Vérification dynamique formelle de propriétés temporelles sur des applications distribuées réelles

Marion Guthmuller

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Vérification dynamique formelle de propriétés temporelles sur des applications distribuées réelles

THÈSE

présentée et soutenue publiquement le 29 Juin 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Marion GUTHMULLER

Composition du jury

Président :	Gaël THOMAS	Professeur – Telecom SudParis
Rapporteurs :	Jacques JULLIAND	Professeur – FEMTO-ST
	Vivien QUÉMA	Professeur – Grenoble INP
Examinateur :	Jean-Marc VINCENT	MCF – Université Joseph Fourier
Directeurs de thèse :	Sylvain CONTASSOT-VIVIER	Professeur – Université de Lorraine
	Martin QUINSON	MCF – Université de Lorraine

LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications) — UMR 7503

Remerciements

Mes premiers remerciements vont bien sûr à mes deux directeurs de thèse, Sylvain Contassot-Vivier et Martin Quinson, qui m'ont offert un sujet de thèse et ont accepté de m'encadrer à nouveau mais dans une aventure de plusieurs années cette fois-ci. J'ai eu la chance de découvrir des personnes passionnées par leur métier et leur domaine de recherche, qui ont réussi à me transporter également dans leur univers. Je souhaite les remercier particulièrement pour leur patience et leur soutien dans les moments forts (personnels et professionnels) qui ont ponctué plus ou moins régulièrement mes quatre années de doctorat.

Je remercie ensuite l'ensemble des personnes qui m'ont fait l'honneur d'accepter de faire partie de mon jury de soutenance : Vivien Quéma et Jacques Julliand, en tant que rapporteurs, Jean-Marc Vincent en tant qu'examinateur et Gaël Thomas, examinateur et président du jury.

Mes remerciements vont ensuite à l'équipe AlGorille et chacun de ses membres dont mon encadrant de l'ombre qui se reconnaîtra. Bien que n'ayant connu et travaillé que dans cette équipe à chacune de mes venues au Loria, j'ai pu constater à travers différentes conversations avec d'autres stagiaires et doctorants que l'environnement offert au sein de cette équipe se distinguait des autres de par l'ambiance mais également la communication entre tous les membres de l'équipe, quelque soit son statut.

Je remercie aussi l'équipe du projet SimGrid. Une légende dit que mon entrée dans le projet a été marquée par des propos tout à fait innocents contre un certain F., mais comme toute légende, impossible de savoir le vrai du faux .... J'ai eu l'occasion de découvrir le fonctionnement d'un projet de grande envergure avec des spécialistes du domaine, répartis sur toute la France. Je reste toutefois partagée sur le fait que les *coding sprints* me manqueront ou pas ;-)

Heureusement pour moi, ma vie ne s'est pas arrêtée durant cette thèse. Partagée entre le virtuel avec le canal IRC `#linux` et ses *trolldis* mémorables, et le réel, notamment avec Christopho, Jarod, Jan et Lucas, LE Quorum du Chtimi toujours partant, je remercie toutes les personnes qui m'ont permis de survivre dans cette aventure.

Enfin, un grand merci à mes parents et mon nini, sans qui je n'aurais pu me lancer et surtout terminer cette thèse.

Sommaire

Introduction

De plus en plus populaires, les systèmes distribués deviennent également plus grands et, par conséquent, plus complexes. Ils restent toutefois soumis à de fortes contraintes et exigences, parmi lesquelles la nécessité d'assurer une certaine qualité dans leur développement et surtout une fiabilité dans leur utilisation dans le monde réel. La qualité d'un logiciel revêt une importance grandissante dans notre société où l'informatique est de plus en plus présente dans notre quotidien. Pour accroître cette qualité, l'une des conditions à respecter est la correction du système. Celle-ci doit assurer que le système ne connaîtra pas de défaillances mais également qu'il s'exécutera correctement grâce à la détection d'erreurs. De par leur taille, leur complexité et leurs nombreux domaines d'applications, un compromis entre la qualité et le coût de développement des logiciels est souvent réalisé. Cependant, certains logiciels peuvent être critiques, au sens où leur défaillance peut avoir des conséquences dramatiques tant du point de vue humain que matériel ou financier. Dès lors, la qualité du logiciel doit primer même si l'on souhaite contenir les coûts de développement.

Dans le cas des systèmes distribués, assurer cette correction est rendu plus difficile par leur hétérogénéité mais également par leurs spécificités communes. Un système distribué met en œuvre un ou plusieurs programmes exécuté(s) sur plusieurs machines qui communiquent entre elles à travers le réseau. Les algorithmes correspondants deviennent alors rapidement complexes et la prédiction de leur comportement difficilement réalisable sans une étude avancée. De plus, la nature distribuée des plates-formes sur lesquelles sont exécutées ces applications accroît leur risque de défaillance, en raison du manque de fiabilité des canaux de communication entre les machines ou de problèmes sur les machines distantes.

Motivations. Pour étudier ces systèmes, les approches pratiques les plus courantes sont les tests et la simulation. Toutefois, celles-ci deviennent inadaptées dans le cas d'applications de grandes tailles au comportement non-déterministe avec notamment des résultats incomplets. En effet, réaliser des tests exhaustifs et, ainsi, pouvoir affirmer qu'il n'y a aucune erreur indétectée à la fin de la phase de tests, implique deux contraintes majeures : cela nécessite de déterminer l'ensemble des entrées possibles d'un programme puis l'ensemble des exécutions possibles selon ces entrées. Ceci peut se révéler difficile avec l'utilisation seule des méthodes de test classiques face à des systèmes complexes. La mise en œuvre de techniques de test logiciel telles que la couverture de code aide toutefois à assurer qu'un ensemble adéquat des comportements possibles du programme a été observé, mais ceci ne garantit pas l'exhaustivité de la vérification. Dans le cas de la simulation, des modèles théoriques servent à étudier le comportement et les propriétés d'un système modélisé ainsi qu'à en prédire son évolution. Il n'est cependant possible d'étudier qu'un chemin d'exécution à la fois, soumis aux conditions de la plate-forme expérimentale simulée. Dès lors, il est impossible de garantir le bon fonctionnement et la sûreté complète du système puisqu'il n'est que partiellement étudié. C'est dans ce contexte qu'une vérification formelle à travers la preuve de propriétés sur le système doit être considérée.

Plusieurs méthodes formelles basées sur la sémantique des programmes, c'est-à-dire sur une description formelle du sens d'un programme donné par son code source (ou, parfois, son code objet), peuvent s'appliquer à la vérification des systèmes distribués. Parmi elles, le *model checking* est une approche intéressante dans le cas des systèmes distribués car il permet d'étudier tous les chemins d'exécution possibles. La présence de non-déterminisme reste courante dans ce type de systèmes et nécessite donc d'être prise en compte pour assurer une vérification exhaustive. Toutefois, dans son utilisation traditionnelle, le *model checking* reste difficile à appliquer à des applications réelles car il nécessite un modèle complet de l'application, généralement dans un formalisme spécifique, très différent du code source utilisé pour exprimer l'application étudiée.

Objectif de la thèse. L'objectif de cette thèse est d'adapter des techniques de *model checking* pour la vérification d'applications en travaillant sur leur implémentation réelle et non sur un modèle abstrait de celles-ci. Nous appellerons l'approche des travaux présentés *Vérification dynamique formelle*, ou bien *Vérification de modèle basée sur l'exécution* (*Execution-based Model Checking*). Dans cette approche, l'implémentation réelle d'un programme est fournie en entrée de l'outil de vérification et l'espace d'états correspondant est construit, exploré et vérifié à la volée à travers l'exécution réelle du programme que l'on souhaite vérifier. Nous exploitons donc les techniques du *Software Model Checking* pour permettre une vérification formelle exhaustive et les techniques de la Vérification dynamique, souvent associée au Test logiciel, en vérifiant l'application réelle à travers son exécution.

Problématique. L'enjeu majeur dans la vérification dynamique formelle est de réussir à appliquer les techniques associées au *model checking* dans le cas d'une vérification sur des applications réelles et non plus sur des modèles. Sachant que ces techniques supposent majoritairement que le modèle complet et l'ensemble des informations associées sont connus *a priori*, il est nécessaire de les réadapter à l'étude d'applications réelles.

Dans cette thèse, nous nous intéressons en particulier aux applications distribuées écrites en C, C++ ou Fortran qui permettent à des processus séquentiels communiquants (CSP) d'interagir par échange de messages en utilisant l'interface MPI ou des interfaces de programmation équivalentes. La première difficulté est d'extraire des informations pertinentes pour la vérification formelle depuis le système en fonctionnement, sachant que les langages cibles n'offrent ni introspection, ni même gestion automatisée de la mémoire. Une fois ces informations obtenues, la seconde difficulté est alors d'adapter des méthodes classiques pour la vérification formelle de certaines propriétés exprimées en logique temporelle, linéaire ou arborescente, sur ces applications. Grâce à l'utilisation de modalités, il est possible d'exprimer des informations sur le temps et ainsi de spécifier des propriétés plus complexes et plus pertinentes dans le contexte de la vérification des systèmes distribués.

Contributions. Nous présentons dans ce document trois contributions, une majeure et deux secondaires, répondant à cette problématique. La contribution majeure est une analyse sémantique dynamique par introspection mémoire d'un état système. Cette analyse est basée sur une introspection de la mémoire dynamique d'un état système à l'aide d'outils de *debug* tels que DWARF et libunwind qui permettent de reconstruire la sémantique d'un état système. Il est alors possible d'analyser et d'interpréter l'ensemble des données dynamiques qui constituent un état système et ainsi les exploiter de différentes manières. Nous évaluons notamment cette contribution à travers trois types d'expériences : la mise en œuvre d'une vérification *stateful* avec détection des états déjà visités, la vérification de protocoles cycliques et l'analyse partielle de la terminaison de programmes.

La seconde contribution, secondaire, est la vérification dynamique formelle de propriétés de vivacité exprimées à l'aide de la logique $\text{LTL}_{-\mathbf{X}}$. Cette vérification étant basée sur la détection de cycles acceptants, elle nécessite d'analyser les états explorés au cours de la vérification afin de déterminer si un cycle apparaît sur une exécution. En s'appuyant sur la contribution précédente pour l'analyse des états, nous mettons en œuvre un nouvel algorithme d'exploration en profondeur de l'ensemble des exécutions possibles, basé sur l'algorithme NDFS (*Nested Depth First Search*). Cet algorithme réalise un unique parcours en profondeur, et non deux comme dans sa version originale, et permet la recherche de plusieurs cycles acceptants en même temps pour chaque chemin d'exécution. Cette contribution est évaluée à travers la vérification d'une pro-

priété de vivacité sur l'algorithme d'exclusion mutuelle centralisé.

Enfin, nous présentons en troisième contribution, également secondaire, la vérification dynamique formelle du déterminisme des communications dans les applications MPI. Cette contribution répond à un besoin pratique dans la communauté HPC à très large échelle notamment dans le contexte du développement de protocoles de tolérances aux pannes efficaces et adaptés au comportement des applications MPI, aucun outil automatique ne permettant actuellement cette étude. Cette dernière consiste à évaluer les communications réalisées par chaque processus d'une application MPI pour chaque exécution et à vérifier que celles-ci sont réalisées dans le même ordre sur l'ensemble des exécutions, caractérisant ainsi la présence d'un déterminisme. Basée sur une formulation à l'aide de la logique CTL et un algorithme *ad-hoc* spécifique à l'étude du déterminisme, cette vérification permet alors de caractériser automatiquement la présence ou non de certains déterminismes dans les communications d'applications MPI.

Structure du document. Avant d'exposer nos travaux, nous commençons par détailler dans la partie I le contexte scientifique et technique de cette thèse. Pour cela, nous présentons dans le chapitre 1 les systèmes distribués et les propriétés de correction pouvant être définies sur ces derniers. Nous terminons ce chapitre en définissant le champ d'étude des travaux présentés dans ce document, notamment le type des applications visées ainsi que le type de propriétés que l'on souhaite vérifier. Puis, nous analysons dans le chapitre 2 les différentes approches de vérification des systèmes distribués, basées sur le test logiciel et la vérification formelle, et discutons des limites de chacune. Nous terminons cette partie en présentant dans le chapitre 3 le contexte technique de nos travaux avec SimGridMC, l'outil de vérification dans lequel nos travaux ont été implémentés.

La partie II introduit la première contribution de cette thèse : l'analyse sémantique dynamique par introspection mémoire d'un état système. Cette partie se décompose en trois chapitres. Le chapitre 1 commence par présenter les motivations de cette analyse ainsi qu'un état de l'art des approches existantes. Nous détaillons ensuite, dans le chapitre 2, les différents verrous rencontrés lors de l'analyse sémantique d'un état système et les solutions que nous apportons permettant la détection d'états sémantiquement identiques. Enfin, ces travaux sont évalués expérimentalement dans le chapitre 3.

En partie III, nous présentons les deux contributions secondaires de cette thèse : la vérification dynamique formelle de propriétés de vivacité exprimées à l'aide de la logique LTL_{-X} ainsi que l'étude du déterminisme des communications MPI exprimé en logique CTL. Cette partie se décompose alors en deux chapitres.

Le chapitre 1 détaille dans un premier temps la vérification de propriétés de vivacité. Pour exposer nos travaux, nous commençons par expliquer les motivations de cette vérification, en particulier la catégorie de propriétés dans le contexte de l'étude des systèmes distribués. Puis, nous introduisons dans un second temps le principe général de leur vérification, notamment leur formulation à l'aide d'une logique temporelle et l'algorithme d'exploration habituellement mis en œuvre en *model checking*. Nous détaillons alors ensuite notre approche permettant de réaliser cette vérification sur des implémentations réelles avec notamment la mise en œuvre d'un nouvel algorithme d'exploration. Enfin, cette approche est évaluée à travers la vérification d'une propriété de vivacité exprimée en $\text{LTL}_{-\mathbf{X}}$.

Le chapitre 2 aborde la dernière contribution de cette thèse : la vérification dynamique formelle du déterminisme des communications dans les applications MPI exprimé avec la logique CTL. Nous commençons ce chapitre en exposant les motivations de cette vérification dans le contexte particulier de la tolérance aux pannes. Puis, nous présentons les différents

déterminismes pouvant être exprimés selon les communications MPI étudiées. La vérification de ces déterminismes est réalisée grâce à un algorithme *ad-hoc*, détaillé ensuite dans le chapitre. Enfin, nous validons et évaluons cette contribution à travers l'étude de plusieurs déterminismes sur différentes applications MPI.

Nous terminons ce document en établissant en dernière partie une conclusion sur l'ensemble des travaux présentés ainsi qu'une analyse des perspectives possibles.

Première partie

Contexte et État de l'art

Chapitre 1

Contexte scientifique

Nous présentons dans ce premier chapitre le contexte scientifique des travaux présentés dans cette thèse, c'est-à-dire les systèmes distribués et les propriétés de correction pouvant être vérifiées sur ces derniers et définissons le champ d'étude des travaux réalisés au cours de cette thèse.

1 Les systèmes distribués

1.1 Définition

Un *système distribué* peut se définir par opposition à un *système centralisé* [Gho10]. Dans un système centralisé, tout est regroupé sur une seule et même machine. L'ensemble des informations est accessible aux programmes qui s'exécutent sur cette dernière. Il y a alors une notion de localité des ressources. Dans le cas d'un système distribué, un ou plusieurs programmes, que nous appellerons *applications distribuées* dans la suite de ce document, s'exécutent sur plusieurs machines indépendantes (éventuellement hétérogènes) réparties sur un réseau et qui communiquent via ce dernier grâce à des messages. Bien que physiquement distinct, cet ensemble reste transparent du point de vue de l'utilisateur qui le voit comme une seule entité.

Les premières mises en œuvre de processus concurrents qui communiquent à travers des messages remontent aux années 1960 et 1970, notamment dans l'architecture des systèmes d'exploitation. L'exemple majeur d'un système distribué à grande échelle est tout simplement Internet, où une myriade de machines autonomes sont interconnectées pour former un réseau dans lequel elles communiquent à travers des messages. La communication entre les machines est régie par des mécanismes appelés protocoles et qui permet à n'importe quelle machine de communiquer avec n'importe quelle autre machine. Aujourd'hui, l'utilisation de systèmes distribués peut être motivée par deux raisons principales : le partage de ressources distantes et la robustesse du système.

1.2 Catégories

Les systèmes distribués peuvent être classés en plusieurs catégories, selon les motivations de leur utilisation mais également la topologie des réseaux sur lesquels ils sont interconnectés.

HPC et Grilles. Le principe de ces systèmes est de réunir plusieurs machines (de quelques dizaines à plusieurs milliers) dans le but d'agréger leur puissance de calcul permettant ainsi la résolution de gros calculs en un temps réduit. Ce type de système est notamment utilisé pour

la simulation de phénomènes étudiés dans des disciplines telles que la physique des particules, la biologie ou la cosmologie. Dans le cas des systèmes de calculs haute performance (HPC), l'ensemble forme des superordinateurs composés de plusieurs milliers de machines homogènes sur un même site (on parle alors de *cluster* HPC), chacune possédant des processeurs multi-cœurs. On notera, par exemple, le superordinateur Tianhe-2 du Centre National des superordinateurs de Guangzhou en Chine, le plus rapide au monde (classement de Juin 2014 du TOP500 [1]) composé de 3 120 000 cœurs, répartis sur 32 000 microprocesseurs Intel Xeon, pour une puissance de calcul de 33 862 TFlops (opérations à virgule flottantes par seconde). Les grilles de calculs sont quant à elles composées de machines généralement hétérogènes sur des sites différents, ce qui les différencie des clusters HPC. Ces systèmes posent des défis d'échange de données et d'intéropérabilité entre organisations virtuelles.

Systèmes pair-à-pair. Ces systèmes font partie des systèmes distribués « grand public ». En effet, il existe de nos jours très peu de différences du point de vue matériel entre les ordinateurs personnels et les machines qui constituent les clusters HPC présentés précédemment. Avec la popularisation des connexions réseaux à grande vitesse, il devient possible d'exploiter facilement ces machines disponibles sur le réseau, notamment pour le partage de fichiers ou bien la diffusion de vidéos. La principale difficulté dans ces systèmes est le manque d'organisation entre les machines qui constituent l'ensemble. Il est alors nécessaire de mettre en œuvre des solutions totalement distribuées où les ordinateurs communiquent directement entre eux, sans serveur central, justifiant leur nom de systèmes pair-à-pair. L'absence de point central dans le système nécessite de gérer la volatilité des machines qui peuvent quitter le système sans en informer les autres pairs, générant une latence avant de retrouver un système stable.

Les *Clouds*. Le *cloud computing* ou informatique en nuage met en œuvre des techniques de virtualisation permettant l'encapsulation de programmes au sein de machines virtuelles. Il est ainsi possible de créer plusieurs machines virtuelles sur une seule machine physique et ainsi en exploiter au maximum les ressources. Ces systèmes sont aujourd'hui industrialisés notamment par IBM, Amazon ou Microsoft par exemple et permettent l'accès à des ressources distantes via un réseau de télécommunications. Grâce à une approche de *self-service* avec un paiement à l'usage, ces systèmes sont exploités à la fois pour le calcul et le stockage de données sur les serveurs qui les composent. Le terme de nuage est utilisé car les utilisateurs ne savent pas où sont situés les serveurs qu'ils exploitent. La difficulté dans la gestion de ces systèmes se situe du côté des fournisseurs qui doivent optimiser au mieux l'utilisation des plates-formes, notamment le placement des ressources virtuelles sur les différentes ressources physiques selon les applications afin de garantir les meilleures performances.

1.3 Problèmes spécifiques aux applications distribuées

Bien qu'il existe différentes catégories de systèmes distribués, leur architecture reste commune à tous, avec notamment un modèle algorithmique identique. Ces systèmes représentent un ensemble d'entités autonomes qui s'exécutent de façon asynchrone et qui communiquent à travers l'échange de messages. Les algorithmes développés pour ce type de système se singularisent par trois caractéristiques :

1. `http://www.top500.org/`

Vue limitée de l'état global du système. Dans un système centralisé, l'ensemble des éléments qui constituent le système ont une vision globale de celui-ci, rendant ainsi observable l'état complet du système à tout instant de l'exécution. Dans le cas des systèmes distribués, ceci est plus complexe car chaque entité s'exécute de façon indépendante par rapport au reste du système. Il y a alors un état local à chaque processus. Pour obtenir des informations sur l'état des autres processus, il est nécessaire de communiquer avec ces derniers. Il subsiste donc toujours un temps, plus ou moins important selon la fréquence des échanges d'informations, durant lequel chaque processus a potentiellement des informations obsolètes sur l'état global du système, ce qui peut conduire à des erreurs.

Pas d'horloge globale. Dans un système centralisé sans parallélisme, les évènements produits par chaque processus sont naturellement totalement ordonnés selon l'instant de leur exécution. Il est ainsi possible d'ordonner deux à deux chaque évènement en déterminant lequel a eu lieu avant l'autre. Dans le cas des systèmes distribués, cet ordre total ne peut être observé car les processus s'exécutent sur des machines distinctes dont les horloges ne sont pas synchronisées. En l'absence d'horloge globale, il n'est ainsi pas toujours possible de déterminer avec certitude si tel évènement a eu lieu avant, après ou même pendant une autre action.

Non-déterminisme. Cette caractéristique est commune aux systèmes distribués et aux systèmes centralisés dès lors qu'il existe du parallélisme dans le système. En effet, lorsque plusieurs processus s'exécutent en parallèle, des situations de compétition (*race conditions*) peuvent facilement apparaître, menant à des résultats d'exécution différents selon l'ordre dans lequel les processus se sont exécutés. Pour pallier cela, il est nécessaire de mettre en œuvre des schémas de synchronisations pour assurer la cohérence des résultats. Toutefois, ceci complexifie les algorithmes et rend alors particulièrement difficile l'analyse du comportement de l'application. Il peut donc en résulter rapidement des interblocages (*deadlocks ou livelocks*) ou des problèmes d'équité entre processus.

La vue limitée de l'état global du système, l'absence d'horloge globale et la présence de non-déterminisme rendent donc le développement d'algorithmes distribués plus complexe. Ceci augmente à la fois le risque d'erreurs lors de l'implémentation mais également la difficulté à détecter celles-ci.

2 Propriétés de correction

Les propriétés les plus simples pouvant être vérifiées sont *l'accessibilité* et *l'invariance*. L'accessibilité définit qu'un état donné est accessible depuis l'état initial. Par exemple, « *il existe un état accessible où x est égal à 0* » est une propriété d'accessibilité. L'invariance définit que tous les états du système vérifient une certaine propriété. Par exemple, « *x est différent de 0 dans tous les états accessibles* » est une propriété d'invariance. Cependant, ces propriétés ne permettent pas de formuler des propriétés d'exécutions, en particulier sur des séquences d'états. Pour cela, la logique temporelle est souvent utilisée. Il existe alors deux catégories de propriétés : les propriétés de sûreté et les propriétés de vivacité [Lam77].

