[54] C. Flanagan and P. Godefroid.*Dynamic partial-order reduction for model checking software*.In POPL'05, 2005.

[55] C. Rosa, S. Merz, and M. Quinson.*A Simple Model of Communication APIs Applicationto Dynamic Partial-order Reduction.*In 10th International Workshop on Automated Verification of Critical Systems - AVOCS 2010, 2010.

[56] K. Ozdemir and H. Ural.*Protocol validation by simultaneous reachability analysis.InComputer Communications,*1997.

[57] K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi.*Efficient verification of real-timesystems : Compact data structure and state-space reduction.*In Proc. of Real-Time SystemsSymposium (RTSS'97), page 14-24, 1997.

[58] J. Geldenhuys and A. Valmari.*A nearly memory-optimal data structure for sets andmappings.*In Proc. of Model Checking Software (SPIN), volume 2648 of LNCS, page136-150. Springer, 2003.

[59] G. Behrmann, K.G. Larsen, and R. Pelanek.*To store or not to store*.In Proc. of ComputerAided Veri_cation (CAV 2003), volume 2725 of LNCS.Springer, 2003.

[60] P. Godefroid, G. J. Holzmann, and D. Pirottin*. State space caching revisited*. In Proc.OfComputer Aided Verification (CAV 1992), volume 663 of LNCS, page 178-191. Springer,1992.

[61] H. Garavel, R. Mateescu, and I. Smarandache.*Parallel state space construction for modelchecking.*In Proc. SPIN Workshop, volume 2057 of LNCS, page 217-234.Springer, 2001.

[62]*From distributed memory cycle detection to parallel LTL model checking*. In ENTCS, page21-39, 2005.

[63] J. Barnat, L. Brim, and J. Stribrna. *Distributed LTL model-checking in spin.*In Proc. OfSPIN Workshop on Model Checking of Software, volume 2057 of LNCS, page 200-216.Springer, 2001.

[64]J. Barnat, L. Brim, and P. Rockai.*Scalable multi-core ltl model-checking.*In Proc.OfSPIN Workshop, volume 4595 of LNCS, page 187-203. Springer, 2007.

[65]M. Mihail and C. H. Papadimitriou.*On the random walk method for protocol testing*.InProc. Computer Aided Verification (CAV 1994), volume 818 of LNCS, page 132-141.Springer, 1994.

[66]P. Godefroid and S. Khurshid.*Exploring very large state spaces using genetic algorithms.*In Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002),volume 2280 of LNCS, page 266-280. Springer, 2002.

[67]R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun.*Static partial-order reduction*.In Tools and Algorithms for the construction and analysis of systems, volume 1384 of LNCS, pages 345-357. Springer, 1998.

[68] G.J. Holzmann, The Spin Model Checker: *primer and reference manual,* Addison-Wesley, 2004.

[69] M. Dwyer, G. Avruin, J. Corbett, and Y. Hu. *Patterns in property specification for finite-state verification. In Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15. In M. Ardis, editor, 1977.

[70] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[71] H. Sivaraj and G. Gopalakrishnan. *Random walk based heuristic algorithms for distributed memory model checking*. In PDMC, ENTCS, pages 51–67. Elsevier, July 2003.

[72] R. Grosu and S. A. Smolka. *Monte carlo model checking*. In TACAS, LNCS, pages 271–286, Berlin, Heidelberg, Avril 2005. Springer.

[73] H. Kautz and B. Selman. Pushing the envelope : *Planning, propositional logic and stochastic search*. In AAAI / IAAI, pages 1194–1201, Menlo Park, CA, August 1996. AAAI Press / MIT Press.

[74] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis. *Deep random search for efficient model checking of timed automata*. In Monterey Workshop, LNCS, pages 111–124, Berlin, Heidelberg, October 2006. Springer.