2.1 Logique temporelle

La logique temporelle [GHRF94] est une logique modale issue de la logique classique à laquelle ont été intégrés des opérateurs dédiés au temps mais également des connecteurs temporels

ainsi que des quantificateurs de chemins. Qu'elle soit linéaire ou arborescente, elle permet ainsi d'exprimer des propriétés de correction dont la satisfaction varie au cours de l'exécution du système. Il est par exemple possible de formuler des propriétés telles que « La variable x est *toujours* égale à 0 », « La variable x sera *finalement* égale à 0 » ou « La variable x sera égale à 0 jusqu'à ce que la variable y soit égale à 0 ». Grâce à la logique temporelle, un raisonnement peut donc être fait dans le temps.

Sachant qu'il existe plusieurs types de modalités, il en découle plusieurs logiques temporelles. Celles-ci se différencient selon qu'elles sont linéaires ou arborescentes, selon leur expressivité (syntaxique et sémantique), avec ou sans passé et bien sûr selon leur complexité. Parmi elles, les plus couramment utilisées sont la logique temporelle linéaire (LTL) et la logique temporelle arborescente (CTL).

2.1.1 Logique temporelle linéaire (LTL)

Également appelée logique temporelle de propositions (PTL ou PLTL), la logique LTL [Pnu77] est un sous-ensemble de la logique CTL* [ES89]. Elle permet d'exprimer des propriétés d'exécution en prenant en compte le futur. Il est, par exemple, possible d'exprimer qu'une certaine propriété sera finalement vraie ou bien qu'une condition est vraie jusqu'à ce qu'une autre le devienne. Le domaine d'interprétation d'une formule LTL est un ensemble de chemins tel que chaque chemin est une séquence infinie d'états $s \in S$. L'évaluation de la satisfaction de la propriété se fait donc pour chaque chemin d'exécution indépendamment des autres, sans prendre en compte les entrelacements des différents futurs possibles à un instant donné de l'exécution. Autrement dit, l'évaluation se fait sur un ensemble d'exécutions indépendantes plutôt que sur un arbre des exécutions possibles.

Définition 1 (Formule LTL). Une formule LTL est définie par :

- un ensemble fini AP de variables de propositions (propositions atomiques)
- des opérateurs logiques : $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, *vrai* et $faux$
- des opérateurs modaux :
  - **X** (autre syntaxe : $\bigcirc$) : *neXt (demain)* tel que $\mathbf{X}p$ signifie que p est vraie dans l'état suivant le long de l'exécution ;
  - **G** (autre syntaxe : $\Box$) : *Globally (toujours)* tel que $\mathbf{G}p$ signifie que p est vraie dans tous les états ;
  - **F** (autre syntaxe : $\Diamond$) : *Finally (un jour)* tel que $\mathbf{F}p$ signifie que p est vraie plus tard au moins dans un état ;
  - **U** : *Until (jusqu'à ce que)* tel que $p\mathbf{U}q$ signifie que p est toujours vraie jusqu'à un état où q est vraie ;
  - **R** : *Release* tel que $p\mathbf{R}q$ signifie que q est toujours vraie sauf à partir du moment où p est vraie, sachant que p n'est pas forcément vraie un jour.

Exemple 1. La propriété d'invariance « *Durant toute l'exécution, x est différent de 0* » équivaut à la formule LTL : $\mathbf{G}\neg(x = 0)$.

Exemple 2. $\mathbf{G}(p \rightarrow \mathbf{F}q)$ signifie « *Pendant toute l'exécution, si p est vraie à un état donné, alors q sera finalement vraie au moins dans un des états suivants.* »

2.1.2 Logique temporelle arborescente (CTL)

La logique CTL [CE82] est une restriction de la logique CTL*. Elle permet de tenir compte de tous les futurs possibles de chaque état de l'exécution. Le futur y est donc considéré comme indéterminé contrairement à LTL. Les propriétés sont alors exprimées sur l'arbre des exécutions (interdépendantes) et non plus sur chaque chemin indépendamment des autres. Par exemple, il est possible de formuler en CTL qu'à partir du moment où une condition est vraie, alors pour tous les chemins d'exécution possibles il n'y aura pas de mauvais comportement (≪ *si toutes les variables d'un programme sont strictement positives, alors il n'y aura jamais de division par 0* ≫).

Définition 2 (Formule CTL). Une formule CTL est définie par les mêmes informations qu'une formule LTL, complétées du quantificateur de chemin **E** : *Exists (il existe au moins un chemin)*. Une restriction impose également que les opérateurs modaux **X**, **G**, **F** et **U** soient précédés d'un quantificateur de chemin.

Cette obligation de toujours quantifier les futurs possibles d'un état limite l'expressivité de la logique CTL. Il est alors impossible de combiner plusieurs opérateurs modaux tout en se référant à un seul chemin d'exécution. Par conséquent, une formule CTL est considérée comme une formule d'état. Cela signifie que sa satisfaction dépend de l'état courant et des états accessibles depuis celui-ci, et non pas du chemin d'exécution courant. Toutefois, cette restriction est compensée par une complexité réduite pour la vérification de formules CTL par rapport à des formules LTL, les algorithmes de vérification CTL étant polynômiaux tandis que ceux de LTL sont PSPACE.

Exemple 3. La propriété d'accessibilité ≪ *Il existe un état où x est égal à 0* ≫ se traduit par la formule CTL : $\mathbf{EF}(x = 0)$.

Exemple 4. La formule $\mathbf{AGF}p$ n'est pas une formule CTL valide.

2.2 Propriétés de sûreté

Les propriétés de sûreté permettent de spécifier que ≪ *quelque chose de mauvais n'arrivera jamais* ≫. L'invariance est donc un cas particulier de sûreté mais pas l'accessibilité.

Exemple 5. Les propriétés suivantes sont des propriétés de sûreté :
— ≪ *Deux processus ne peuvent être au même moment en section critique.* ≫
— ≪ *x est toujours différent de 0.* ≫
— ≪ *Il n'y a pas de deadlock.* ≫

Leur satisfaction est évaluée sur une séquence finie d'états, en vérifiant indépendamment chacun des états du chemin. En cas de violation, le contre-exemple fourni par l'outil de vérification sera donc fini (contrairement aux propriétés de vivacité). Ainsi, une propriété de sûreté peut seulement être satisfaite en un temps infini (on n'est jamais sûr) mais violée en un temps fini (quand le ≪ mauvais ≫ arrive). On parle alors de correction partielle lors de ce type de vérification.

Une propriété de sûreté peut être formulée avec la logique temporelle sous la forme $\mathbf{AG}\neg p$. Dès lors, il est possible de la vérifier avec les algorithmes de vérification LTL ou CTL. Cependant, la vérification de ces propriétés se réduit à l'analyse de l'accessibilité d'un état dans lequel la

propriété n'est pas satisfaite. Il est alors possible de simplifier l'expression d'une propriété de sûreté par une assertion. L'algorithme de vérification consiste dans ce cas à simplement parcourir de façon exhaustive l'espace d'états en vérifiant pour chaque état que l'assertion est toujours vraie.

2.3 Propriétés de vivacité

La vérification de propriétés de sûreté n'est pas suffisante pour établir la correction d'un système. En effet, elles ne tiennent pas compte de l'évolution du système durant son exécution. En particulier, si aucune vérification n'est faite sur l'évolution du système, il est facile d'affirmer qu'une propriété de sûreté est satisfaite sans pour autant que le système avance. Si le système n'évolue pas, il n'y a pas de risque que « quelque chose de mauvais » ait lieu. Il est donc nécessaire d'ajouter une notion de progrès dans les propriétés d'exécution. Les propriétés obtenues sont alors des propriétés de vivacité. Elles spécifient que « *quelque chose de bon arrivera finalement* ».

Exemple 6. Les propriétés suivantes sont des propriétés de vivacité :
— « *L'exécution se terminera finalement.* »
— « *Si un client envoie une requête, il recevra finalement une réponse.* »

Inversement aux propriétés de sûreté, une propriété de vivacité peut seulement être satisfaite en un temps fini (lorsque le « bon » arrive) mais violée en un temps infini (on espère toujours que le « bon » finira par arriver). Plus précisément, leur non-satisfaction signifie que « le bon n'arrive jamais », la difficulté est donc de pouvoir affirmer qu'il n'arrivera réellement jamais. Il est donc nécessaire de poursuivre la vérification tant que « le bon » n'arrive pas, dans l'espoir qu'il arrivera finalement. Sans poursuite dans le futur, nous ne pouvons ni affirmer que la propriété est satisfaite, ni qu'elle est violée. Cette étude est donc réalisée sur des exécutions infinies. Une approche pour vérifier ce type de propriété est alors de rechercher un cycle durant l'exécution du système dans lequel la propriété n'est pas satisfaite, permettant ainsi d'affirmer que le « bon n'arrivera jamais » dans ce cas.

3 Champ d'étude

Les applications distribuées implémentées pour les systèmes présentés en section 1 de ce chapitre pouvant êtres nombreuses et de natures différentes, nous définissons un champ d'étude précis pour la mise en œuvre des contributions exposées dans ce document. Nous détaillons ainsi la catégorie d'applications visées et le modèle d'atomicité qui en découle ainsi que le type d'étude envisagée, c'est-à-dire la définition des propriétés que l'on souhaite vérifier et le type de vérification en terme d'exhaustivité.

3.1 Applications visées

3.1.1 Type d'applications

Les travaux réalisés au cours de cette thèse se concentrent en particulier sur les systèmes implémentant des applications distribuées MPI écrites en Fortran, C ou C++ mettant en œuvre des processus séquentiels communiquants (CSP).

Ce type d'application est basé sur le principe du rendez-vous. Chaque processus communique avec les autres processus par l'intermédiaire d'un point de rendez-vous. Lorsqu'un processus

souhaite communiquer avec un autre processus, il envoie un message sur un point de rendez-vous donné en précisant le processus destinataire. Tout processus peut alors se rendre sur ce point de rendez-vous et vérifier si un message lui est destiné. Le message n'est ainsi réellement transmis que lorsque le processus destinataire accepte de le recevoir en consultant le point de rendez-vous. Ce mécanisme permet alors de mettre en œuvre deux scénarios possibles d'envoi de message entre processus. Dans un premier cas, le processus qui envoie le message effectue l'envoi de celui-ci sur le point de rendez-vous avant que le processus destinataire ne le consulte et récupère ainsi le message. Dans un second cas, le processus destinataire peut consulter le point de rendez-vous avant que tout message lui étant destiné n'y soit déposé. Que ce soit pour l'envoi ou la réception de messages, ces opérations peuvent être synchrones et donc bloquantes, ou asynchrones et donc non bloquantes.

Parmi ces applications, nous nous intéressons à deux comportements spécifiques : les applications issues essentiellement du monde HPC, qui correspondent majoritairement à de l'algèbre numérique et dont seul le nombre de processus varie, et les applications implémentant des protocoles cycliques au comportement potentiellement infini.

3.1.2 Modèle

Un système distribué est formé par un ensemble de processus communiquant par messages, sans mémoire partagée ni horloge globale. On note P, l'ensemble de ces processus qui sont interconnectés à travers des canaux de communication au sein d'un même réseau. Chaque processus $p \in P$ a un état local noté s_p qui évolue au cours de l'exécution du système. On note E l'ensemble des évènements d'une exécution et $e^p \subseteq E$ l'ensemble des évènements exécutés par un processus $p \in P$. Un évènement peut être de deux natures : il peut s'agir soit d'une communication, soit d'un calcul.

L'ordre total de ces évènements ne pouvant être évalué correctement dans le contexte des systèmes distribués, l'approche consiste à se baser sur l'ordre partiel de la relation arrivé-avant (*happened-before relation*) qui repose sur la causalité entre évènements. Introduite par Leslie Lamport en 1978 [Lam78], cette relation définit que tous les évènements exécutés par un même processus sont totalement ordonnés selon l'ordre séquentiel naturel. Ainsi, étant donnés un évènement e_1^p exécuté par un processus $p \in P$ puis un évènement e_2^p exécuté ensuite par ce même processus, on dit que l'évènement e_1^p précède localement l'évènement e_2^p et on le note $e_1^p \prec e_2^p$. De plus, un évènement $e^{p_1} \in E$ exécuté par un processus $p_1 \in P$ précède globalement un évènement $e^{p_2} \in E$ exécuté par un processus $p_2 \in P$, si e^{p_1} correspond à l'envoi d'un message par le processus p_1 et e^{p_2} à la réception du message correspondant par le processus p_2. On le note alors $e^{p_1} \rightsquigarrow e^{p_2}$. Il en résulte la définition 3 suivante :

Définition 3 (Relation arrivé-avant). La relation arrivé-avant entre évènements, notée $\rightarrow$, est définie par :

1. $\forall (e_1, e_2) \in E \times E : (e_1 \prec e_2) \vee (e_1 \rightsquigarrow e_2) \Rightarrow (e_1 \rightarrow e_2)$;
2. $\exists e_3 \in E : (e_1 \rightarrow e_2) \wedge (e_2 \rightarrow e_3) \Rightarrow (e_1 \rightarrow e_3)$.

Une exécution d'un système distribué est alors définie par un couple $(E, \rightarrow)$ tel que E correspond à l'ensemble des évènements et $\rightarrow$ est une relation d'ordre partiel sur ces évènements. Le comportement d'un système distribué, c'est-à-dire l'ensemble des exécutions possibles, représentable à travers un graphe d'états, est caractérisé par l'ensemble des relations $\rightarrow$ possibles entre l'ensemble des évènements E. Sachant que certains évènements peuvent être concurrents, il est

alors possible d'avoir deux évènements non ordonnés par la relation arrivé-avant créant ainsi le non-déterminisme des exécutions et donc la présence de plusieurs exécutions possibles pour un même système.

L'une des difficultés dans la vérification des systèmes distribués est de définir et représenter l'espace d'états, c'est-à-dire l'ensemble des états qui composent chaque chemin d'exécution possible ainsi que les transitions entre chacun d'eux, formant ainsi un graphe.

Chaque nœud correspond à l'état global courant du système. Celui-ci contient l'état local de chaque processus ainsi que l'état du réseau. L'état du réseau est le seul élément partagé entre les processus et est caractérisé par les messages en transit. Il comprend l'ensemble des communications déposées dans les points de rendez-vous ainsi que les zones mémoires sources et destinations des processus associées à ces communications.

La principale difficulté dans la définition et la représentation de l'espace d'états réside cependant dans la détermination de la granularité des transitions. Il est en effet important de définir des transitions qui sont atomiques sachant que des états non observables peuvent résulter d'une partie de l'exécution d'une transition. Dès lors, la vérification peut être incomplète et donc incorrecte. Une approche serait donc de définir une granularité fine afin de garantir une vérification précise et complète. Toutefois, certaines actions du système sont déterministes et n'affectent pas son état global mais uniquement l'état local du processus à l'origine de celles-ci. Associer une transition à ce type d'action peut alors engendrer des états non pertinents dans la vérification et donc impacter considérablement le temps de vérification. Dans notre contexte, deux actions peuvent être réalisées par les processus du système : un calcul ou une communication. Un calcul est déterministe et est associé à un seul processus, n'affectant alors que son état local. Le résultat de celui-ci peut éventuellement être transmis aux autres processus du système en le communiquant par message. Dès lors, l'état du réseau, seul élément partagé entre les processus et affectant donc l'état global du système, ne peut être modifié que lorsqu'une communication est réalisée par l'un d'eux.

C'est ainsi que nous ne considérons que les actions correspondant à une communication comme visibles pour la représentation de l'espace d'états. De plus, le mode d'exécution peut être différent selon les systèmes : soit un seul processus effectue une ou plusieurs actions à chaque pas d'exécution, soit tous les processus ayant une ou plusieurs actions à exécuter le font en même temps à chaque pas d'exécution. Le premier mode d'exécution est mis en œuvre dans notre contexte. Une transition vue par l'outil de vérification comprend donc la modification de l'état du réseau suite à une communication d'un des processus mais également l'ensemble des instructions de calculs suivant celle-ci jusqu'à la prochaine communication.

3.2 Type d'étude envisagée

Parmi les propriétés de correction pouvant être vérifiées sur ces applications en particulier, nous nous intéressons dans ce document à des propriétés intégrant une notion de progrès et tenant compte de l'évolution du système. Il s'agit notamment de la vérification de la progression du système à travers la détection de *livelocks* ou cycles mais également des propriétés de vivacité plus spécifiques au comportement de l'application étudiée, telle que l'exclusion mutuelle lorsque des accès concurrents protégés sont mis en œuvre.

De plus, le modèle d'atomicité choisi pour les applications visées étant basé sur les communications réalisées au cours de l'exécution, nous nous intéressons également à des propriétés caractérisant ces dernières, telle que leur déterminisme sur l'ensemble des exécutions possibles.

3.3 Exhaustivité de la vérification

L'approche de vérification que nous mettons en œuvre dans les travaux présentés dans ce document est basée sur une exploration exhaustive des exécutions possibles à partir d'un état initial donné. Les applications étudiées pouvant varier en nombre de processus mais également avec des données en entrée du système, l'état initial est construit pour une valeur de ces variables. Cette valeur initiale du nombre de processus ou des éventuelles données en entrée est définie soit par l'utilisateur en paramètre d'exécution, soit directement dans l'implémentation de l'application. Nous parcourons alors l'ensemble des exécutions possibles pour cet état initial, en explorant les différents ordres de communications possibles.

Chapitre 2

Vérification des systèmes distribués

Nous présentons dans ce chapitre les différentes approches de vérification des systèmes distribués ainsi que les outils existants dans ce domaine. Nous terminons par une discussion sur les limites de chacune d'entre elles dans notre contexte et justifions ainsi l'approche mise en œuvre dans les différentes contributions.

1 Introduction

La première approche pour s'assurer du bon fonctionnement d'un logiciel est de simplement l'exécuter, sans instrumentation particulière, et de constater les éventuelles erreurs au cours de l'exécution. Face à des projets de grande taille, des plates-formes expérimentales telles que Grid5000 [CDD+05] ou PlanetLab[CCR+03] ont été développées et permettent aujourd'hui de réaliser des expérimentations à grande échelle pour l'étude de nombreux types de systèmes. Cependant, en l'absence d'instrumentation de l'exécution, cette approche ne garantit pas que le logiciel ne contient pas d'erreur sous d'autres conditions d'expérimentation. La simple exécution d'un programme n'est donc pas suffisante pour la vérification d'un logiciel dans le but d'améliorer sa correction.

À l'opposé, la simulation permet de réaliser des expériences dans des environnements virtuels divers en les simulant à l'aide de modèles adéquats. Cependant, cette approche nécessite généralement de réécrire l'application étudiée en utilisant une interface spécifique au simulateur. Pour éviter cette phase de réécriture qui peut être extrêmement coûteuse pour des systèmes complexes, une approche intermédiaire est possible : l'émulation. Elle consiste à exécuter l'application réelle dans un environnement virtuel pour permettre à l'expérimentateur de se placer dans les conditions expérimentales souhaitées. Ces approches, simulation et émulation, permettent donc d'étudier une application sous différentes conditions et ainsi obtenir différentes exécutions. Il est alors possible d'étudier la correction d'un système sur de plus nombreux scénarios expérimentaux, sans toutefois parvenir à garantir l'exhaustivité de l'étude. Il en résulte que ces approches ne permettent pas de pallier entièrement l'étude du non-déterminisme présent dans la majorité des systèmes distribués. La simulation et l'émulation ne sont donc pas suffisantes pour assurer la correction complète d'un système.

Pour répondre à cela, deux approches sont alors possibles. Il existe tout d'abord le test au sens du génie logiciel avec la mise en œuvre de cas de tests qui permettent de vérifier, statiquement ou dynamiquement, plusieurs scénarios d'exécutions possibles. La seconde approche est la vérification formelle [Bje05]. Celle-ci permet de prouver la correction d'un système d'après une spécification formelle de ce dernier en fournissant la preuve formelle d'une propriété. Cette

technique est notamment utilisée pour la vérification de protocoles cryptographiques, de circuits mais également de logiciels. La vérification formelle peut s'effectuer à travers la preuve de théorèmes, il s'agit alors de vérification déductive, ou par *model checking*.

2 Test logiciel

Le test logiciel est aujourd'hui considéré comme obligatoire en génie logiciel [Mat08] et fait partie intégrante des différentes étapes de conception. Il a généralement quatre objectifs appliqués au logiciel :

- la correction : capacité à donner des résultats corrects dans des conditions d'exécution normales ;
- les performances ;
- la fiabilité : aptitude à fonctionner même en cas d'évènements exceptionnels ;
- la sécurité : capacité à résister aux actions d'attaquants cherchant à nuire.

Nous nous intéressons ici à l'aspect correction. Dans ce cas, le test logiciel doit permettre de déterminer si le logiciel satisfait les spécifications, s'il fait ce pourquoi il a été conçu ou s'il contient des erreurs. Dans cette section, nous présentons le principe de cette vérification, les approches pour sa mise en œuvre et ses limites dans le cas de la vérification des systèmes distribués.

2.1 Principe

Le test logiciel est défini dans la littérature comme le processus consistant à exécuter un programme dans le but de trouver des erreurs [MSB11]. On dira alors qu'un test est réussi s'il trouve une ou plusieurs erreurs. Cette vérification se fait dès le début de la phase de conception du logiciel et ce jusqu'au rendu final. Selon l'état d'avancement du logiciel, l'objectif des tests sera différent ainsi que la technique employée.

Le test logiciel doit permettre à la fois de vérifier mais aussi de valider le logiciel. La validation assure qu'un logiciel respecte les besoins de l'utilisateur tandis que la vérification assure que le logiciel respecte les spécifications définies après l'analyse des besoins de l'utilisateur. Si les spécifications ne correspondent pas aux besoins de l'utilisateur, il est alors possible d'avoir un logiciel qui est vérifié mais non validé. Trouver des erreurs, c'est-à-dire des états d'exécution qui n'ont pas été spécifiés et qui ne sont donc pas supposés être atteints, et corriger les fautes à l'origine de ces dernières est inutile si le logiciel ne fait pas ce pourquoi il a été conçu. L'absence d'erreur ne signifie donc pas qu'il est valide, il est donc nécessaire de réaliser des tests complémentaires en ce sens.

Il existe deux catégories de tests : les tests statiques, qui consistent à analyser le système sans l'exécuter, et les tests dynamiques, durant lesquels le système testé est réellement exécuté. La vérification d'un logiciel est généralement assurée par des tests statiques tandis que les tests dynamiques ont pour objectif la validation. Toutefois, cette classification est basée sur l'apport maximal de chacune des approches, en ce sens que la validation d'un logiciel ne peut être réalisée sans l'exécuter mais elle nécessite également la vérification de ce dernier. Ainsi une approche statique ne permet que de vérifier tandis qu'une approche dynamique peut permettre à la fois de vérifier et de valider. Les tests statiques et les tests dynamiques restent cependant complémentaires car ils permettent d'identifier des problèmes différents.

2.2 Tests statiques

Les tests statiques peuvent se faire à partir du code source et, éventuellement, sur la documentation mais également directement sur le code binaire d'un programme. Ils permettent à la fois de détecter des fautes mais également de mettre en lumière des éventuelles ambiguïtés ou erreurs dans les spécifications ou la documentation du logiciel. Selon la technique mise en œuvre, ces tests vont de la vérification syntaxique à la vérification basée sur une analyse sémantique. Ils peuvent être classés en deux catégories : la revue de code et l'analyse statique de code.

2.2.1 Revue de code

Les principales techniques pour la réalisation de tests statiques sont basées sur la revue de code [CBDT06]. Cette approche consiste à examiner systématiquement le code source d'un logiciel. Elle peut être informelle, technique, se faire pas à pas ou à travers une inspection spécifique. Ces techniques peuvent faire intervenir les développeurs mais également d'autres personnes du projet. Dans le cas de revues techniques, il existe un processus complet allant de la planification de la revue à la documentation des résultats. La revue de code peut se faire pendant ou après son écriture, grâce notamment à la programmation par binôme. Dans cette technique, une personne joue le rôle de conducteur et écrit le code, tandis que l'autre observe et surtout relit le code écrit. La revue se fait donc à la volée.

Généralement manuelle, la revue de code peut aussi se faire de façon automatique à l'aide d'outils dans lesquels sont définies des règles ou de bonnes pratiques. On notera par exemple les greffons intégrés à des environnements de développement tels que Eclipse ou Visual Studio. Enfin, il existe également des assistants de revue de code [Rem05] tels que Gerrit [2], CodeStriker [3], Crucible [4] ou Review Board [5], qui permettent de structurer les revues, de faire de la revue de code collaborative avec commentaires et sauvegarde des résultats.

Malgré l'aide d'outils, la revue de code reste une opération assez lourde et souvent incomplète [ML09]. Avec un nombre moyen de cent cinquante lignes de code relues par heure, cette tâche peut rapidement devenir plus longue face à des systèmes complexes. Pour pallier cela, il existe une autre approche de test statique plus automatique : l'analyse statique de code[BBC+10] [ZWN+06].

2.2.2 Analyse statique de code

Cette approche est complémentaire avec la revue de code et est généralement réalisée avant cette dernière. Contrairement à la revue de code qui ne peut se faire que sur le code source (ou la documentation associée au logiciel), l'analyse statique peut être également faite sur le code binaire du logiciel [BJAS11] [BGRT05]. Elle est toujours réalisée à l'aide d'outils et peut également être incluse dans les compilateurs. Le premier à avoir été créé dans les années 1970 est l'utilitaire `lint` pour le langage C. Depuis, de nombreux outils ont été développés pour l'analyse de divers langages [6]. L'analyse réalisée par ces outils va de l'étude du comportement d'états individuels ou de déclarations jusqu'à une analyse complète pour la détection d'erreurs ou l'extraction d'informations. Les informations obtenues permettent à la fois d'établir des métriques

2. `http://code.google.com/p/gerrit/`
3. `http://codestriker.sourceforge.net/`
4. `https://www.atlassian.com/software/crucible/`
5. `https://www.reviewboard.org/`
6. Une liste d'outils existants est disponible à l'adresse `http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`

sur le logiciel mais également de réaliser des vérifications formelles.

2.3 Tests dynamiques

Bien que les outils d'analyse statique de code aient prouvé leur efficacité dans la détection d'erreurs, l'approche privilégiée dans le test logiciel reste les tests dynamiques [JSMHB13].

Contrairement aux tests statiques, le logiciel testé est réellement exécuté lors de tests dynamiques [Bei03]. Certaines erreurs, notamment celles liées à la gestion de la mémoire, peuvent être difficiles à détecter par une simple analyse du code, même détaillée. Il est donc nécessaire de réaliser une vérification *at run-time* par l'exécution réelle du système. Tester dynamiquement consiste à stimuler le système sous test en fournissant des entrées et à observer le résultat de son exécution en termes de sorties. Les observations réellement obtenues après exécution sont ensuite comparées avec celles attendues dans le cas d'un système correct. Pour cela, des cas de tests contenant des valeurs d'entrée et le résultat attendu pour chaque valeur sont définis.

Les tests dynamiques sont réalisés tout au long du cycle de développement du logiciel et peuvent être classés selon leur phase de réalisation. Il existe quatre catégories de tests dynamiques : les tests unitaires qui vérifient une partie de code spécifique indépendamment du reste du système, les tests d'intégration qui sont réalisés lors de l'intégration d'un nouveau code au sein du système existant, les tests système qui vérifient que le logiciel respecte les spécifications et, enfin, les tests acceptants qui assurent que le logiciel respecte les besoins définis par l'utilisateur ou le client.

La mise en œuvre de ces tests peut être automatisée de deux manières orthogonales : par la génération automatique des cas de tests et l'exécution automatique des tests.

2.3.1 Génération automatique

La génération automatique de tests consiste à générer un jeu de tests permettant de couvrir un critère lié à un flot de contrôle [Cla98]. Depuis plusieurs dizaines d'années, de nombreuses techniques [ABC+13] ont été développées, permettant cette génération. Elles peuvent êtres spécifiques à certains langages ou communes à plusieurs d'entre eux. Parmi elles, il y a l'exécution symbolique [Kin76], le test de couverture [HLL94], la génération de tests basée sur un modèle [UL10], le test combinatoire [NL11], le test aléatoire [Ham94] et enfin le test basé sur une recherche [McM04]. Ces approches peuvent être combinées pour une meilleure efficacité. Il existe également d'autres approches, en marge, telles que le test par mutation [JH11], qui modifie le code source ou binaire d'un programme pour ainsi forcer son exécution courante vers de nouvelles exécutions, ou bien le test à données aléatoires (*fuzzing* ou *fuzz testing*) [Sha07].

2.3.2 Exécution automatique

Une fois ces tests générés (ou créés manuellement), des outils automatisent leur exécution et leur analyse. Il est ainsi possible de les exécuter rapidement et surtout de façon répétée. Généralement, cette approche permet de mettre en œuvre des tests de non-régression qui assurent la pérennité du fonctionnement du logiciel pendant son développement dans la même optique que l'intégration continue. Parmi les outils existants, on notera TestComplete et QuickTestPro [KK11].

Qu'ils soient statiques ou dynamiques, les tests ne permettent donc pas de réaliser une vérification exhaustive. De plus, leur mise en œuvre permet de détecter d'éventuelles erreurs mais pas de démontrer l'absence de ces dernières. Ces limites amènent alors à considérer une autre approche : la vérification formelle.

3 Vérification formelle déductive

La vérification formelle déductive consiste à produire, à partir de la description d'un système et de ses spécifications, une propriété vérifiée par ces deux éléments. L'objectif est donc de déduire des propriétés propres à un système d'après ses spécifications, c'est-à-dire ce qu'il est censé faire, et de prouver que sa description, c'est-à-dire ce qu'il fait réellement, les vérifie. Cette preuve peut être déduite à la main suite à une étude approfondie du système, dans un langage naturel. Toutefois, cette technique tend aujourd'hui à être considérée comme trop informelle notamment en raison des ambiguïtés du langage utilisé. Plusieurs approches, en partie ou entièrement automatisées, permettent alors de pallier cela et d'assurer la correction d'une preuve : les assistants de preuves, la preuve automatique de théorèmes et les solveurs SAT/SMT.

3.1 Assistant de preuve

Un assistant de preuve est un logiciel qui permet d'écrire formellement et de vérifier des preuves mathématiques sur des théorèmes ou sur des assertions relatives à l'exécution d'un programme. L'écriture de preuves formelles peut être extrêmement fastidieuse au point d'omettre certaines informations considérées comme évidentes pour un lecteur familier des mathématiques. La vérification de ces preuves devient alors complexe voire impossible. Grâce à un assistant de preuve, il est possible d'automatiser en partie cette tâche et ainsi de renforcer la confiance dans une preuve.

L'objectif est de mettre à la disposition du mathématicien, ou tout autre utilisateur souhaitant écrire une preuve, un système informatique interactif et un formalisme de démonstration qui lui permettent d'élaborer une version formelle du résultat auquel il s'intéresse en interagissant avec l'assistant de preuve. L'outil libère ainsi l'utilisateur d'une partie du travail d'écriture et réalise les opérations nécessaires à chaque étape de l'explication d'une démonstration formelle. De plus, l'assistant de preuve va stocker les résultats démontrés formellement précédemment pour les rendre ainsi réutilisables pour la démonstration de nouvelles preuves. Parmi les 100 théorèmes mathématiques les plus importants, 85 ont déjà pu être formalisés grâce à des assistants de preuve [Del11]. Parmi les outils existants, on notera Coq [HH14] et Isabelle [NPW02] (successeur de HOL).

3.2 Preuve automatique de théorème

La démonstration de la correction d'une preuve peut être réalisée automatiquement et entièrement grâce à la preuve automatique de théorème [Pla14]. Contrairement aux assistants de preuves qui nécessitent un guidage humain, un prouveur automatique est capable de démontrer une propriété sans l'aide d'un utilisateur. La preuve formelle est entièrement construite par l'outil d'après une description du système, un ensemble d'axiomes logiques et de règles d'inférence.

Les techniques les plus populaires pour réaliser cette preuve automatique sont la méthode des tableaux sémantiques [DG99] qui consiste à réfuter une formule en la décomposant en sous-formules atomiques, la réécriture de termes [BN99] et les diagrammes de décision binaire [Ake78] qui permettent de représenter des fonctions booléennes.

Pour une description approfondie de la preuve automatique de théorème et des techniques mises en œuvre, nous invitons le lecteur à se référer à [Har13].

3.3 Solveurs SAT/SMT

Les assistants de preuve et les outils de preuve automatique de théorèmes peuvent être combinés avec des solveurs SAT/SMT (*SATisfiability problem / Satisfiability Modulo Theories*).

Le problème de satisfaisabilité [DGP+97] est un problème de décision défini par des formules logiques qui détermine si une formule possède une solution, c'est-à-dire s'il existe des valeurs rendant la formule vraie. La satisfaisabilité modulo théories [DMB11] est un autre problème de décision qui détermine la satisfaisabilité d'une formule logique par rapport à des théories sous-jacentes exprimées dans la logique classique du premier ordre.

Ces techniques sont très populaires dans le domaine de la preuve automatique pour leur capacité à combiner efficacement des procédures de décisions pour différents fragments logiques. Parmi les outils existants, nous pouvons citer veriT [BDODF09], Z3 [DMB08], SMTInterpol [CHN12] et CVC4 [BCD+11].

Quelle que soit la technique mise en œuvre, la vérification déductive nécessite des connaissances formelles importantes qui peuvent rapidement freiner son utilisation. Le *model checking* est une autre approche de vérification formelle qui permet d'établir la correction d'un système. Au lieu d'essayer de démontrer qu'un système satisfait une propriété, l'approche consiste à rechercher un état qui ne la satisfait pas.

4 *Model checking*

Le *model checking* [CGP99] est une technique de vérification formelle automatique permettant de déterminer si un modèle donné (le système lui-même ou une abstraction de celui-ci) satisfait une spécification donnée, souvent formulée en logique temporelle. Cette approche est basée sur des méthodes d'exploration de l'espace d'états du système pour démontrer qu'un état (ou un ensemble d'états) indésirable(s) est accessible ou inaccessible. Pour cela, l'ensemble des exécutions possibles est déterminé d'après la description du système, à partir d'un état initial donné. L'exploration s'effectue ensuite jusqu'à ce qu'une violation de propriété soit détectée ou bien jusqu'à exploration complète de l'espace d'états.

L'avantage majeur du *model checking* est qu'il fournit un contre-exemple lorsqu'une propriété n'est pas satisfaite. Celui-ci correspond à l'exécution qui mène à l'état (ou l'ensemble d'états) violant la propriété spécifiée. Cette technique de vérification est donc plus intéressante en pratique que les approches de vérification déductive basées sur les preuves mathématiques classiques, tout en assurant une preuve formelle de la correction du système étudié.

4.1 Principe

Le *model checking* s'effectue habituellement en trois phases : (1) Modélisation du système et formulation de la propriété, (2) Vérification par exploration exhaustive et (3) Analyse des résultats.

Dans un premier temps, le système étudié est donc modélisé sous forme d'un système de transitions à états et la spécification à vérifier sous forme d'une propriété. Chaque état du système de transitions correspond à un pas d'exécution du système et chaque transition à une évolution possible d'un état donné vers un autre, selon l'action exécutée par le système. Ce

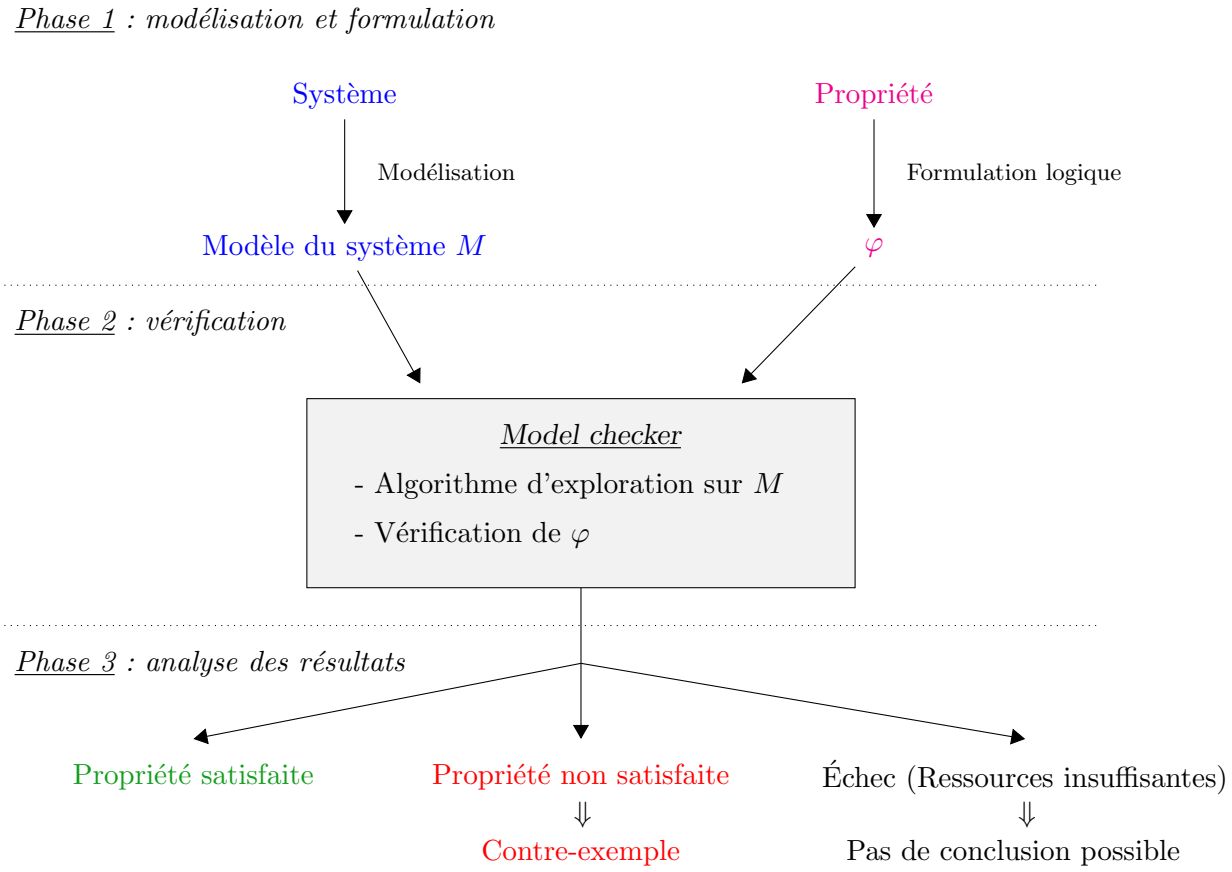

Figure 2.1 – Principe du *model checking*.

système de transitions peut être représenté par un graphe orienté, un automate à états fini, une machine de Turing ou un réseau de Petri.

Définition 4 (Système de transitions à états). Un système de transitions à états S est un 4-uplet $S = (Q, I, T, \rightarrow)$ tel que :

- Q est l'ensemble des états ;
- $I \subseteq Q$ est l'ensemble des états initiaux ;
- T est l'ensemble des transitions ;
- $\rightarrow \subseteq Q \times T \times Q$ est une relation de transition entre états qui vérifie : $\forall q \in Q$, $\exists q' \in Q$ et $\exists t \in T$, tels que $(q, t, q') \in \rightarrow$.

Une transition $t \in T$ est dite *exécutable* si et seulement s'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 $(q_i, t_i, q_{i+1}) \in \rightarrow$ pour un certain $t_i \in T$. Enfin, 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 systèmes non-déterministes, 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' et 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. Enfin, l'exécution est séquentielle, c'est-à-dire que chaque processus ne peut exécuter qu'une seule action à la fois.

En *model checking*, les systèmes de transitions à états sont en plus munis d'une fonction d'étiquetage pour les états, donnant alors une structure de Kripke [BCG87]. Il s'agit d'un graphe orienté dont les nœuds représentent les états accessibles du système et les arcs représentent les transitions entre les états. De plus, la fonction d'étiquetage fait correspondre à chaque état un ensemble de propositions logiques vraies (propositions atomiques) dans cet état.

Définition 5 (Structure de Kripke). Soit AP un ensemble de propositions atomiques, c'est-à-dire des expressions booléennes portant sur des variables, des constantes et des prédicats. Une structure de Kripke est un 4-uplet $M = (S, s_0, R, L)$ tel que :

- S est un ensemble d'états fini ;
- $s_0 \in S$ est un état initial ;
- $R \subseteq S \times S$ est une relation de transition entre états qui vérifie : $\forall s \in S$, $\exists s' \in S$ tel que $(s, s') \in R$;
- $L : S \to 2^{AP}$ est une fonction d'étiquetage.

Dans un second temps, le *model checker* vérifie si le modèle M obtenu satisfait la propriété φ. Pour cela, il effectue une exploration exhaustive de l'espace d'états. Il s'agit de l'ensemble des états d'un système de transitions accessibles à partir d'un état initial en suivant les chemins du système de transitions. Il a une structure plus simple que le système de transitions complet puisque l'on ne garde que les états en omettant la structure du graphe.

Enfin, une analyse des résultats obtenus est effectuée : si M vérifie φ, le modèle du système est déclaré correct. Si le modèle ne satisfait pas la propriété, l'outil de vérification retourne un contre-exemple qui est rejoué sur le système réel. Deux cas de figures sont alors possibles : le rejeu du contre-exemple sur le système réel engendre effectivement une erreur ou bien, la violation de propriété rapportée par l'outil n'est présente que sur le modèle si bien qu'aucune erreur n'est soulevée lors du rejeu. Cela signifie alors que le modèle ne représente pas correctement le système réel, il doit donc être raffiné avant de réaliser une nouvelle vérification.

4.2 Explosion combinatoire de l'espace d'états

En pratique, une limitation majeure du *model checking* est la taille des systèmes étudiés. Si on considère un système de n processus qui possèdent chacun m états, l'espace d'états à explorés correspondant est alors de m^n états. Il y a donc une augmentation exponentielle du nombre d'états à explorer selon le nombre de processus et les actions exécutées par ces derniers au sein du système étudié. Ce phénomène est appelé explosion combinatoire de l'espace d'états [CKNZ12].

La gestion de cette explosion a fait l'objet de nombreux travaux depuis l'émergence du *model checking* dans les années 1980. Plusieurs techniques ont ainsi été développées pour la pallier. Leurs objectifs sont de réduire la taille des modèles en créant des modèles équivalents plus petits mais également de représenter les états et les transitions plus efficacement pour ne vérifier ensuite que les états pertinents vis-à-vis de la propriété. Pour cela, les trois approches majeures sont le *model checking* symbolique (qui englobe le *model checking* borné), les techniques de réduction et l'abstraction.

4.2.1 *Model checking* symbolique

Proposé par Ken McMillan en 1992 dans sa thèse, le *model checking* symbolique [CMCHG96] fait référence à l'ensemble des techniques qui représentent de façon symbolique (implicite) les états et les transitions du système. Cette représentation permet ensuite de les regrouper sous forme de fonctions booléennes. Dès lors, l'exploration ne se fait pas état par état mais en considérant plusieurs états et transitions en une seule étape.

Les techniques de *model checking* symbolique sont généralement classées en deux catégories : celles basées sur les diagrammes de décision binaire (BDDs) et celles basées sur les procédures SAT. Historiquement, les BDDs sont associés à la vérification de propriétés CTL tandis que les procédures SAT sont utilisées pour la vérification de propriétés LTL.

4.2.1.1 Diagrammes de décision binaires (BDDs)

Un diagramme de décision binaire est une structure de données compacte utilisée pour représenter l'arbre sémantique associée à une formule, et donc en particulier toutes les interprétations qui la satisfont. Les BDDs prennent donc en entrée n variables booléennes qui composent la formule étudiée et retournent un booléen en sortie. La fonction booléenne associée à la propriété y est représentée par un graphe orienté acyclique avec une racine, des nœuds de décision et deux nœuds terminaux appelés 0-terminal et 1-terminal. Chaque nœud de décision est étiqueté par une variable booléenne et a deux nœuds fils, appelés fils bas et fils haut. L'arête d'un nœud vers le fils bas correspond à une variable à 0 et à 1 pour le fils haut. L'utilisation d'un graphe orienté acyclique plutôt qu'un arbre permet de réduire de façon efficace la taille du modèle en supprimant les redondances et en partageant les sous-arbres isomorphes.

Si cette réduction est maximale et que les variables partant de la racine jusqu'aux feuilles sont toujours dans le même ordre sur l'ensemble des chemins, un diagramme de décision binaire ordonné (OBDD) est alors obtenu, avec canonicité de la représentation par rapport à la propriété (il y a un seul OBDD pour une formule donnée). Des heuristiques sont mises en œuvre pour obtenir cette ordonnancement unique des variables pour l'ensemble de la structure. Une fois le BDD (ou OBDD) construit, il suffit de vérifier si la propriété est vérifiée d'après les valeurs d'entrées en état initial. nuXmv [CCD+14] (successeur de nuSmv et SMV) est le *model checker* symbolique basé sur les BDDs le plus populaire.

4.2.1.2 Procédures SAT

Dans le cas où le BDD obtenu n'est pas ordonné, la représentation correspondante n'est pas canonique rendant alors l'approche inefficace. L'évaluation du bon ordonnancement pour obtenir un BDD le plus petit possible peut exiger du temps et parfois une intervention manuelle. Il n'est cependant pas garanti d'obtenir la meilleure représentation, notamment lors de la vérification de propriétés LTL. Pour pallier cela, de nouvelles techniques basées sur les procédures SAT ont été proposées.

Également basée sur l'utilisation de formules booléennes, cette approche ne repose toutefois pas sur la canonicité de la représentation. Ainsi, il est possible de travailler avec plusieurs ordonnancements des variables sur différents chemins sans subir l'explosion de la représentation connue avec les BDDs. L'approche la plus connue dans ce domaine est le *model checking* borné [CBRZ01]. Le principe est de construire une formule booléenne qui est satisfaite si et seulement s'il existe un contre-exemple de longueur bornée. La recherche de celui-ci se fait par itérations jusqu'à atteindre la borne définie. Cela permet ainsi d'obtenir un contre-exemple le plus petit

possible et donc plus rapidement. De plus, des heuristiques permettent de réaliser cette recherche de façon automatique sans nécessiter une intervention manuelle contrairement aux BDDs. BMC [BCCZ99] est le *model checker* de référence dans cette approche.

4.2.2 Réduction de l'exploration

La seconde technique pour faire face à l'explosion combinatoire de l'espace d'états est de détecter les redondances dans les différents chemins d'exécution du système pour ainsi ne parcourir qu'une seule occurrence de celles-ci.

Dans le cas de la réduction par ordre partiel [God91], le principe est de détecter des classes d'équivalences entre les différentes exécutions pour ne parcourir qu'une seule d'entre elles. Cette détection est basée sur l'évaluation de la dépendance entre les transitions qui sont exécutées dans des ordres différents selon les exécutions. Des transitions sont déclarées indépendantes si leur ordre d'exécution n'influe pas sur l'état résultant. Il est ainsi possible de ne parcourir qu'un seul ordre d'exécution tout en assurant une exhaustivité de l'exploration de l'espace d'états. Il existe plusieurs versions de cette technique de réduction : la méthode basée sur les *stubborn set* [Val91], la méthode avec *ample set* [Pel93] et la méthode avec *persistent set* [GvLH$^+$96]. Initialement réalisée à partir du modèle complet du système, cette approche a depuis été appliquée de façon dynamique [FG05] lors de la construction et la vérification du modèle à la volée.

Sur le même principe, il existe également la réduction par symétries [CEFJ96] qui détecte les ensembles d'états symétriques en les remplaçant dans la structure de Kripke par un seul ensemble de chaque classe d'équivalence. Le modèle résultant est appelé structure quotient. L'idée dans cette approche est de ne pas construire le modèle original du système mais de représenter à la volée la structure quotient correspondante grâce à l'identification statique de groupes de symétries. SymmExtractor [DM05] et TopSPIN [DM06] permettent de détecter automatiquement ce type de symétries et d'extraire la structure quotient résultante. Cette approche reste toutefois plus appropriée au *model checking* explicite qu'au *model checking* symbolique. En effet, appliquer la réduction par symétries dans le contexte des BDDs nécessite de sauvegarder les classes d'équivalence, ce qui peut se révéler extrêmement coûteux.

4.2.3 Abstraction

La dernière catégorie de techniques pour pallier l'explosion combinatoire de l'espace d'états repose sur la génération d'une abstraction du modèle [CGL94] pour ainsi obtenir un espace d'états plus petit. Généralement, l'abstraction obtenue ne satisfait pas exactement les mêmes propriétés que le modèle original, un raffinement est donc nécessaire. Pour cela, il existe l'approche CEGAR (*CounterExample-Guided Abstraction Refinement*) [CGJ$^+$00] qui consiste à vérifier une propriété donnée sur une abstraction du modèle, en le raffinant au fur et à mesure de la vérification grâce aux contre-exemples obtenus. Une première vérification est réalisée sur l'abstraction qui est généralement incomplète. Un contre-exemple est donc fourni. Ensuite, ce contre-exemple est rejoué sur le système réel pour déterminer s'il s'agit d'une vraie violation de propriété ou bien si l'erreur provient de l'abstraction en elle-même. Ainsi, l'abstraction est affinée de façon itérative jusqu'à obtenir un contre-exemple présent à la fois sur celle-ci et sur le système réel. Le *model checker* BLAST [BHJM07] implémente une version optimisée de l'approche CEGAR appelée *lazy abstraction* [HJMS02]. Les étapes de calcul de l'abstraction du modèle et de son raffinement sont optimisées grâce à la sauvegarde des opérations effectuées lors de chaque itération. Ainsi, l'abstraction est construite à la volée et localement raffinée si besoin.

5 *Software model checking*

Comme son nom l'indique, le *model checking* est initialement destiné à vérifier des modèles. En particulier, cette approche a été appliquée dans un premier temps à la vérification des circuits électroniques. Une description formelle en HDL (*Hardware Description Language*) permet d'obtenir facilement un modèle sous la forme de machine à états finie qui est ensuite fournie aux *model checkers* permettant ainsi une vérification automatique. Rapidement, cette technique a été appliquée à d'autres domaines tels que la vérification de programmes. Cependant, un modèle du système étudié est toujours requis. Il est donc nécessaire de construire manuellement ce modèle d'après la description du programme. Cette opération pose deux problèmes majeurs. La construction manuelle du modèle demande un temps et une expertise considérables qui augmentent bien sûr avec la taille du système étudié mais également avec sa complexité. De plus, alors que cette construction peut prendre plusieurs semaines voire plusieurs mois, le programme peut continuer à évoluer pendant ce temps rendant le modèle rapidement obsolète vis-à-vis de l'état d'avancement courant dans l'implémentation du programme.

Un nouvel axe de recherche est donc apparu dans les années 1990 : le *software model checking* [HS99]. L'enjeu est d'appliquer les techniques de *model checking* courantes à la vérification de programmes, bien plus complexes que les modèles étudiés jusqu'à présent. Il en résulte deux approches principales : l'extraction automatique d'un modèle abstrait et la vérification basée sur l'exécution systématique du programme réel. Cette seconde approche constitue le cœur des travaux présentés dans ce document.

5.1 Extraction automatique de modèle

5.1.1 Principe

L'extraction automatique d'un modèle est réalisée depuis le code source du programme étudié grâce à une analyse statique (présentée en section 2.2) ou une exécution symbolique (présentée en section 4.2). La vérification par *model checking* est ensuite réalisée sur le modèle abstrait obtenu. Pour plus d'efficacité, la précision de l'abstraction par rapport au programme est souvent peu optimisée. Généralement, seule une partie des informations d'exécution est capturée en utilisant une sémantique abstraite du programme [CC77]. Le choix du domaine d'abstraction mais également de la sémantique assure donc la correction de la vérification (les erreurs détectées sont présentes dans le modèle abstrait mais également dans le programme réel) et son efficacité.

5.1.2 Outils existants

Le *model checker* SLAM [BLR11] est le premier à avoir implémenté l'extraction automatique avec raffinement par contre-exemples pour des programmes C. Pour cela, il s'appuie sur l'utilisation de programmes booléens, c'est-à-dire des programmes impératifs où chaque variable est un booléen, comme langage intermédiaire pour représenter les abstractions. Le programme C à vérifier est fourni en entrée de l'outil c2bp avec un ensemble de prédicats. Un programme booléen est généré en sortie, dans lequel chaque variable correspond à un prédicat. Les affectations et conditions sur celles-ci correspondent au prédicat cartésien abstrait du programme C. Ensuite, le programme généré est soumis au *model checker* symbolique BEBOP. Le raffinement de l'abstraction est quant à lui réalisé grâce à l'outil newton.

BLAST implémente l'approche « d'abstraction paresseuse » (*lazy abstraction*), présentée en section 4.2, pour la génération d'un modèle abstrait à partir d'un programme C. Il permet ainsi de vérifier des propriétés de sûreté sur la mémoire en prouvant statiquement, grâce à

l'analyseur CCURED [NCH+05], que le programme vérifie la propriété ou, en fournissant un chemin d'exécution comme contre-exemple en cas de violation. Il peut à la fois être utilisé pour la vérification de programmes mais également pour la génération automatique de tests (présentée en section 2).

MAGIC [CCG+04] est également conçu pour la vérification modulaire de programmes C, en particulier les programmes concurrents par passage de messages. Il permet de spécifier un système de transitions étiqueté quelconque et de vérifier qu'un ensemble de traces d'exécutions générées par le programme étudié est contenu dans le langage généré par le système de transition étiqueté. Il a, depuis 2004, été abandonné au profit du *model checker* Copper inclus dans le *framework* ComFoRT [CISW05]. Celui-ci est essentiellement basé sur l'approche CEGAR présentée en section 4.2 tout comme le *model checker* F-Soft [ISG+05].

Enfin, le *model checker* SPIN [Hol04] implémente l'outil AX (Automaton eXtractor) [HHS01] qui permet de vérifier des propriétés LTL sur des systèmes distribués écrits en ANSI-C. Pour cela, il extrait un modèle avec un niveau d'abstraction défini par l'utilisateur, qui est ensuite vérifié à l'aide de SPIN. Celui-ci implémente la plupart des techniques de vérification de modèle abstrait.

Bien que cette approche soit de plus en plus précise, des difficultés subsistent pour construire un modèle complet et cohérent avec le programme réel. L'approche pour pallier cela est de vérifier, toujours par *model checking*, l'implémentation réelle du programme à travers son exécution systématique.

5.2 Exécution systématique

5.2.1 Principe

La vérification par exécution systématique construit et explore l'espace d'états du système à la volée, à travers son exécution réelle. Dans cette approche, l'exploration s'effectue pas à pas sur l'ensemble des exécutions possibles en exécutant simplement les processus ordonnancés.

L'avantage majeur de cette technique est que la phase de formalisation du système étudié est totalement abandonnée. Il n'est plus nécessaire de déterminer la sémantique du langage de programmation et les instructions pour définir la relation de transition. Chaque état correspond à l'état courant en mémoire du programme exécuté tandis que chaque transition correspond alors simplement à une action exécutée par un processus. Pour cela, le contrôle de l'exécution et son observation sont, par exemple, réalisés par l'interception des appels systèmes effectués par le système étudié. Dès lors, le *model checker* est capable de guider l'exécution du système à travers les différents scénarios possibles pour ainsi construire dynamiquement l'espace d'états et le vérifier de façon exhaustive. Dans ce cas, le modèle, à proprement parler, est exploré en pratique mais reste toutefois inaccessible avant l'exploration.

De plus, en cas de non satisfaction de la propriété à vérifier, l'outil fournit un contre-exemple généré depuis l'exécution réelle du système. Il est ainsi possible de le reproduire, grâce notamment à l'utilisation de *debuggers*, et ainsi de trouver plus rapidement la faute à l'origine de l'erreur détectée. Cependant, contrairement au test classique (présenté en section 2), le *model checker* est ici capable d'explorer systématiquement l'ensemble des exécutions possibles pour un seul état initial donné. C'est ainsi que nous parlons de Vérification dynamique formelle.

5.2.2 Difficultés

Cette approche pose toutefois deux difficultés majeures. Sachant que l'on travaille sur le programme réel, un état est composé de la mémoire complète des processus, c'est-à-dire la pile, le tas, les données globales ainsi que l'ensemble des informations sur l'état du réseau. Ceci est donc extrêmement grand si bien que la sauvegarde des états visités au cours de la vérification est difficilement réalisable en pratique.

C'est dans ce contexte que le *model checker* Verisoft [God05] a proposé pour la première fois une exploration sans conservation des états visités (*Stateless search*). Pour cela, il s'appuie sur le fait que chaque exécution est caractérisée par l'ordonnancement des processus exécutés. Il est ainsi possible de parcourir différentes exécutions en traitant simplement les différents ordonnancements possibles. L'algorithme correspondant consiste à parcourir itérativement en profondeur l'espace d'états jusqu'à une borne définie en termes de profondeur. Cette profondeur correspond au nombre de transitions exécutées durant l'exécution courante. Une fois toutes les exécutions de longueurs égales à la profondeur maximale explorées, la borne est incrémentée et le processus d'exploration et d'incrémentation de la borne est de nouveau réalisé jusqu'à détection d'une violation de propriété. Pour réaliser le changement de chemin et ainsi explorer une nouvelle exécution, l'état initial est sauvegardé au début de la vérification (il s'agit du seul état sauvegardé) et est donc restauré pour la réexécution depuis celui-ci. Cependant, en l'absence de sauvegarde des états visités, il est nécessaire d'adapter l'ensemble des techniques de réduction présentées précédemment. En complément de l'exploration sans sauvegarde des états, Godefroid *et al.* [FG05] ont également été les premiers à proposer la Réduction Dynamique par Ordre Partiel (DPOR) qui permet d'appliquer la réduction présentée en section 4.2, même sans la sauvegarde de l'ensemble des états.

De plus, les autres techniques de réduction étant essentiellement basées sur l'identification d'états identiques, cette détection devient extrêmement complexe dans le cas de programmes réels. Il est nécessaire d'accéder à l'ensemble des informations qui composent un état, ce qui peut se révéler plus ou moins facile selon le langage de programmation, mais il faut également parvenir à les analyser en un temps informel, c'est-à-dire sans influer sur le temps de vérification. L'une des approches les plus classiques pour pallier cela est d'utiliser une fonction de hachage pour la sauvegarde des états au lieu de stocker l'état complet. Non seulement cette technique permet de limiter la consommation mémoire mais elle rend également la comparaison plus efficace et donc la détection des états identiques plus rapide. Toutefois, le risque de collision des *hash* rend cette approche incertaine quand le nombre d'états augmente. De plus, la construction de ce *hash* est parfois impossible à réaliser sur la globalité de la mémoire, certaines informations n'étant pas directement accessibles et identifiables selon le langage de programmation du système étudié.

5.2.3 Outils existants

La vérification par *model checking* basée sur l'exécution réelle du programme constitue un axe majeur de recherche en vérification de programmes. Bien que plusieurs concepts issus du *model checking* puissent être appliqués facilement, cette approche nécessite de nouvelles techniques, tant pour la vérification en elle-même des propriétés que pour la gestion de l'explosion combinatoire de l'espace d'états.

Comme évoqué précédemment, Verisoft est un des précurseurs dans ce domaine. Il s'intéresse en particulier aux programmes C et C++ implémentant des protocoles de communication et permet de vérifier des propriétés de sûreté telles que les *deadlocks, livelocks* et assertions.

CMC [MPC+02] permet également la vérification de propriétés de sûreté sur le même type de programmes mais en sauvegardant une représentation canonique des états visités (*Stateful search*) ce qui lui permet de détecter les états identiques et ainsi réduire l'espace d'états. Cette représentation lui permet alors de ne stocker que l'empreinte (le *hash*) des états au lieu de l'état complet et ainsi réduire la consommation mémoire. En 2007, l'un des créateurs de CMC a proposé le projet CHESS [MQ07] qui permet la vérification de programmes Windows *multithreadés*. Tout comme Verisoft et CMC, CHESS se base sur l'interception des appels systèmes pour l'exploration exhaustive de l'espace d'états. Cependant, il propose un nouveau concept essentiellement basé sur la présence de *multithreading* dans les programmes vérifiés : la définition d'une borne itérative liée au changement de contexte entre *threads*. L'idée est donc de définir une borne initiale puis de parcourir l'ensemble des exécutions ne dépassant pas cette borne en l'incrémentant de façon itérative. Cette approche repose notamment sur le constat que de nombreuses erreurs se manifestent lors de longues exécutions mais avec peu de changements de contextes.

Depuis Verisoft, différents outils ont été développés permettant la vérification d'autres langages de programmation. Java PathFinder (JPF) [VHB+03] est le *model checker* de référence pour la vérification de programmes Java. Il permet de vérifier des propriétés de sûreté sur du *bytecode* Java mais également des propriétés LTL grâce au traducteur LTL2JPF [NK10]. Pour cela, il modifie la machine virtuelle Java (JVM) permettant ainsi de rechercher systématiquement les différents ordonnancements possibles de programmes *multithreadés*. Cette exploration est réalisée avec sauvegarde des états visités, ce qui permet de réduire l'espace d'états à travers différentes techniques telles que la réduction par ordre partiel, la détection des symétries ou l'abstraction. De plus, cet outil utilise des heuristiques pour déterminer les différentes exécutions possibles ainsi que les techniques mises en œuvre dans l'exécution symbolique avec abstraction. Initialement, JPF permettait simplement d'extraire un modèle en PROMELA depuis le *bytecode* Java [Hav99] pour ensuite réaliser une vérification à l'aide de SPIN, tout comme Bandera [HD01].

MaceMC [KAJV07] se distingue des autres *model checkers* car il prend en entrée une description en langage Mace du système distribué et réalise ensuite la vérification directement sur cette description ou sur une version compilée de celle-ci en C++. Il exploite le fait que les programmes écrits en MACE sont structurés sous forme de systèmes de transitions à états pour réaliser l'exploration. Il permet de vérifier des propriétés de sûreté en combinant une recherche sans sauvegarde des états visités et une fonction de hachage, ainsi que des propriétés de vivacité exprimées en LTL sous la forme d'invariants ($\Box\Diamond p$, où p est une propriété sans quantificateur). Les propriétés de vivacité sont vérifiées grâce à une heuristique qui détermine les chemins qui peuvent violer la propriété. Lorsqu'un état dans lequel la propriété est fausse est détecté, l'exploration est effectuée à partir de celui-ci en recherchant un état dans lequel la propriété devient vraie. Si aucun état correspondant n'est trouvé, alors l'exécution courante est considérée comme contre-exemple. Pour rendre plus efficace cette détection face à des espaces d'états très grands, une heuristique recherche les transitions critiques qui mènent le système vers un chemin dans lequel la propriété sera violée de façon permanente. Ensuite le parcours à partir de cette transition est réalisé de façon aléatoire, ce qui rend l'approche imparfaite bien que de nombreuses erreurs ont pu être trouvées dans divers algorithmes grâce à celle-ci. Pour pallier l'explosion combinatoire, MACE implémente l'outil CrystalBall [YKKK09] qui permet à chaque processus de prédire les conséquences de ses actions pour ainsi détecter les erreurs avant même leur exécution. L'idée est donc de placer le *model checker* directement sur des états depuis lesquels une erreur est prédite et d'explorer le chemin qui en découle. Pour cela, chaque nœud du système exécute un algorithme d'exploration en continu depuis un état récemment sauvegardé et prédit donc les futures violations de la propriété spécifiée.

Enfin, nous notons également l'outil ISP [VSGK08] qui permet la vérification de systèmes

distribués HPC écrits en MPI. Tout comme les outils précédents, celui-ci se base sur l'interception des appels réalisés par le système étudié, en particulier les appels MPI à travers l'interface PMPI. Grâce à un *profiler* interne, ISP réécrit les appels MPI en modifiant les paramètres ou les délais d'exécution des processus pour ainsi réaliser une exploration des différentes exécutions possibles. L'explosion combinatoire de l'espace d'états est gérée grâce à DPOR adaptée à la sémantique des appels MPI. Une version distribuée de l'outil est proposée avec DAMPI [VAG+10]. S'agissant de l'outil le plus proche en lien avec nos travaux, une présentation technique détaillée de celui-ci est réalisée au chapitre suivant.

6 Limites des approches présentées

« *Program testing can be used to show the presence of bugs, but never to show their absence !* » (Dijkstra, 1970). Bien que l'objectif principal du test logiciel soit la détection d'erreurs, celui-ci ne garantit pas l'exhaustivité de la détection. Dès lors, celui-ci permet uniquement d'affirmer que toutes les erreurs connues ont été détectées. Il est impossible de déterminer s'il existe des erreurs non détectées. Face à l'impossibilité de garantir une exhaustivité de vérification, le test logiciel ne permet pas d'établir qu'un système fonctionne correctement sous toutes les conditions mais éventuellement qu'il ne fonctionne pas dans certains cas. L'utilisation de techniques telles que la couverture de code aide toutefois à s'assurer qu'un ensemble adéquat des comportements possibles a été observé, mais ceci reste insuffisant, particulièrement dans le contexte des systèmes distribués avec des processus concurrents. Le test logiciel permettra d'évaluer le comportement du système pour différentes valeurs d'entrées mais il ne permet pas de vérifier toutes les exécutions possibles pour une seule valeur d'entrée.

De plus, il est nécessaire d'assurer la maintenance des tests face à un système qui évolue durant son développement. De même, les tests sont créés d'après les spécifications du système. Des spécifications incomplètes, ambiguës ou simplement modifiées au cours du développement peuvent alors mener à des tests inadéquats. Il n'est toutefois pas toujours évident d'établir un tel diagnostique, le développement du logiciel peut alors continuer jusqu'à un point critique pour la découverte de ce type de problème.

À l'opposé, la vérification formelle permet de gérer l'aspect exhaustif qui fait défaut au test logiciel. La vérification déductive, à travers la preuve de théorèmes, ainsi que le *model checking* classique, nécessite cependant une formalisation du système étudié afin de générer un modèle qui sera soumis à vérification. Cette contrainte pose alors deux difficultés majeures qui limitent l'utilisation de ces approches.

Tout d'abord, selon la complexité du système, sa taille et son langage de programmation, extraire un modèle du système est parfois impossible à réaliser correctement, même à l'aide d'outils automatiques. Dans le cas où un modèle a pu être extrait, celui-ci correspond à une abstraction plus ou moins précise. Il est alors courant que le résultat obtenu soit une approximation du comportement du programme réel.

De plus, certains aspects du système étudié peuvent être particulièrement complexes à modéliser, même à l'aide d'une analyse statique sur le code source, certaines informations ne pouvant être connues que lors de l'exécution. Dès lors, il persiste l'incertitude que la vérification est faite sur un modèle du système et non sur l'implémentation réelle de ce dernier. L'éventuelle correction établie d'après cette vérification peut-elle alors être garantie également sur le système réel ? La seconde limite réside dans la nécessité de reconstruire un modèle après chaque modification du système afin de garantir la cohérence entre les deux entités. Selon les modifications

réalisées, cette reconstruction peut concerner une petite partie du modèle ou son ensemble, ce qui peut être coûteux à mettre en œuvre régulièrement.

La vérification dynamique formelle, à travers l'exécution systématique du système réel, permet de ne pas se soumettre à cette contrainte, en travaillant directement avec l'implémentation réelle du système. L'approche est particulièrement adaptée à l'étude des systèmes distribués car elle tient compte du non-déterminisme dans l'exécution de ces derniers mais elle permet également de travailler sur des langages de programmation tels que le C/C++ ou Fortran, majoritairement présents dans le domaine mais souvent difficiles à modéliser. Cependant, travailler sur le programme réel, et non un modèle simplifié de celui-ci, implique de prendre en compte l'ensemble des informations qui le composent. Il est alors nécessaire de gérer leur sauvegarde mais également leur analyse. Les outils existants mettant en œuvre cette approche prennent souvent le parti de limiter les informations sauvegardées au cours de l'exécution ce qui limite fortement le type de vérification pouvant être réalisée. Notamment dans le cas des programmes C/C++ ou Fortran, soit seules les propriétés de sûreté sous forme d'assertions peuvent être vérifiées, soit une catégorie spécifique de propriétés de vivacité exprimées sous forme d'invariants. La correction des systèmes distribués basée sur ces vérifications s'en trouve alors fortement restreinte.

C'est dans ce contexte que nous présentons dans ce document des travaux qui mettent en œuvre de nouvelles techniques permettant d'étendre le type de vérifications réalisées à travers cette dernière approche.

Chapitre 3

Contexte technique

Dans ce chapitre, nous présentons le contexte technique des travaux exposés dans ce document. Nous détaillons dans un premier temps les mécanismes de base nécessaires à la mise en œuvre d'une vérification dynamique formelle. Puis, nous présentons SimGrid, et plus particulièrement SimGridMC, l'outil de vérification dans lequel les contributions réalisées durant cette thèse ont été implémentées.

1 Mécanismes de base nécessaires à la vérification

La vérification dynamique formelle d'applications distribuées nécessite la mise en œuvre de plusieurs mécanismes de base, issus des techniques traditionnelles de *model checking* mais adaptés à l'étude d'implémentations réelles.

1.1 Médiation des communications

Afin de vérifier exhaustivement l'ensemble des exécutions possibles, il est nécessaire d'évaluer l'ensemble des ordonnancements possibles des communications des processus. Pour cela, une médiation doit être mise en œuvre au niveau des communications afin de créer et explorer toutes les correspondances possibles entre les `Send` et `Recv` et donc garantir l'exhaustivité de la vérification.

L'origine des différents ordonnancements possibles est multiple. Les points de rendez-vous mis en œuvre dans les applications CSP permettent tout d'abord de se mettre en attente d'un message avant même son envoi. Il est donc possible qu'un processus fasse un `Recv` avant que tout autre processus n'ait fait un `Send` pour lui dans ce même point de rendez-vous. C'est ainsi que pour chaque paire de communications, deux ordres d'exécution sont possibles : la réception du message est déclenchée avant son envoi ou inversement.

De même, selon si ces communications sont bloquantes ou non, un processus peut exécuter plusieurs communications successivement ou non, créant ainsi de nouvelles exécutions possibles.

Enfin, certains types de communications permettent de ne pas spécifier la source ou la destination (de type `Any`) du message correspondant. Si deux processus envoient alors un message à un même processus, il est nécessaire de vérifier toutes les associations possibles.

1.2 Contrôle de l'exécution

Les cas d'indéterminisme des communications identifiés, la mise en œuvre de la médiation des communication nécessite alors de pouvoir contrôler l'application étudiée durant son exécution.

En particulier, il faut être capable d'identifier et analyser chaque communication réalisée au cours d'une exécution afin d'identifier les autres chemins d'exécutions possibles et ainsi garantir une vérification exhaustive.

Pour cela, un mécanisme d'interception des communications doit être mis en œuvre avant ou juste après l'exécution de chacune de celles-ci afin d'évaluer les éventuels autres appariements possibles entre `Send` et `Recv` et de forcer alors l'exploration sur les nouveaux chemins d'exécutions qui en résultent. Il en est de même pour les communications de type `Any`.

1.3 Exploration de l'espace d'états

Le dernier mécanisme indispensable à la mise en œuvre de la vérification dynamique formelle d'applications distribuées réelles est l'exploration de l'espace d'états construit grâce aux mécanismes précédents. Sous forme d'un graphe, cet espace d'états peut être exploré en profondeur ou en largeur. Selon l'approche choisie, plusieurs difficultés peuvent se poser.

Lors d'un parcours en profondeur, la principale difficulté réside dans la mise en œuvre d'un mécanisme de *backtracking* permettant de revenir en arrière sur des états précédemment rencontrés mais depuis lesquels d'autres chemins d'exécution non explorés sont possibles. Sachant que l'application étudiée est réellement exécutée, chaque état correspond à l'état courant du système en mémoire. Il faut donc dans un premier temps réussir à capturer correctement cet état et ensuite le restaurer à un instant donné. Nous avons vu dans le chapitre I de cette partie que deux approches sont proposées dans la littérature pour permettre ce retour en arrière. Le *backtracking* est soit réalisé depuis l'état initial du système (seul état sauvegardé et restauré au cours de l'exploration) avec le rejeu des transitions menant à un état donné. Ou bien, le retour s'effectue immédiatement sur l'état donné grâce à la sauvegarde de tous les états rencontrés au cours de l'exploration.

Lors d'un parcours en largeur, ce mécanisme de *backtracking* n'est pas nécessaire. Toutefois, la difficulté est alors de gérer l'état de l'application sur les multiples chemins d'exécution parcourus en parallèle.

2 Mise en œuvre dans SimGrid

SimGrid [CGL+14] est un outil de simulation offrant les fonctionnalités nécessaires à l'étude d'applications distribuées hétérogènes en environnements distribués. Il facilite la recherche dans le domaine de la programmation d'applications distribuées et parallèles sur des plates-formes de calcul distribué à partir d'un système allant du poste de travail aux grilles de calculs.

2.1 Étude conjointe de la performance et de la correction

Dans son développement initial, SimGrid permet de simuler des applications distribuées avec plusieurs milliers (voire des millions) de nœuds, face à des conditions très diverses et avec une modélisation précise des ressources. Il est ainsi possible de réaliser à la fois des études de performance mais également d'étudier les propriétés d'un système et de prédire son évolution.

Les motivations pour ajouter un mécanisme de vérification des applications développées directement au sein de cet outil sont doubles. Évaluer la correction d'un algorithme est tout autant important que l'étude de ses performances dès lors qu'un algorithme rapide mais incorrect n'est guère plus intéressant qu'un algorithme plus lent tout aussi incorrect. De plus, certaines fonctionnalités sont communes entre les simulateurs pour l'étude de performances et les outils de vérification, il est donc parfaitement envisageable de combiner cet ensemble dans un seul outil.

SimGrid permet de simuler des applications distribuées dans un environnement contrôlé. Il contrôle notamment l'état du système, la mémoire mais également les communications réalisées par les processus simulés. Ces derniers peuvent donc être exécutés de manière sélective mais surtout être interrompus à chaque communication. Dès lors, nous pouvons nous appuyer sur ces fonctionnalités de médiation pour mettre en œuvre un mécanisme de vérification et réaliser une simulation exhaustive à travers l'étude du comportement de l'application sur n'importe quelle plate-forme. En intégrant cette vérification directement au sein de SimGrid, il est également possible de travailler avec les applications développées au sein de celui-ci sans avoir à les réécrire dans un formalisme particulier, ce qui rend l'approche immédiatement plus pratique par rapport aux outils de vérification imposant un formalisme spécifique.

2.2 SimGridMC

Depuis 2007, un nouveau module est proposé au sein de SimGrid : SimGridMC. Celui-ci permet aux utilisateurs de vérifier l'implémentation des applications développées au sein du simulateur sans les modifier. Grâce à son intégration directement au sein de l'outil, il est possible de vérifier une application à tout moment au cours de son développement. La figure 3.1 présente comment SimGridMC est intégré au sein de l'architecture globale de SimGrid. Il met en œuvre un algorithme d'exploration de l'espace d'états en générant exhaustivement l'ensemble des exécutions possibles depuis un état initial du système étudié, notamment en évaluant le non-déterminisme des communications de l'application.

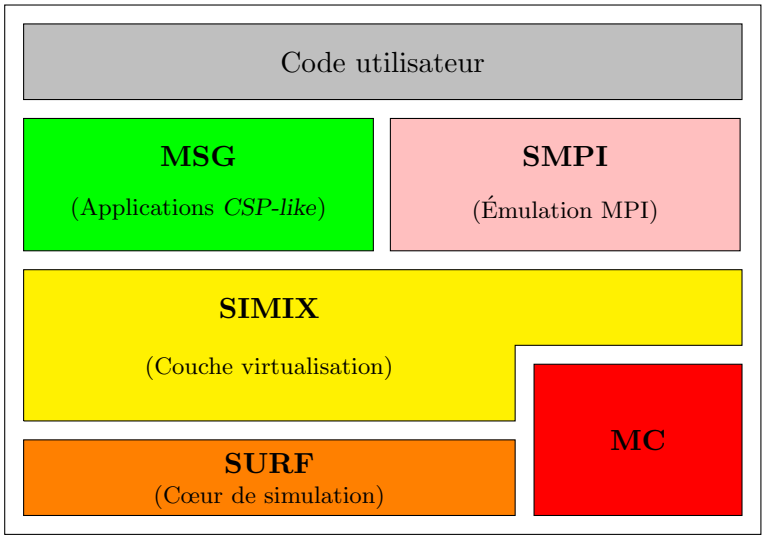

Figure 3.1 – Intégration de SimGridMC dans l'architecture en couches de SimGrid.

2.2.1 Construction à la volée de l'espace d'états

SimGridMC se base sur le modèle d'atomicité, présenté dans la section 3.1.2 du chapitre 1 de cette partie, pour représenter l'espace d'états correspondant au système étudié. Cet espace d'états est construit à la volée lors de la vérification du système à travers son exécution.

Pour générer l'ensemble des exécutions possibles, SimGridMC intercepte les communications en bloquant les processus avant qu'ils ne modifient l'état du réseau. Cela permet de reporter les décisions d'association d'une communication avec une source et une destination. Ceci est réalisé à chaque fois que tous les processus ont annoncé leur communication et permet ainsi de considérer toutes les possibilités.

Cette interception est réalisée par la couche SimIX de SimGrid. Elle implémente une interface basée sur quatre opérations : `Send`, `Recv`, `Wait` et `Test`. Les deux premières opérations sont responsables de la gestion des sources et destinations des communications tandis que les deux suivantes sont responsables du transfert des données associées aux communications. Bien que restreinte, celle-ci reste toutefois suffisante pour l'implémentation des interfaces de programmation au niveau utilisateur dans les modules supérieurs. La séparation des fonctionnalités associées à chaque opération permet un contrôle à grain fin de l'état du réseau et ainsi d'obtenir des interfaces de programmation *CSP-like* ou MPI adaptées au développement et à la vérification d'applications distribuées.

Ainsi, chaque nœud de l'espace d'états, représenté sous la forme d'un graphe, correspond à l'état global courant du système. Celui-ci contient l'état local de chaque processus ainsi que l'état du réseau. Dans SimGrid, l'état de chaque processus est composé des registres CPU, de la pile et de son tas. L'état du réseau est le seul élément partagé entre les processus et est caractérisé par les messages en transit. Il comprend l'ensemble des communications déposées dans les points de rendez-vous ainsi que les zones mémoires source et destination des processus associés à ces communications. Les transitions qui relient ces nœuds correspondent quant à elles à l'exécution d'une communication de type `Send`, `Recv`, `Wait` ou `Test` ainsi que toutes les éventuelles instructions de calcul jusqu'à la prochaine communication.

2.2.2 Exploration en profondeur

SimGridMC réalise une vérification explicite des états (par opposition à symbolique), c'est-à-dire que les états et les transitions y sont représentés individuellement. En effet, il semble extrêmement difficile, voire impossible, de les représenter de façon symbolique notamment à cause des informations mémoires contenues dans l'état global du système à chaque instant de l'exécution mais également à cause de la relation de transition entre les états, déterminée par l'exécution de code C. SimGridMC explore donc l'espace d'états en parcourant en profondeur l'ensemble des ordres d'exécution possibles des processus en sauvegardant sous la forme d'une pile les états à explorer. Chaque entrée dans cette pile contient l'ensemble des processus à explorer, ceux déjà explorés et la transition sortante à exécuter.

La vérification commence par une phase d'initialisation, représentée dans la figure 3.2, durant laquelle tous les processus à exécuter sont ordonnancés jusqu'à l'instruction (exclue) correspondant à leur première communication. Nous obtenons alors un état global initial S_0 contenant les transitions à exécuter et qui est donc ajouté dans la pile d'exploration.

L'exploration, présentée dans la figure 3.3, commence ensuite par le traitement de l'état présent en haut de la pile d'exploration. Pour cela, une des transitions à exécuter au sein de celui-ci est choisie. Les changements induits par son exécution sont répercutés sur l'état du réseau, celle-ci est marquée comme exécutée au sein de l'état, et les nouveaux processus à exécuter sont ordonnancés pour l'état suivant. De plus, l'ensemble des instructions présentes entre la communication exécutée et la prochaine sont également exécutées. L'exécution du programme

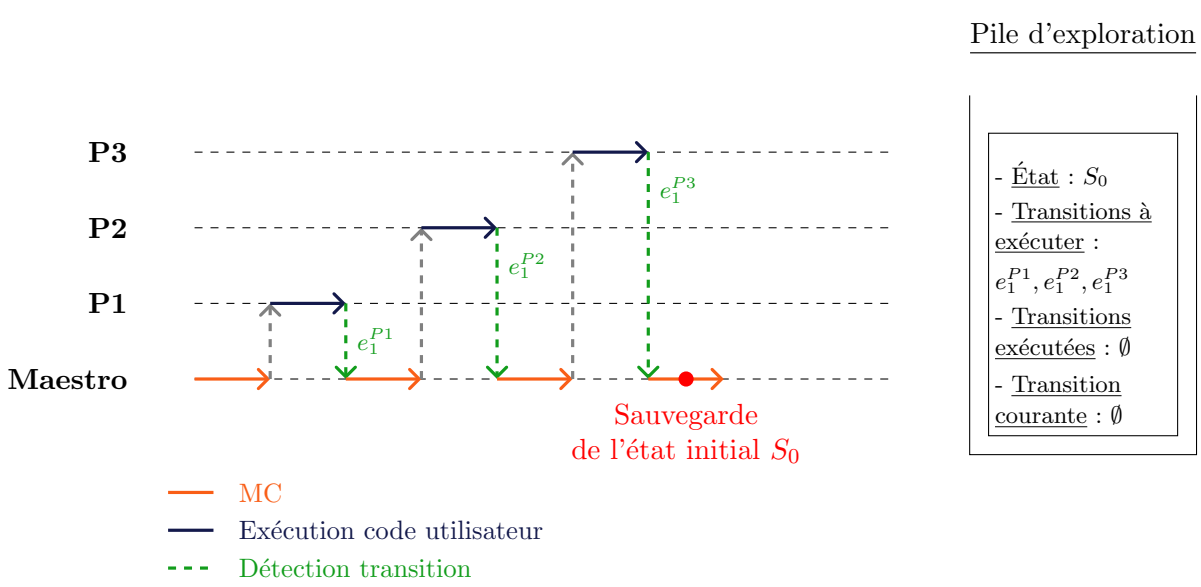

Figure 3.2 – Phase d'initialisation de SimGridMC.

est donc ininterrompue jusqu'à la prochaine communication d'un processus. Un nouvel état est alors obtenu et ajouté à la pile d'exploration.

Le même traitement que pour l'état initial est réalisé sur celui-ci : une des transitions à exécuter dans l'état courant est choisie, exécutée et marquée comme telle. L'état du réseau est mis à jour après l'exécution de celle-ci ainsi que l'état local de tous les processus, qui sont à nouveau ordonnancés pour l'état suivant. Cette technique est réalisée pour chaque état de la pile d'exploration. Un état est définitivement retiré de la pile lorsque l'ensemble des transitions à exécuter au sein de celui-ci ont été exécutées. L'exploration se termine lorsque l'espace d'états a été entièrement exploré (la pile d'exploration est vide) ou lorsqu'une violation de propriété a été détectée ou, éventuellement, lorsqu'il n'y a plus assez de ressources pour poursuivre la vérification.

2.2.3 Exploration exhaustive sans sauvegarde des états

SimGridMC propose dans son premier développement une exploration exhaustive sans sauvegarde des états (*Stateless search*). Cette approche, introduite par Godefroid en 1997 [God97], permet de réaliser une exploration exhaustive avec gestion de non-déterminisme pour la vérification des programmes concurrents notamment en contrôlant automatiquement l'exécution des opérations de synchronisation.

La principale difficulté dans l'application de cette approche à la vérification des systèmes distribués, réside dans l'implémentation du mécanisme de retour en arrière (*backtracking*) nécessaire pour explorer d'autres chemins d'exécution et donc avoir une exploration exhaustive. En effet, lorsque le parcours en profondeur est terminé sur une branche d'exécution, s'il existe des états sur ce chemin ayant plusieurs transitions à exécuter, il est nécessaire d'y revenir pour parcourir les transitions non exécutées. Sachant que les états explorés ne sont pas sauvegardés, il n'est pas possible de revenir directement sur ces états avec des transitions non

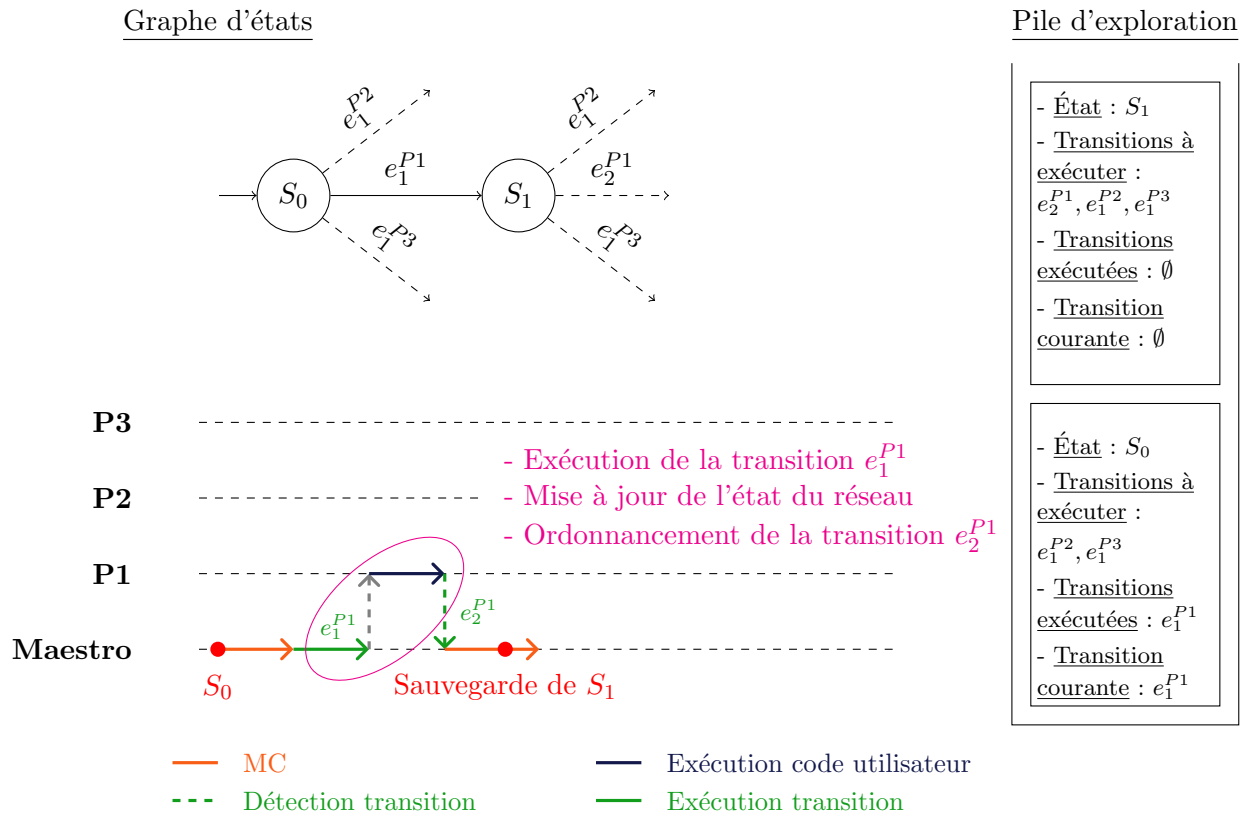

Figure 3.3 – Phase d'exploration d'une transition dans SimGridMC.

exécutées.

SimGridMC réalise donc ce retour en arrière en restaurant l'état initial S_0 du système (celui sauvegardé avant que l'exploration ne commence) et en rejouant l'ensemble des transitions menant à l'état ciblé. Il s'agit des transitions courantes dans les entrées de la pile d'exploration. Ce mécanisme de *backtracking* sur un état précédent est présenté dans la figure 3.4.

Au début de la vérification, avant de commencer l'exploration, l'ensemble des segments mémoire contenant les informations dynamiques qui caractérisent l'état global du système sont sauvegardés (**Étape 0**). L'exploration débute ensuite. Chaque transition exécutée lors du parcours en profondeur est sauvegardée dans la pile d'exploration. L'exploration d'une branche se termine lorsqu'il ne reste aucune nouvelle transition à exécuter. Un dépilement des éléments présents dans la pile d'exploration est alors réalisé, tel que tout état ne possédant plus de transitions à exécuter est définitivement supprimé de celle-ci (**Étape 1**). Dès que l'on rencontre dans cette pile un état pour lequel il existe une autre transition à exécuter, la phase de dépilement est stoppée. Commence alors la restauration du système pour revenir sur cet état. Pour cela, les segments mémoires sauvegardés dans l'état initial sont restaurés (on écrase les segments courants) (**Étape 2**) puis on traverse la pile d'exploration en partant du bas de la pile et en rejouant la transition exécutée dans chaque entrée. Ceci est réalisé jusqu'à la dernière entrée de la pile qui contient l'état avec une autre transition à exécuter (**Étape 3**).

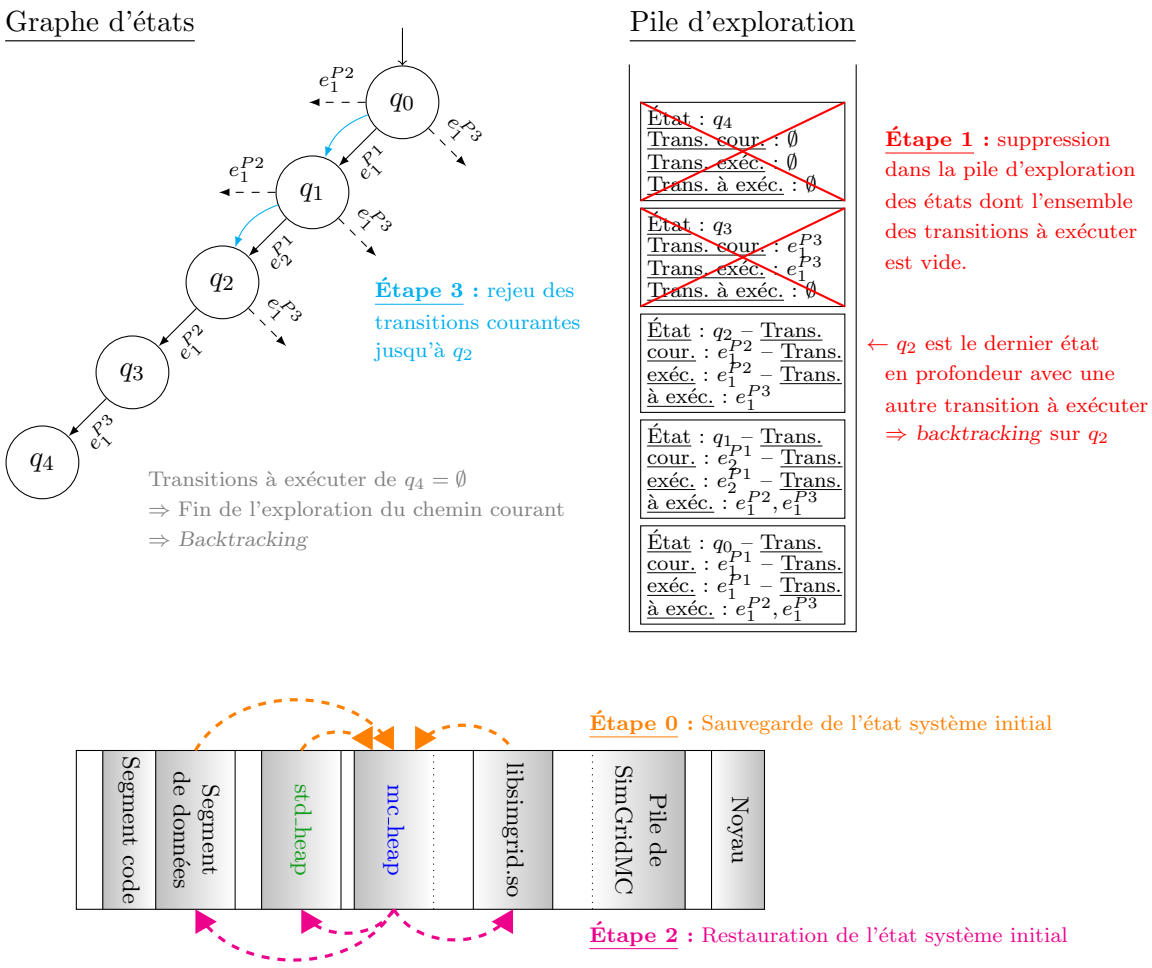

Figure 3.4 – Mécanisme de backtracking avec SimGridMC : exemple de backtracking sur q_2 depuis q_4 par restauration de l'état initial q_0 et rejeu des transitions e_1^{p1} et e_2^{p1}

2.2.4 Propriétés de correction vérifiées

SimGridMC permet initialement de vérifier uniquement des propriétés de sûreté exprimées sous la forme d'assertions locales et intégrées au sein même du code du système étudié. Ces assertions permettent d'exprimer des propriétés sur des variables mais pour un seul processus. Cette caractéristique est cruciale dans la mise en œuvre d'une technique de réduction implémentée au sein de SimGridMC et présentée dans la section 2.2.5. Cependant, il est possible de dépasser certaines limitations, en particulier pour les variables locales en utilisant simplement des références globales disponibles à tout instant au cours de l'exécution. Une primitive `MC_assert` permet donc actuellement de définir des points d'exécution pour lesquels le résultat de l'assertion doit être évalué. Enfin, un *deadlock* se caractérisant par une attente de tous les processus dans le simulateur, une vérification automatique de ce type d'erreur est également réalisée pendant l'exploration en plus des éventuelles assertions.

2.2.5 Réduction de l'espace d'états

Une seule technique de réduction [CDSM10] est proposée avec la vérification de propriétés de sûreté au sein de SimGridMC. Celle-ci est inspirée de la réduction dynamique par ordre partiel (DPOR) proposée par Godefroid *et al.* dans [God05] et présentée dans la section 4.2 du chapitre 2 de cette partie.

Elle permet d'évaluer dynamiquement la dépendance entre transitions lors de leur exécution. Cette opération est réalisée à la fin de chaque exécution une fois que toutes les transitions ont été exécutées. À cet instant, l'ensemble des informations liées aux transitions ainsi que leur influence sur l'état du réseau sont connus et peuvent donc être utilisés pour évaluer les transitions dépendantes.

À chaque étape de l'exploration est déterminé l'ensemble des processus activés (ayant une transition à exécuter). Lors d'une exploration sans réduction, à chaque création d'un nouvel état, toutes les transitions à exécuter sont ajoutées à l'ensemble des transitions à exécuter. Chaque transition de cet ensemble est traitée grâce au mécanisme de *backtracking* présenté dans la section précédente. Avec DPOR, une seule transition à la fois est ajoutée dans cet ensemble lors de la création d'un nouvel état même si plusieurs sont actives. Les autres transitions actives sont ajoutées lorsque le chemin courant été entièrement exploré, selon les dépendances détectées. Le principe de cette réduction est donc de n'explorer que les transitions dépendantes dans chaque état, c'est-à-dire celles dont l'ordre d'exécution influe sur les états suivants de l'exécution.

2.3 Comparaison avec les autres outils de vérification d'applications MPI

SimGridMC permet plus particulièrement de vérifier l'implémentation réelle des applications distribuées développées directement au sein du simulateur. Ceci suppose donc une dépendance avec les interfaces de programmation proposées par ce dernier, limitant son champ d'application. Cependant, grâce à SMPI, l'interface utilisateur de SimGrid qui réimplémente le standard MPI au dessus du noyau de SimGrid, il est possible de réaliser une vérification dynamique formelle d'applications MPI quelconques. Bien que le standard MPI soit majoritairement utilisé dans la communauté HPC en particulier, peu d'outils offrent aujourd'hui cette fonctionnalité malgré un besoin établi [GKS+11].

Pour réaliser cette vérification, SimGridMC s'appuie donc sur SMPI qui réimplémente une partie du standard MPI (versions 2 et 3) ainsi que les algorithmes de MPICH, OpenMPI et StarMPI. Pour cela, il capture la sémantique des communications point-à-point à travers un modèle défini dans SimGrid [BDG+13]. Ces communications sont ensuite interceptées au niveau de la couche SimIX dans lesquelles chaque primitive MPI est décomposée en utilisant uniquement les quatre opérations : `Send`, `Recv`, `Wait` et `Test`. La vérification réalisée par SimGridMC s'effectue alors à chacune de ces opérations, ce qui garantit une exploration fine et donc une détection précise d'éventuelles erreurs, même au sein des appels MPI.

Grâce à la réimplémentation d'une partie du standard MPI associée aux mécanismes de simulation proposés dans SimGrid, l'approche mise en œuvre par SimGridMC se distingue singulièrement des autres outils de vérification MPI.

ISP [VSGK08] (DAMPI [VAG+10] dans sa version distribuée) propose par exemple une approche basée sur l'interface de profilage PMPI et la réécriture des appels MPI. PMPI permet dans un premier temps d'intercepter chaque appel MPI réalisé par l'application étudiée. Ensuite, un protocole distribué est utilisé entre les processus pour déterminer quels messages doivent être différés et lesquels peuvent être immédiatement transmis. De plus, une modification des pa-

ramètres des appels MPI est réalisée lorsque plusieurs comportements peuvent y être associés pour ainsi générer de nouveaux chemins d'exécution à vérifier et donc prendre en compte la présence éventuelle de non-déterminisme dans le comportement de l'application. Cependant, cette interface de profilage ne permet de voir certains appels MPI que sous une forme atomique. Les communications point-à-point qui composent ces appels deviennent alors totalement invisibles du point de vue de l'outil de vérification. Ceci est notamment valable dans le contexte des communications collectives dans lesquelles tous les processus d'un communicateur donné sont impliqués. Sans la décomposition des communications émises lors de ces appels, certains cas d'erreurs peuvent donc ne pas être détectés.

Contrairement à SimGridMC et ISP, TASS [SZ11] vérifie les applications MPI à travers une exécution symbolique. Pour cela, il extrait automatiquement une abstraction de l'application étudiée et réalise ensuite la vérification sur cette dernière. La génération de cette abstraction nécessite alors de définir à la fois les primitives du langage C et celles du standard MPI. C'est pourquoi cet outil est aujourd'hui limité à un sous-ensemble restreint des programmes C mais également des applications MPI pouvant être vérifiées.

Enfin, très récemment, le projet AISLINN [7] propose de vérifier des applications MPI écrites en C/C++ sans modification, ni recompilation. Pour cela, il s'appuie sur l'outil de débogue Valgrind et met en œuvre une architecture client/serveur autour de celui-ci. Le client situé au dessus de Valgrind est chargé de déterminer les appels à intercepter et réalise également le mécanisme de sauvegarde et restauration de l'état du système pour le *backtracking*. De l'autre côté, le serveur contient la sémantique MPI et réalise la vérification de la correction haut-niveau mais également une réduction par ordre partiel de l'espace d'états à explorer. Cet outil est le successeur de Kaira [BBMS13], un autre outil de vérification MPI mais qui nécessite une formalisation du programme à vérifier dans le langage Kaira, tout comme MPI-SPIN [Sie07] qui prend en entrée une description en Promela de l'application MPI étudiée.

7. `http://verif.cs.vsb.cz/aislinn`

Deuxième partie

Analyse sémantique dynamique d'un état système par introspection mémoire

L'enjeu majeur de la vérification dynamique formelle d'applications réelles est de réussir à accéder aux informations qui constituent un état système pour ensuite les analyser. Dans ce contexte, *dynamique* signifie que les informations sont supposées accessibles uniquement lors de l'exécution de l'application étudiée, sans analyse statique préalable.

Plusieurs approches permettent à la fois d'accéder à ces données mais également de les analyser. Dans le cas d'applications Java, l'état courant du système est contenu dans la machine virtuelle (JVM). Dès lors, grâce aux métadonnées fournies par celle-ci, il est possible de reconstruire la sémantique de chaque octet de la mémoire. Cependant, cela nécessite de modifier la JVM comme le propose JavaPathFinder[8]. Il est ainsi possible d'aller au delà d'une introspection Java classique, qui va simplement évaluer les valeurs contenues dans un objet mais pas leur sémantique. Ceci reste toutefois difficilement applicable à la vérification d'applications C ou Fortran puisqu'il n'existe, par défaut, ni machine virtuelle ni système de ramasse-miettes permettant la gestion automatique de la mémoire.

De nombreux travaux basés sur l'analyse statique ont été réalisés dans le cadre de programmes C. Les erreurs trouvées par ces outils vont de la détection de code inaccessible aux erreurs de l'environnement d'exécution pour certains. Il est également possible de détecter des dépassements mémoire ou des fuites, ainsi que la non-terminaison de programmes. Cependant, cette approche souffre de deux problèmes. Tout d'abord, les erreurs rapportées peuvent correspondre à des faux-positifs, c'est-à-dire que celles-ci n'existent pas réellement lors de l'exécution du système. Selon les outils et la précision de l'analyse réalisée, ce taux de « fausses erreurs » peut être assez important pour rendre l'approche inefficace lors de la vérification d'un système. De plus, il subsiste de nombreuses erreurs détectables uniquement à travers l'exécution réelle du système, certaines informations ne pouvant être accessibles qu'à cet instant. Ceci est notamment le cas face à des programmes avec des allocations mémoires dynamiques. Une approche plus complète serait donc de combiner analyse statique avec analyse dynamique. Dans le cadre de cette thèse, nous limitons nos recherches à l'approche dynamique, moins étudiée dans la littérature.

Nous proposons dans cette partie une approche bas niveau permettant de réaliser une analyse sémantique de l'état d'un système par introspection mémoire sur du code C, C++ ou Fortran. Pour cela, nous exposons dans un premier chapitre les motivations de cette analyse, notamment ses enjeux dans le cadre de la vérification dynamique formelle, ainsi qu'un état de l'art des techniques existantes. Puis, dans un second chapitre, nous identifions cinq obstacles à cette analyse et détaillons pour chacun d'entre eux une solution. Enfin, une évaluation des travaux est réalisée dans un troisième chapitre à travers plusieurs expérimentations.

8. http ://javapathfinder.sourceforge.net/The_Model_Java_Interface.html

Chapitre 1

Motivations

1 Enjeux

1.1 Analyse syntaxique *vs* Analyse sémantique

L'analyse syntaxique d'un programme s'intéresse uniquement à sa forme et non à sa signification. Elle va permettre, par exemple, de détecter qu'une parenthèse ouverte est bien fermée plus tard. L'analyse sémantique d'un programme permet quant à elle de vérifier le sens d'un programme. Ces deux phases d'analyse sont par exemple réalisées par un compilateur. Une analyse syntaxique est faite dans un premier temps. Celle-ci permet de vérifier que la grammaire correspondant au langage de programmation est respectée. Puis, une analyse sémantique permet de vérifier la signification du programme en détectant notamment les variables non initialisées mais utilisées, les erreurs de types, *etc.*

Vérifier un programme réel durant son exécution consiste à analyser son état à un instant donné de l'exécution. Cet état est caractérisé par le contenu de la mémoire. Dans ce cas, faire une simple analyse syntaxique consisterait uniquement à regarder la valeur de chaque octet qui la compose sans savoir à quoi il correspond. Ceci ne permet donc pas d'établir des propriétés d'exécution sur le programme étudié. L'objectif de l'analyse sémantique dans le contexte de la vérification dynamique formelle de programmes est donc dans un premier temps de caractériser l'état du système durant son exécution pour ainsi le vérifier ensuite par rapport à certaines propriétés d'exécution définies.

1.2 Analyse sémantique pour la vérification dynamique formelle

La caractérisation de l'état du système au cours de son exécution grâce à une analyse sémantique permet de répondre à plusieurs problématiques de la vérification dynamique formelle d'applications distribuées réelles.

1.2.1 Réduction de l'espace d'états par la détection des états déjà visités

La première problématique forte est l'adaptation des techniques de réduction existantes à la vérification d'applications réelles. En effet, la majorité des approches présentées dans la section 4.2 du chapitre 2 de la partie I repose sur le fait que la vérification est faite sur des modèles ou sur des applications réelles mais dont la faible complexité permet finalement de travailler sur une abstraction minimale de celles-ci. Parmi ces techniques de réduction, la plus remarquable est la

détection des états déjà visités. Grâce à cette détection, il est possible de stopper toute exploration d'une branche des exécutions possibles si celle-ci contient un état précédemment vérifié. Ainsi, l'espace d'états ne contient que l'ensemble des états du système réellement différents, ce qui peut réduire considérablement sa taille pour certaines applications. Dans le cadre de la vérification de modèles, cette détection est facilement réalisée d'après les caractéristiques du modèle étudié. Sur des applications réelles, la solution adoptée par de nombreux outils est d'utiliser des *hash* à la place de l'état complet du système. Cela permet de réaliser une exploration avec sauvegarde des états sans subir une explosion de la consommation mémoire mais également de détecter facilement et rapidement les états identiques par simple comparaison de *hash* et ainsi réduire l'espace d'états.

Cependant, cette approche soulève deux faiblesses. La génération de *hash*, d'après l'état complet du système, implique immédiatement un risque de collision, c'est-à-dire que des états initialement différents peuvent avoir le même *hash* d'après la fonction de hachage implémentée. Dès lors, la correction de la réduction ne peut être assurée puisque l'exploration sera potentiellement réduite alors qu'il s'agit d'un état qui peut n'avoir jamais été visité. La seconde faiblesse est liée à la difficulté voire à l'impossibilité de calculer ce *hash* pour l'ensemble de l'état du système dans certains cas. En effet, dans le cas d'applications avec des allocations mémoire dynamiques telles que le permettent les langages C, C++ ou Fortran, il n'est pas correct de réaliser un simple *hash* de la mémoire telle quelle notamment à cause des pointeurs qui peuvent être différents alors que le contenu des zones mémoires pointées est identique. Il est donc nécessaire de travailler sur une forme canonique de la mémoire et de prendre en compte ce type d'informations. Cependant, cette mise sous forme canonique ne peut être entièrement réalisée dans certains cas, certaines zones mémoires n'étant pas toujours identifiées au cours de l'exécution. Il est alors nécessaire de travailler avec l'état complet du système et non plus avec une forme hachée de celui-ci.

1.2.2 *Backtracking* directement sur un état donné

Au-delà d'une réduction de l'espace d'états grâce à la détection des états visités, la gestion (sauvegarde et analyse) de l'état complet du système durant l'exploration permet de mettre en œuvre un mécanisme de *backtracking* directement sur un état donné. En effet, sans sauvegarde des états, lorsqu'il est nécessaire de revenir sur un état dans lequel il existe des transitions actives non exécutées, le *backtracking* est réalisé grâce à la restauration de l'état initial du système et le rejeu des transitions menant à l'état ciblé. Toutefois, cette approche s'avère inadaptée dans le cadre de la vérification des systèmes HPC réalisant de longs calculs. Un rejeu depuis l'état initial signifie alors qu'en plus des communications à rejouer, il faut également refaire tous les calculs qui précèdent l'état sur lequel on souhaite revenir. Grâce à la sauvegarde complète de chaque état du système au cours de l'exploration, il est donc possible de réduire ce temps de *backtracking* en restaurant directement l'application dans un état donné. Plusieurs outils de vérification mettent en œuvre une exploration avec sauvegarde des états, mais ils ne conservent soit qu'un *hash* des états, soit une partie d'entre eux. Il est alors impossible de réaliser une restauration complète de ces états.

1.2.3 Détection de cycles

La détection de cycles d'exécutions infinies joue un rôle central dans la vérification de programmes. Leur détection permet notamment de vérifier dans un premier temps des propriétés de vivacité quelconques exprimées en LTL. Comme expliqué en section 2 du chapitre 1 de la partie I, une propriété de vivacité est satisfaite en un temps fini mais violée sur des exécutions infinies.

Cette violation se caractérise alors par un contre-exemple correspondant à une exécution infinie. Au sein de ce contre-exemple, il existe un cycle d'exécution dit « acceptant » dans lequel la propriété n'est pas satisfaite qui permet ainsi d'affirmer que la propriété ne sera jamais satisfaite sur le chemin d'exécution étudié. Il est donc nécessaire de réussir à identifier lorsque le système entre dans un cycle acceptant. Pour cela, la détection des états sémantiquement identiques est indispensable.

De plus, le problème de la terminaison de programme fait partie des propriétés de vivacité incontournables dont la vérification est basée (en partie) sur la détection de cycles de non-progression. Cette notion de « cycle de non-progression » est à différencier de la définition établie par SPIN. En effet, ce dernier définit des actions dites « de progrès » et déclare tout cycle sans action de progrès comme un cycle de non-progression. Dans notre contexte, la notion de progrès fait ici référence à l'exploration de nouveaux états sur un chemin courant. Dès lors que l'exploration retombe sur un état précédemment visité, aucun nouvel état n'est créé sur le chemin courant. L'exploration n'évolue et ne progresse donc plus sur celui-ci en terme de création de nouveaux états, nous parlons alors de cycle de non-progression sur le chemin étudié. Plus simplement, nous pouvons comparer ces cycles de non-progression à des boucles d'exécution infinies. Il est naturellement impossible de résoudre le problème de la terminaison de programmes en toute généralité, mais il reste toutefois possible de prouver si un programme se termine ou non [CPR11]. La vérification dynamique formelle permet d'étudier en partie cette terminaison sur des applications réelles. En effet, la non-terminaison d'un programme peut être évaluée de deux manières. Soit le programme change d'état perpétuellement, soit celui-ci boucle dans un état déjà rencontré au cours de l'exécution. Dans le premier cas, si l'état du système correspond à chaque fois à un nouvel état, la non-terminaison ne pourra être détectée par la détection de cycles. Cependant, cette dernière est tout à fait valide dans le cas où le programme tombe dans une boucle infinie dans laquelle son état correspond à un état déjà connu au cours de l'exécution.

1.2.4 Vérification dynamique formelle de protocoles cycliques

Les cycles de non-progression ne signifient pas toujours un mauvais comportement de l'application étudiée. En particulier, il existe de nombreux protocoles cycliques dans les systèmes pair-à-pair, dont le comportement évolue d'après des évènements externes périodiques. Parmi eux, il y a, par exemple, le protocole Chord [SMK+01]. Il s'agit d'un protocole de table de hachage distribuée au sein duquel des nœuds sont répartis d'après une topologie en anneau, tel que chaque nœud ne connaît que son prédécesseur et son successeur directs. Une fonction de hachage génère une clé pour chaque nouveau nœud. Chaque nœud est ensuite placé dans l'anneau de telle manière que les clés soient ordonnées par ordre croissant. Lorsque l'on souhaite accéder à une donnée associée à une clé, la recherche se fait depuis un nœud connu qui transmet la requête à un de ses voisins directs. Lui-même diffuse la requête à un de ses voisins directs, jusqu'à trouver le nœud responsable de la clé recherchée. Ce protocole fonctionne même en présence d'arrivées et de départs continus de nœuds en déployant des actions périodiques pour maintenir la cohérence de l'anneau. Si aucun nœud ne quitte ou n'arrive dans l'anneau entre deux mêmes évènements périodiques, le système entre alors dans un cycle et se stabilise. Cet état stable peut perdurer de façon infinie. Dès lors, son état ne change plus ce qui est normal dans ce contexte. Cependant, si le comportement du système étudié est infini, il est difficile d'assurer une vérification exhaustive sans détection de ces cycles.

La réduction dynamique par ordre partiel (DPOR), présentée dans la section 4.2 du chapitre 2 de la partie I, souffre particulièrement de ce cas de figure. En effet, celle-ci impose dans sa version de base deux conditions : l'espace d'états du système étudié doit être fini et acyclique.

Cette réduction ne peut donc s'appliquer à l'étude de protocoles cycliques finis ou infinis qui font pourtant partie intégrante des systèmes distribués. Une extension de la réduction par ordre partiel [CMM13] offre la possibilité de faire une exploration bornée, c'est-à-dire de limiter la profondeur d'exploration de chaque branche d'exécution tout en garantissant la correction de la réduction et donc l'exhaustivité de l'exploration. Cette approche n'a toutefois pas encore pu être appliquée aux systèmes distribués et ne concerne actuellement que les systèmes concurrents *multithreadés* dont la principale caractéristique est l'utilisation d'une mémoire partagée. Grâce à la détection d'états sémantiquement identiques, il est possible de supprimer de l'exploration ces cycles et ainsi réaliser une exploration exhaustive même sur des espaces d'états initialement cycliques, finis ou infinis.

2 État de l'art

Du point de vue de la littérature, ce sujet a fait l'objet de plusieurs travaux permettant la vérification de programmes avec sauvegarde des états (*stateful model checking*). L'objectif de cette sauvegarde étant de réduire l'espace d'états à explorer, grâce à la détection des états déjà visités. Cette réduction est souvent appliquée pour la vérification de modèles mais peu lorsqu'il s'agit de travailler avec des applications réelles. Une raison à cela est la complexité des états qui peut être accrue selon le niveau du langage de programmation du système étudié.

Dans [LV01], les auteurs sont les premiers à s'intéresser à cette problématique dans le contexte de la vérification des programmes Java. Au-delà des difficultés liées à la manipulation d'états issus de l'implémentation réelle du système, ils identifient la dynamicité des systèmes comme principal obstacle à la détection des états identiques dans ce contexte. En effet, tout langage mettant en œuvre une gestion dynamique de la mémoire induit des allocations et libérations mémoires continues ainsi que des ordres d'allocations des objets différents. Ceci mène alors à des représentations des états différentes du point de vue de la mémoire. Toute détection de symétries au sein de ces représentations devient donc difficilement réalisable sans une introspection détaillée. Celle-ci pouvant être coûteuse, ils proposent de travailler en amont de la comparaison en instrumentant les allocations réalisées lors de l'exécution, c'est-à-dire en ordonnant les objets alloués en mémoire. Ceci permet alors d'obtenir une forme canonique pour chaque état et de ne sauvegarder que cette représentation. Dès lors, la génération d'un *hash* ou la comparaison complète de la représentation canonique sont facilement applicables et permettent donc de détecter des états identiques.

Cette approche a ensuite été poursuivie par [Ios04]. Ne se limitant plus aux programmes Java mais en travaillant avec d'autres langages haut niveau orientés objets tel que le C++, ces travaux se concentrent sur l'étude des systèmes dynamiques concurrents. La recherche de symétries s'effectue donc dans le tas de la mémoire mais également dans les *threads*. Cela suppose que chaque *thread* possède une structure linéaire notamment en l'absence de références partagées entre eux. Cette étude des symétries repose essentiellement sur la présence d'un système de traçage des allocations mémoires réalisées, de type ramasse-miettes. Les allocations ne sont pas instrumentées, contrairement à l'approche précédente, mais sont tracées par le ramasse-miettes. Il est ainsi possible de retrouver à tout instant quel objet est alloué à quel endroit de la mémoire et d'extraire un arbre représentant l'ensemble des objets existants. Ceci permet à nouveau de calculer une forme canonique d'un état et d'en déduire des classes d'équivalences de symétries. En combinant les deux réductions par symétries, l'auteur propose une nouvelle réduction pouvant être combinée avec la réduction par ordre partiel.

Enfin, une autre approche consiste à décomposer les états pour ne sauvegarder que les parties

pertinentes dans leur caractérisation et surtout pour la vérification [SBM+13]. Une fonction permet dans un premier temps de déterminer quelles parties sont pertinentes dans la caractérisation d'un état, puis une sérialisation est réalisée sur celles-ci. Le résultat obtenu est ensuite mis sous forme canonique puis un *hash* de celle-ci est calculé. Dès lors, la détection d'états identiques est effectuée avec les *hashs* obtenus pour chaque état. Fortement basée sur l'approche mise en œuvre dans JavaPathFinder, cette technique permet toutefois de réaliser en plus la mise sous forme canonique grâce à l'ajout de contraintes locales dans la phase de sérialisation.

Bien que ces travaux aient montré leur efficacité, ils se limitent aux langages orientés objets tels que Java ou le C++, ou bien ils nécessitent la présence d'un ramasse-miettes. À notre connaissance, aucune approche dynamique n'a été proposée pour l'analyse sémantique de programmes C ou Fortran pour lesquels l'absence d'outils automatiques de gestion de la mémoire rend particulièrement complexe la détection d'états identiques. La seule approche réalisant une introspection mémoire adaptée à ces langages (bien qu'incomplète) est une approche statique : *shape analysis* [YLB+08]. Cette analyse statique de code permet de découvrir et de vérifier des propriétés sur des structures de données allouées dynamiquement. Elle permet notamment de trouver les fuites mémoires, les doubles-libérations de mémoires, les dépassements d'indices de tableaux ou le déréférencement de « pointeurs pendouillants » (*dangling pointers*), etc.

Dans le chapitre suivant, nous présentons une approche permettant l'analyse sémantique dynamique par introspection mémoire d'un état système sur des programmes C, C++ ou Fortran.

Chapitre 2

Introspection mémoire d'un état système

Dans ce chapitre, nous présentons une approche d'introspection mémoire d'un état système permettant une analyse sémantique de celui-ci. Pour cela, nous détaillons dans un premier temps la composition d'un état système. Puis, nous identifions cinq obstacles à l'analyse d'un état dans le but de détecter une égalité sémantique entre deux états. Nous proposons pour chacun d'entre eux plusieurs solutions combinées permettant de détecter des états sémantiquement identiques. À noter qu'une partie de ces travaux a été réalisée en collaboration avec Gabriel Corona, Ingénieur de recherche confirmé au sein du projet SimGrid.

1 Composition d'un état système

Un état système est composé de l'ensemble des informations dynamiques de la mémoire du processus associé à son exécution, c'est-à-dire celles qui changent au cours de l'exécution de ce dernier. Ces informations correspondent aux variables globales de l'application, aux allocations mémoire dynamiques situées dans le tas et aux variables locales de chaque processus de l'application situées dans chaque pile d'appels de ces derniers. De plus, dans le contexte de la vérification au sein de SimGrid, l'état du système est également caractérisé par l'état global du réseau qui est défini par les variables globales du simulateur. L'ensemble de ces informations est géré par le système d'exploitation dans des segments mémoire séparés. Nous représentons dans la figure 2.1 ces segments et identifions ceux qui définissent l'état d'un système.

Les variables globales de l'application sont contenues dans les segment BSS et le segment de données. Une distinction entre les variables initialisées et les variables non-initialisées est effectuée par le système ce qui engendre deux segments séparés. Les informations sur l'état du réseau sont présentes dans le segment mémoire BSS et le segment de données associés à la bibliothèque dynamique de SimGrid (`libsimgrid.so`). Toutes les allocations mémoires dynamiques sont quant à elles associées au segment Tas. Enfin, dans une exécution classique, les variables locales sont présentes dans le segment Pile. Dans notre contexte, l'application à vérifier étant exécutée au sein de SimGrid, nous sommes soumis à l'architecture du simulateur. C'est ainsi que les piles des processus sont allouées directement au sein du tas.

Pour accéder à ces segments et déterminer leur adresse en mémoire, nous utilisons les informations présentes sous Linux dans le répertoire `/proc/#PID` associé à chaque processus. La liste des segments mémoire se trouve dans le fichier `maps` de ce répertoire. Une fois les segments

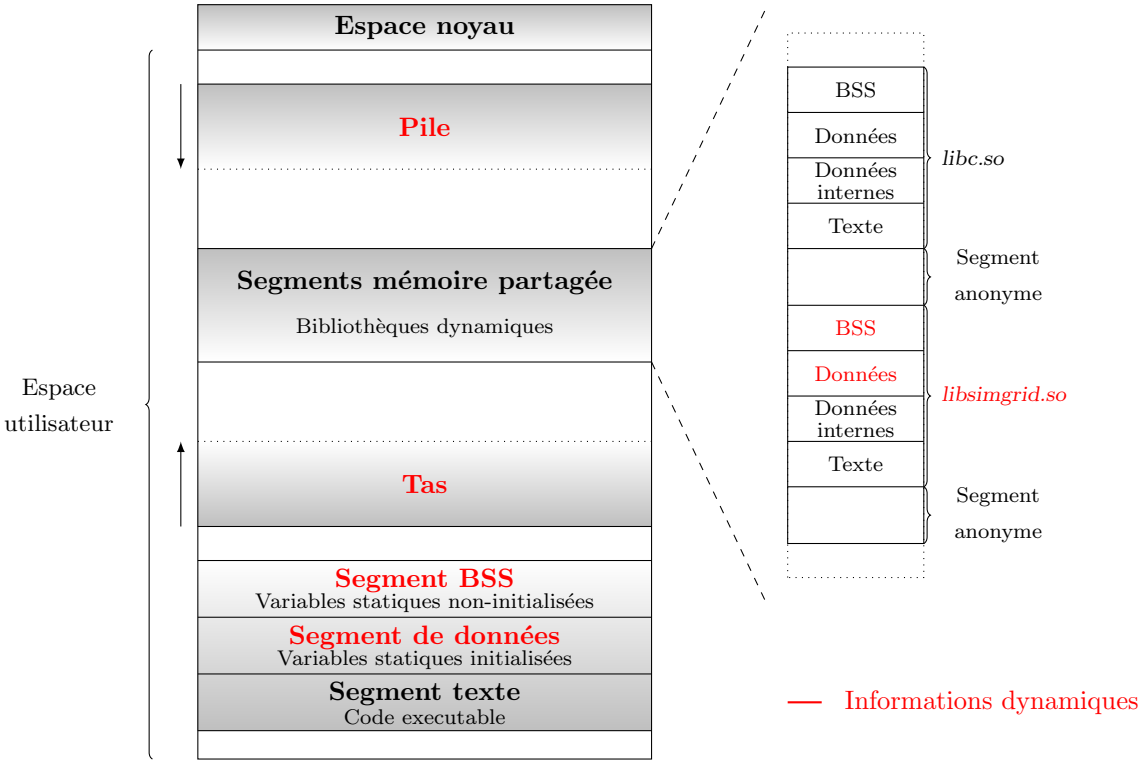

Figure 2.1 – Représentation mémoire d'un processus.

de l'état du système identifiés, une sauvegarde complète de chacun d'entre eux est réalisée. L'ensemble des données sauvegardées constitue alors un état système.

2 Détection d'états système sémantiquement identiques

L'objectif de l'analyse sémantique d'un état système est de caractériser l'ensemble des informations qui le constitue. L'interprétation de ces informations doit permettre ensuite de comparer un état système avec d'autres précédemment vérifiés au cours de l'exploration et ainsi détecter les états sémantiquement identiques.

Dans ce contexte, une première approche classique est de réaliser une comparaison linéaire, octet par octet, entre deux états donnés, en supposant au préalable que ces états sont de taille identique. Cependant, cette comparaison est inadaptée car il s'agit alors d'une analyse syntaxique et non sémantique. De nombreuses différences syntaxiques sont rapportées sans que celles-ci soient pertinentes vis-à-vis d'une analyse sémantique. Par exemple, les valeurs numériques de deux pointeurs peuvent être différentes alors que le contenu des zones mémoire pointées est sémantiquement identique.

La clé de l'analyse sémantique d'un état système est donc de réussir à introspecter la mémoire dans le but de reconstruire la sémantique complète de l'état. Dans la suite de cette section, nous identifions les causes de ces différences syntaxiques et proposons pour chacune une solution permettant de retrouver des informations supplémentaires nécessaires à l'analyse sémantique.

2.1 *Overprovisioning*

L'*overprovisioning* consiste à fournir plus de mémoire que demandé. En C ou Fortran, la gestion de la mémoire dynamique est entièrement réalisée par le développeur. La fonction `malloc` permet d'allouer de la mémoire tandis que la fonction `free` permet sa libération. En C++, il est également possible d'utiliser ces fonctions, mais les opérateurs `new` et `delete` sont privilégiés. Ils effectuent automatiquement l'allocation ou la libération mémoire de façon transparente.

Pour éviter la fragmentation des données (et ainsi ne pas impacter les performances du système), une grande majorité des implémentations de `malloc` alloue des zones mémoires dont la taille est une puissance de deux. Quelle que soit la taille demandée pour chaque allocation, `malloc` retourne une zone mémoire dont la taille est une puissance de deux, supérieure ou égale à celle demandée au système. Ainsi, par exemple, une zone mémoire de 64 octets sera retournée par `malloc` pour toute demande d'allocation de 33 à 64 octets. Le système occupera alors la quantité exacte d'octets demandés mais laissera les octets supplémentaires inutilisés comme illustré sur la figure 2.2.

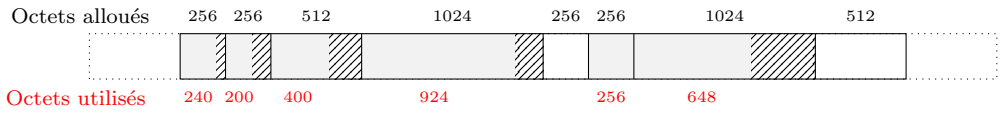

Figure 2.2 – Allocation de la mémoire dynamique par `malloc` avec *overprovisioning*.

Se pose alors le problème de la gestion des octets inutilisés. Ceux-ci peuvent contenir des informations aléatoires provenant d'allocation précédentes. Il est donc nécessaire de ne pas les inclure dans l'analyse sémantique, au risque de détecter des différences syntaxiques non pertinentes vis-à-vis de l'état courant du système.

Une première approche consiste à ignorer simplement ces octets lors de l'analyse. Grâce à une instrumentation de `malloc`, il est possible de sauvegarder pour chaque allocation la taille réellement demandée au système. Cependant, dans le cas de débordements mémoires (*buffer overflows*), celle-ci implique que des octets utilisés par l'état courant du système ne seraient pas pris en compte dans l'analyse sémantique. En effet, grâce à l'*overprovisioning* réalisé par `malloc`, des écritures peuvent être effectuées par l'application au delà de la zone mémoire demandée tout en restant dans la zone allouée sans corrompre son état. Il est donc nécessaire de tenir compte de ces éventuels octets lors de l'analyse.

Pour garantir une analyse sémantique correcte, même en cas de débordements mémoire, notre approche consiste à remplir de zéros chaque zone mémoire allouée dynamiquement par `malloc` avant toute utilisation par l'application. Ainsi, tout octet inutilisé est défini à 0 et les éventuels débordements mémoires sont préservés. L'implémentation de cette fonctionnalité est réalisée directement au sein de SimGridMC qui utilise déjà une version modifiée de `malloc`, basée sur `mmalloc`. Celle-ci permet notamment de gérer deux tas nécessaires au mécanisme de *backtracking* présenté dans la section 2.2.3 du chapitre 3 de la partie I.

L'*overprovisioning* est également présent au niveau des piles d'exécution des processus de l'application. Pour éviter les allocations et libérations mémoires avant et après chaque appel de fonction, une grosse zone mémoire faisant office de pile est allouée dès le début de l'exécution pour chaque processus. Lorsqu'une fonction se termine, l'espace fenêtre qui lui était alloué dans la pile du processus correspondant n'est pas vidé de son contenu. Bien que n'appartenant plus

à l'état courant de l'application, ces informations peuvent également générer des différences syntaxiques fausses si elles sont prises en compte au cours de l'analyse sémantique.

Plusieurs approches pour la gestion de cet *overprovisioning* dans chaque pile d'exécution peuvent être mises en œuvre. La première consiste à identifier la limite des zones mémoire actuellement utilisées dans les piles de chaque processus par l'état courant du système. Celle-ci correspond au pointeur de pile de chacune d'entre elles. Pour retrouver cette information, nous utilisons la bibliothèque libunwind [9] qui permet de dérouler le contenu d'une pile donnée et ainsi retrouver la chaîne des appels effectués par le système. Ainsi, lors de l'analyse sémantique, nous ne considérons que les informations contenues entre le début de la pile et le pointeur de pile qui représente la limite des données actuellement utilisées par le système. Les débordements mémoire ne sont pas à traiter dans ce contexte grâce à la protection (*Stack-Smashing Protector*) directement réalisée par le compilateur et le système d'exploitation contre ce type d'attaques de sécurité.

La seconde approche est plus complexe mais elle permet cependant de pallier d'autres verrous exposés dans la suite de cette section. Celle-ci consiste à réaliser la même opération que sur le tas, c'est-à-dire initialiser à zéro l'ensemble des octets alloués avant de donner accès à la zone mémoire à l'application. Cette approche nécessite de se placer au niveau du code assembleur et de le modifier pour intégrer la mise à zéro de la zone mémoire. Durant la phase de compilation, le compilateur parcourt le code source du programme et génère le code assembleur correspondant. Ce dernier est ensuite transmis à l'assembleur qui le transforme alors en code objet. L'opération de mise à zéro des zones mémoires dans la pile doit être réalisée avant que le code assembleur soit transmis pour la génération de code objet. Notre approche consiste alors à modifier ce premier code pour y intégrer l'opération de mise à zéro. Pour cela, nous créons dans un premier temps un script qui va prendre en entrée le code assembleur pour y intégrer nos instructions supplémentaires. Nous avons choisi d'ajouter ces instructions au début de chaque appel de fonction. La seconde étape consiste à modifier l'assembleur afin qu'il prenne en compte nos instructions supplémentaires. Pour cela, un script détermine dans un premier temps la taille de l'espace mémoire actuellement utilisé dans la pile par chaque fonction (fenêtre d'appel) en analysant le code assembleur, puis il préfixe les instructions d'assembleur en ajoutant la fonction qui met à 0 les octets correspondants. Dès lors, il est possible d'utiliser notre version modifiée de l'assembleur à la compilation pour générer le code objet avec la mise à zéro des fenêtres d'appel à chaque appel de fonction.

Cette seconde approche répondant à d'autres problématiques lors de l'analyse sémantique d'un état système, elle a été conservée au détriment de la première.

2.2 Alignement mémoire

L'alignement mémoire est la façon dont les données et les instructions en langage machine sont organisées en mémoire. Pour augmenter leurs performances, les processeurs alignent les données en mémoire avec des multiples de 2, 4 ou 8 octets, selon les caractéristiques du processeur. Il est alors possible de représenter la mémoire sous la forme d'un gros bloc divisé en plusieurs petits morceaux multiples de 2, 4 ou 8. L'objectif de l'alignement mémoire est d'éviter que des éléments soient « scindés » dans plusieurs petits morceaux mêmes contigus. Pour respecter cet alignement, le compilateur ajoute donc par défaut des octets de remplissage (*padding bytes*). Placés entre les variables, ceci permet d'accélérer le mouvement des données entre la mémoire et les registres du processeur. Nous présentons dans la figure 2.3 un exemple de struc-

9. `http://www.nongnu.org/libunwind/`

ture de données dans laquelle le compilateur ajoute automatiquement des octets de remplissage entre les éléments qui la composent pour assurer un alignement des données en mémoire.

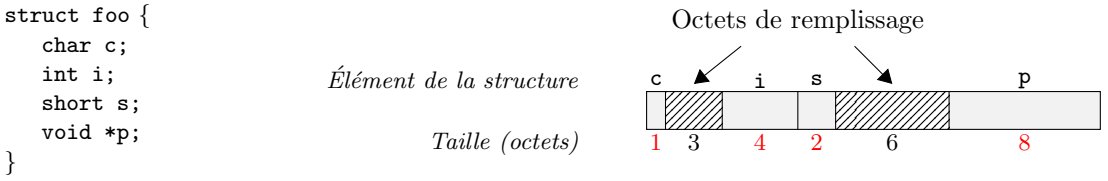

Figure 2.3 – Alignement en mémoire des données avec octets de remplissage.

Sur cet exemple, la structure de données `foo` contient quatre éléments : un caractère `c` qui occupe une taille de 1 octet en mémoire, un entier `i` qui occupe une taille de 4 octets en mémoire, un entier court `s` qui occupe une taille de 2 octets en mémoire et, enfin, un pointeur générique `p` qui occupe une taille de 8 octets en mémoire. Les tailles indiquées correspondent à une compilation pour un processeur 64 bits (x86_64), elles peuvent varier selon l'architecture. Les alignements mémoire dans ce contexte sont soumis aux règles suivantes, imposées par l'architecture du processeur :

- un élément de 1 octet peut être aligné à n'importe quelle adresse ;
- un élément de 2 octets doit être aligné sur une adresse multiple de 2 ;
- un élément de 4 octets doit être aligné sur une adresse multiple de 4 ;
- une structure de 1 à 4 octets de données doit être alignée de telle manière que la taille totale de la structure soit un multiple de 4 ;
- une structure de 5 à 8 octets de données doit être alignée de telle manière que la taille totale de la structure soit un multiple de 8 ;
- une structure de 9 octets de données et plus doit être alignée de telle manière que la taille totale de la structure soit un multiple de 16 ;

Si nous observons la représentation de la structure en mémoire, nous constatons que des octets de remplissage sont placés dans un premier temps entre `c` et `i`. En effet, si `i` était placé immédiatement après `c`, celui-ci ne serait pas aligné sur un multiple de 4 comme défini précédemment. Le processeur ajoute donc 3 octets de remplissage, ce qui permet de placer `i` à un décalage (*offset*) de 4 par rapport au début de la structure `foo`. De même, 6 octets de remplissage sont placés entre `s` et `p`, permettant d'aligner le pointeur `p` à un décalage de 16 (multiple de 8) par rapport au début de la structure. La présence de ces octets de remplissage est également observable lorsque l'on demande la taille de la structure en mémoire à travers la fonction `sizeof` qui retourne une taille de 24 octets au lieu de 15, comme attendu.

Tout comme avec l'*overprovisioning*, ces octets supplémentaires, dont le contenu est inconnu, peuvent induire des différences syntaxiques lors de l'analyse sémantique alors qu'ils ne caractérisent pas l'état courant du système. Il est donc nécessaire de les ignorer lors de l'analyse ou de les « neutraliser ».

Ces octets de remplissage sont à la fois présents dans le tas et les piles des processus de l'application. Toutefois, notre précédente approche qui consiste à définir à zéro l'ensemble des octets pour toute nouvelle allocation (que ce soit dans le tas ou dans une pile d'exécution) permet de répondre également à ce problème.

À noter qu'il est possible, avec certains compilateurs, de désactiver ces octets de remplis-

sage présents par défaut, en mettant l'alignement de tous les éléments agrégés à une limite d'octets spécifique. Ceci peut être réalisé grâce à la primitive `#pragma pack(1)`, où `1` signifie que l'alignement s'effectue sur un octet, éliminant ainsi l'ajout automatique d'octets de remplissage. Cependant, cette approche peut engendrer des pertes de performances significatives dans l'exécution du système. Le compilateur doit en effet générer du code supplémentaire pour accéder aux données non-alignées que le processeur ne peut pas charger en moins de deux cycles dans ses registres, et ce pour chaque variable concernée.

2.3 Allocations mémoire dynamiques

Par définition, les allocations mémoire dynamiques ne concernent que le tas. Il s'agit toutefois de l'obstacle le plus complexe à traiter dans le cadre de l'analyse sémantique d'un état système.

Le tas correspond à une zone mémoire dont la taille varie au cours de l'exécution du système (elle peut s'agrandir au besoin). Cette zone mémoire est divisée en blocs de taille fixe. Ces blocs peuvent être également divisés en fragments de tailles identiques, dans le cas de petites allocations. La bibliothèque malloc en charge de sa gestion maintient deux informations importantes : la liste des blocs ou fragments actuellement alloués et ceux inoccupés. Lorsqu'une nouvelle allocation est demandée par l'application, `malloc` regarde alors dans la liste des zones disponibles s'il existe un ou plusieurs blocs ou fragments contigus correspondant à la taille demandée. Si oui, les zones mémoires correspondantes sont directement allouées et ajoutées à la liste des zones actuellement occupées, sinon le tas est agrandi. Lors d'une libération avec `free`, les blocs ou fragments désalloués sont marqués comme libres et ajoutés à la liste des zones mémoires disponibles.

Pour faciliter et accélérer la recherche des zones mémoires disponibles pour de nouvelles allocations, malloc sauvegarde l'index de la dernière zone libérée. Chaque recherche de nouvelle allocation est alors réalisée à partir de cet index et non pas depuis le début du tas. Ainsi, si une nouvelle allocation est réalisée après une libération mémoire, si la taille demandée correspond exactement à celle qui vient d'être libérée, la zone mémoire correspondante sera immédiatement ré-allouée. Si cette allocation est réalisée entre deux libérations mémoire, selon l'ordre dans lequel les libérations mémoires sont réalisées, ceci peut mener à deux adresses différentes pour la nouvelle allocation. C'est ainsi que pour deux exécutions possibles d'un même système avec la même quantité d'allocations ou libérations mémoires et les mêmes données allouées ou libérées, il est possible d'obtenir deux tas dont le contenu est identique mais dont la répartition en mémoire est différente. Les tas sont alors sémantiquement identiques mais syntaxiquement différents. Une telle situation est particulièrement courante dans le contexte des systèmes distribués où l'ordre d'exécution des processus peut différer entre deux exécutions, modifiant ainsi l'ordre des allocations et libérations mémoire réalisées. Nous présentons dans la figure 2.4 un exemple de tas sémantiquement identiques mais syntaxiquement différents.

Dans cet exemple, si l'on compare les blocs deux à deux entre les deux tas, l'analyse sera identique pour les blocs `0x10` et `0x20`. Les blocs `0x30`, `0x40` et `0x50` seront eux déclarés syntaxiquement différents. En effet, si l'on compare dans un premier temps, le contenu des blocs `0x30` entre eux ainsi que les blocs `0x40`, ceux-ci sont différents. Concernant le bloc `0x50`, celui-ci contient deux pointeurs sur les deux autres blocs, Dans le premier tas, le premier pointeur pointe sur le bloc `0x40` et le second sur le bloc `0x30`. Cet ordre est inversé dans le cas du second tas. Dès lors, une simple analyse syntaxique rapportera que les adresses pointées sont différentes pour chaque pointeur entre les deux tas. Or, si l'on compare les informations contenues dans les zones pointées, celles-ci sont pourtant identiques. Le premier pointeur pointe sur une zone mémoire contenant `gcc` tandis que le second mène sur une zone mémoire contenant `ffe`.

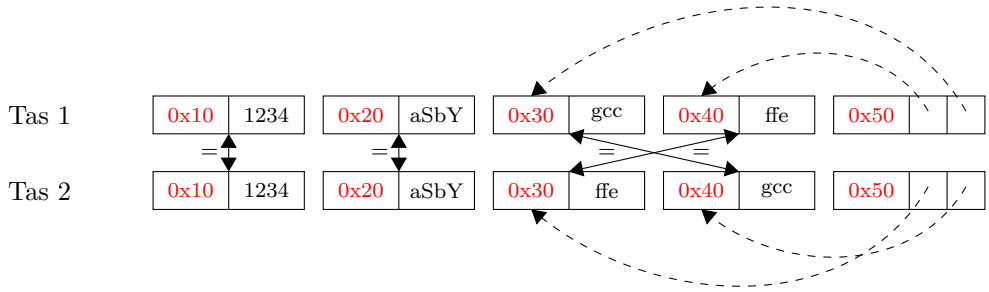

Figure 2.4 – Deux tas sémantiquement identiques mais syntaxiquement différents.

Dans [Ios01], l'auteur propose un algorithme de mise en forme canonique du tas (*heap canonicalization*), qui réordonne les blocs, répondant ainsi au problème présenté. Ainsi, la comparaison des blocs du tas peut-être réalisée linéairement (le bloc 1 du premier tas est comparé avec le bloc 1 du second tas, le bloc 2 avec l'autre bloc 2, *etc*). Cette technique repose toutefois sur la présence d'un mécanisme de type ramasse-miettes et travaille sur un langage dans lequel l'ensemble des références des allocations mémoires est accessible au système. Dès lors, un algorithme de *mark-and-sweep* est appliqué pour chacune des variables. Le graphe de parcours correspondant est déterministe par construction et les blocs sont réordonnés à la volée pour arriver à une forme canonique, sur laquelle s'appuie ensuite l'analyse sémantique.

Cette approche ne peut cependant être appliquée dans notre contexte car elle suppose que le système vérifié utilise un des ramasse-miettes existants pour les programmes C/C++. Il est possible de retrouver les références des variables locales et globales dans les informations de débogage mais celles-ci seraient potentiellement incomplètes au niveau du tas, faute d'informations sémantiques sur le contenu des blocs. À cause de ces références manquantes, il est impossible de mettre en œuvre le ré-ordonnancement physique des blocs mémoires.

Pour s'assurer que l'analyse sémantique du contenu du tas se fait bien entre blocs contenant le même type d'informations, sans réordonner physiquement la mémoire, il est nécessaire de connaître le type des données présentes dans chacun. Dans le cas d'octets correspondant à un pointeur, l'analyse se doit de poursuivre. Pour cela, notre approche s'appuie en partie sur les informations de débogage dans le format DWARF [10]. Ce format est utilisé notamment par de nombreux compilateurs ou débogueurs tel que gdb. Ces informations sont obtenues lorsqu'un programme est compilé avec l'option correspondante (`-g` dans le cas de gcc). La description complète de l'ensemble des variables contenues dans l'état du système ainsi que les différentes fonctions qui composent un programme y sont données. Pour chaque variable, nous obtenons notamment son nom, son type, sa taille et son adresse en mémoire. Dans le cas de variables globales, l'adresse globale est directement fournie tandis qu'une expression permet de retrouver l'endroit où se trouve l'adresse d'une variable locale selon le pointeur d'instruction courant. Nous présentons dans la figure 2.5 un exemple d'informations de débogage au format DWARF obtenues à partir d'un programme simple. L'ensemble des informations fournies permet alors de reconstruire avec précision la sémantique de chaque octet de la mémoire soumise à une analyse.

DWARF ne suffit toutefois pas car nous nous intéressons aux allocations mémoires dyna-

10. `http://www.dwarfstd.org/`

```
1  int global_variable;
2
3  int client(int argc, char *argv[])
4  {
5    global_variable = argc;
6
7    pid_t pid = getpid();
8    char *mailbox = bprintf("%s", argv[1]);
9
10   printf("I'm %s and my pid is %d", mailbox, pid);
11 }
```

```
<1><761>: Abbrev Number: 22 (DW_TAG_subprogram)
    <763>   DW_AT_name        : (indirect string, offset: 0xb335): client
    <769>   DW_AT_prototyped  : 1
    <76a>   DW_AT_type        : <0xc5>
    <76e>   DW_AT_low_pc      : 0x401f90
    <776>   DW_AT_high_pc     : 0x4026cf
    <77e>   DW_AT_frame_base  : 0x16c (location list)
 <2><787>: Abbrev Number: 23 (DW_TAG_formal_parameter)
    <788>   DW_AT_name        : (indirect string, offset: 0x577d): argc
    <78e>   DW_AT_type        : <0xc5>
    <792>   DW_AT_location    : 3 byte block: 91 ec 7e  (DW_OP_fbreg: -148)
 <2><796>: Abbrev Number: 23 (DW_TAG_formal_parameter)
    <797>   DW_AT_name        : (indirect string, offset: 0x5870): argv
    <79d>   DW_AT_type        : <0x2c1>
    <7a1>   DW_AT_location    : 3 byte block: 91 e0 7e  (DW_OP_fbreg: -160)
 <2><7a5>: Abbrev Number: 24 (DW_TAG_variable)
    <7a6>   DW_AT_name        : (indirect string, offset: 0x109c): pid
    <7ac>   DW_AT_type        : <0xc5>
    <7b0>   DW_AT_location    : 2 byte block: 91 6c   (DW_OP_fbreg: -20)
 <2><7b3>: Abbrev Number: 24 (DW_TAG_variable)
    <7b4>   DW_AT_name        : (indirect string, offset: 0x612): mailbox
    <7ba>   DW_AT_type        : <0x2bb>
    <7be>   DW_AT_location    : 2 byte block: 91 60   (DW_OP_fbreg: -32)
<1><9eb>: Abbrev Number: 32 (DW_TAG_variable)
    <9ec>   DW_AT_name        : global_variable
    <9f1>   DW_AT_type        : <0xc5>
    <9f5>   DW_AT_external    : 1
    <9f6>   DW_AT_location    : 9 byte block: 3 24 57 60 0 0 0 0 0  (DW_OP_addr: 605724)
```

Figure 2.5 – Informations de débogage au format DWARF d'un programme simple.

miques réalisées au cours de l'exécution et donc inconnues à la compilation lorsque les informations de débogue sont générées. Ces informations nous permettent de savoir quelles variables et le type de données associées sont en mémoire mais pas leur adresse en mémoire avant exécution. Nous avons donc besoin dans un premier temps de les retrouver en mémoire à chaque instant de l'exécution, puis nous exploitons les informations de débogage pour interpréter le contenu des zones correspondantes. C'est ainsi que nous effectuons un parcours et une analyse du tas à la volée depuis les variables locales et globales pointant sur celui-ci. Grâce à DWARF, nous connaissons pour chacune d'entre elles le type de données et pouvons suivre les éventuels pointeurs contenus dans chaque zone mémoire analysée et qui mènent vers d'autres zones du tas. Il s'agit donc également d'une approche de *mark-and-sweep* où chaque bloc analysé est marqué comme tel. Ce parcours se termine lorsque tous les blocs mémoires ont été analysés ou dès qu'une différence est détectée lors de l'analyse sémantique.

Il s'agit cependant d'une heuristique car l'adresse de certaines variables locales n'est pas

accessible à certains instants de l'exécution (d'après le pointeur d'instruction). Il peut alors exister des blocs mémoires inaccessibles depuis les variables locales et globales mais qui doivent bien sûr être inclus à l'analyse sémantique. De même, nous pouvons être confrontés à des pointeurs génériques (`void *`) pour lesquels nous n'avons aucune information sur la sémantique des données pointées ou à des alias de type masquant une partie des informations nécessaires à l'analyse. Nous réalisons alors une analyse en version boîte noire en comparant octet par octet les blocs concernés. En cas de différence syntaxique, nous effectuons un alignement mémoire afin de déterminer s'il s'agit d'un pointeur valide. Si oui, nous suivons ce pointeur, sinon une différence est rapportée et l'analyse est arrêtée. De cette manière, nous garantissons que l'ensemble des blocs mémoires du tas est analysé.

Il existe enfin une autre difficulté liée à l'utilisation de pointeurs et leur libération. En effet, la valeur numérique d'un pointeur est inchangée après la libération de la zone mémoire pointée. Deux cas de figures se posent alors lorsque l'algorithme précédemment décrit déréférence un pointeur invalide car libéré : la zone mémoire n'a pas été ré-allouée entre sa libération et l'analyse sémantique, elle n'est donc pas prise en compte et aucun problème ne se pose. Si toutefois celle-ci est ré-allouée, elle peut contenir des informations totalement différentes alors que le pointeur est toujours suivi par l'analyse, elle sera donc incluse dans l'analyse sémantique mais mal interprétée. Il s'agit dans ce cas de *dangling pointers* (« pointeurs pendouillants »). Une bonne pratique en programmation veut qu'après toute libération mémoire, un pointeur soit remis à NULL, mais il ne s'agit pas d'une obligation. Dans le cas où la mémoire n'est pas ré-allouée, ceci est facilement détectable. Cela l'est beaucoup moins dans le cas contraire. Notre approche nécessite donc la détection manuelle de ces pointeurs par l'utilisateur afin de garantir une interprétation correcte du contenu des zones mémoires lors de l'analyse sémantique. Cette détection doit être réalisée par l'utilisateur en amont de l'analyse sémantique, en utilisant Valgrind [11] par exemple. L'analyse sémantique est ensuite réalisée automatiquement sans l'intervention de l'utilisateur.

2.4 Variables locales non-initialisées

Après un problème spécifique au tas, nous avons été confrontés à un problème qui ne concerne cette fois que les piles des processus : les variables locales non-initialisées. En effet, lorsqu'une variable est déclarée, même si elle n'est pas encore initialisée, celle-ci est tout de même présente dans la pile. Ainsi, d'après la liste des variables locales associées à chaque fenêtre d'appel obtenue avec DWARF, nous pouvons être amenés à comparer des variables non-initialisées, sans aucune possibilité de les distinguer. Un espace mémoire leur étant réservé dans la pile mais non initialisé, les informations qui y sont présentes sont arbitraires. Comme dans le cas des octets de remplissage, il s'y trouve généralement les données d'une fenêtre d'appel précédente qui n'ont pas été supprimées au retour de la fonction correspondante. Nous présentons dans la figure 2.6 un exemple de programme utilisant des variables locales non-initialisées et leur contenu en mémoire à différents instants d'exécution.

Dans cet exemple, la fonction `f()` est appelée dans un premier temps (ligne 21). Celle-ci contient un tableau d'entiers (ligne 5) dont le contenu est d'abord affiché (lignes 7 à 9) avant d'être initialisé (lignes 11 et 12). Le résultat de ce premier affichage présente alors des données aléatoires. Ensuite, cette même fonction est à nouveau appelée (ligne 22). Le résultat de l'affichage semble alors « correct » même si à nouveau le tableau d'entiers n'a pas été initialisé. Ceci est dû à la non-suppression des données de l'appel de fonction précédente après le retour

11. `http://valgrind.org/`

```
1  #include <stdio.h>
2
3  void f() {
4    int i;
5    int data[16];
6
7    for(i=0; i!=16; ++i)
8      printf("%i ", data[i]);
9    printf("\n");
10
11   for(i=0; i!=16; ++i)
12     data[i] = i;
13 }
14
15 void g() {
16   int i, j, k, l, m, n, o, p;
17   printf("%i %i %i %i %i %i %i %i\n", i, j, k, l, m, n, o, p);
18 }
19
20 int main(int argc, char** argv) {
21   f();
22   f();
23   g();
24   return 0;
25 }
```

Résultat d'une exécution :

```
-1 0 -812203224 32767 -406470232 32655 -400476992 32655 -400465496 32655 0 0 1 0 4195997 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 0 0 15774463 15 14 13 12
```

Figure 2.6 – Introspection mémoire de variables locales non initialisées.

de fonction. S'agissant de la même fonction qui est appelée, celle-ci occupe exactement le même espace en mémoire dans la pile, exactement au même endroit que lors de l'appel précédent. Ce dernier finissant par initialiser le tableau avant le retour de fonction, nous retrouvons ces informations lors du nouvel appel. Cet exemple se termine par l'appel de la fonction `g()` qui contient également des variables locales non initialisées (ligne 16) qui sont ensuite affichées (ligne 17). Tout comme pour le premier appel à `f()`, le résultat de cet affichage présente des données arbitraires. On retrouve en effet certaines valeurs des variables locales précédentes.

Pour éviter ce cas de figure et donc l'analyse de données aléatoires, une solution est à nouveau de mettre à zéro tous les octets d'une fenêtre d'appel au moment de l'appel d'une fonction ou bien à son retour. C'est ainsi que l'approche présentée dans la section 2.1 sur l'*overprovisioning* permet également de répondre à cette problématique de variables locales non-initialisées. Si l'on reprend l'exemple précédent, nous obtenons bien uniquement des 0 lors de l'affichage du contenu de la pile sur les variables non-initialisées.

2.5 Différences non pertinentes

Il subsiste enfin des informations qui appartiennent effectivement à l'état du système mais qui peuvent être considérées comme non-pertinentes lors de l'analyse sémantique. Nous pouvons noter par exemple le compteur des messages envoyés au cours de l'exécution, les PIDs des processus exécutés qui peuvent changer lors de l'exploration d'une nouvelle exécution possible, *etc.* Il est donc nécessaire de prendre en compte la présence de ces informations en proposant à

l'utilisateur un moyen de les exclure de l'analyse.

Pour cela, nous proposons un mécanisme permettant d'ignorer explicitement certaines variables ou directement certaines zones mémoires. L'utilisateur peut ainsi ignorer tout information qu'il considère comme non pertinente pour l'analyse. Les zones mémoires spécifiées par l'utilisateur sont alors soit marquées comme telles sous forme de méta-donnée incluse dans notre implémentation de malloc, soit supprimées de la liste des variables à analyser obtenues avec DWARF dans le cas de variables locales. Durant l'introspection, cette information est exploitée pour ne pas soumettre les zones mémoires correspondantes à une analyse sémantique.

3 Compression mémoire d'un état système

Un état système est composé du tas de l'application, de ses variables globales, des piles des processus ainsi que l'état du réseau ; l'ensemble de ces informations est donc entièrement sauvegardé à chaque étape de l'exploration. Selon les applications étudiées, ceci représente plusieurs dizaines de méga-octets par état. C'est ainsi que même sur des petites applications avec quelques dizaines de milliers d'états, la consommation mémoire devient très rapidement un frein à la mise en œuvre de la vérification dynamique formelle d'applications réelles. Or, pour de nombreuses applications, seulement une petite partie de l'état du système change entre deux étapes de l'exploration. Pour certaines applications, 99% des pages mémoires ne changent pas entre deux états successifs. Une approche pour réduire la consommation mémoire est donc de détecter les pages identiques entre deux états, dans le but de ne garder qu'une seule copie d'entre elles.

Le *Kernel SamePage Merging (KSM)* [AEW09] est une approche automatique qui permet de partager les pages mémoires identiques entre plusieurs processus. Pour cela, une détection des pages mémoires identiques est réalisée dans un premier temps. Si plusieurs pages identiques sont identifiées, elles sont ensuite fusionnées en une seule page qui est partagée entre les processus correspondants. L'unique page obtenue est marquée en copie sur écriture (*copy-on-write*) de façon à ce que si un processus souhaite y accéder et surtout la modifier, une nouvelle copie privée est créée. Ainsi, si la page mémoire n'est jamais modifiée par l'un des processus, une unique sauvegarde est conservée pour tous, réduisant l'espace mémoire utilisé par l'application. Cependant, une difficulté dans cette approche est la quantité de calculs induite. En effet, si cela permet de réduire considérablement la consommation mémoire, cette approche peut augmenter l'utilisation du CPU et donc impacter les performances lors de l'exécution de l'application. Si les pages identiques sont constamment modifiées, il faut à chaque fois faire une nouvelle copie. Le comportement est alors le même que sans le KSM mais avec en plus la phase de détection des pages identiques et leur fusion qui n'est que temporaire.

Ce mécanisme n'étant pas activé par défaut sur les systèmes, il est nécessaire dans un premier temps de spécifier les régions de la mémoire qui seront soumises à cette gestion puis de le lancer en super-utilisateur sur le système d'exploitation. Bien que non-intrusif à première vue, cette approche nécessite une modification de l'environnement sur lequel le système étudié est exécuté avec les droits de super-utilisateur, ce qui n'est pas toujours possible. De plus, elle nécessite que les pages mémoires soient dans un premier temps entièrement copiées avant d'être soumises à comparaison avec les pages existantes pour détecter les contenus identiques. C'est seulement après que la fusion des données est réalisée en une unique page. Une approche plus efficace dans le contexte de la sauvegarde d'états système serait d'éviter cette phase de copie avant fusion.

Notre approche consiste à considérer un état système comme un ensemble de pages mémoires

indépendantes. Nous établissons alors une politique de sauvegarde à ce niveau de granularité.

Lorsqu'un nouvel état de l'exploration doit être sauvegardé, nous déterminons dans un premier temps l'ensemble des pages mémoires qui le composent. Ensuite, pour chaque page mémoire, nous évaluons si celle-ci est identique à une des pages mémoires précédemment sauvegardées. Cette détection est basée sur la génération d'un *hash* d'après le contenu de la page. S'il s'agit d'une nouvelle page, celle-ci est indexée et ajoutée à l'ensemble des pages déjà sauvegardées. Si la page est identique à une autre page existante, une seule copie est conservée et l'index de la page correspondante est partagé avec le nouvel état. Ainsi, un état système correspond à un ensemble d'index de pages mémoire sauvegardées. Pour le premier état de l'exploration, l'ensemble des pages mémoires correspondantes est conservé, l'index de chaque page référencée est ensuite associé à l'état système correspondant. Ces premières pages sauvegardées constituent la base de données de référence pour la recherche de pages identiques pour les états suivants. Ensuite, seules les nouvelles pages mémoires différentes sont sauvegardées et ajoutées à l'ensemble existant. Enfin, sachant qu'il est possible de limiter le nombre d'états pouvant être sauvegardés au cours de l'exploration, un compteur de référence est associé à chaque page mémoire sauvegardée pour déterminer lorsqu'une page n'est plus partagée par aucun état et peut donc être supprimée.

À noter que dans notre contexte, le marquage des pages mémoires en copie sur écriture n'est pas nécessaire car elles ne sont partagées qu'entre les états systèmes sauvegardés au cours de l'exploration mais pas avec l'état courant de l'application exécutée. Nous ne subissons donc pas les éventuelles pertes de performances présentes avec l'approche KSM.

4 Limites

La mise en œuvre d'une heuristique parmi les solutions apportées implique que certains cas ne pourront être correctement traités lors de la comparaison d'états système.

Parmi eux, il y a notamment la gestion des pointeurs génériques (de type `void *` en C). Il nous est impossible dans ce contexte de déterminer le type des données analysées et par conséquent de réaliser une analyse sémantique sur celles-ci, justifiant ainsi la mise en œuvre de l'heuristique. Face à ce cas de figure, nous travaillons à nouveau dans un contexte de boîte noire, et comparons les octets deux à deux, indépendamment des autres et sans interprétation de leur sémantique. En cas de différence syntaxique, l'unique analyse avancée pouvant être réalisée est l'évaluation d'une différence de pointeurs. Par un mécanisme d'alignement mémoire et de déréférencement, nous essayons d'extraire l'éventuelle adresse de la zone mémoire pointée et de comparer ensuite ces nouvelles zones. Cependant, il est possible que la différence ne provienne pas d'un pointeur mais qu'une zone mémoire valide soit obtenue par déréférencement, la comparaison est alors incohérente.

Dans un contexte similaire, l'analyse sémantique ne peut être réalisée face à des `union`. Même si elles ne sont pas anonymes, le principe de ce type de structure de données étant d'allouer la taille maximale, c'est-à-dire la taille de l'élément le plus grand inclus dans l'union, il est impossible d'interpréter sémantiquement chaque octet de cette union. Nous entrons alors à nouveau dans une boîte noire, si bien que seule une analyse syntaxique linéaire entre les deux zones mémoires concernées peut être réalisée. Dès lors, des interprétations incorrectes peuvent survenir.

De plus, il n'est actuellement pas possible à l'utilisateur de définir des interprétations sémantiques sur certains types de données alors que celles-ci peuvent révéler des différences non-pertinentes de son point de vue. Par exemple, il est possible que l'ordre des éléments dans une liste ne soit pas à considérer lors de la comparaison. Dès lors que le contenu sémantique

global est identique, nous pourrions souhaiter considérer l'ensemble comme identique. Notre approche ne traite toutefois pas ce cas de figure. En effet, sur cet exemple, l'interprétation des données de débogage nous permet de détecter que la zone mémoire analysée contient une liste, la taille de cette liste ainsi que le type des éléments de la liste et les informations qui sont alors associées à chaque élément. Mais notre algorithme mis en œuvre va comparer chaque élément de la liste indépendamment des autres. Si une différence sémantique est détectée entre le $i^{\text{ème}}$ élément de la liste du premier état et celui de la liste du second état, les états seront déclarés comme sémantiquement différents.

Enfin, comme nous l'avons précédemment évoqué, aucune égalité sémantique ne peut être détectée en cas de fuite mémoire. Avant toute analyse sémantique, nous commençons par comparer le nombre d'octets actuellement occupés en mémoire. Si celui-ci est différent, il est impossible qu'une égalité sémantique soit détectable. De même, en cas de « pointeurs pendouillants », nous pouvons alors interpréter de façon incorrecte des données si la zone mémoire a été ré-allouée pour un autre type de données. Dans ces deux cas, une intervention de l'utilisateur est nécessaire avant de lancer notre analyse sémantique pour éliminer ces cas de figure.

Nous évaluons, dans le chapitre suivant, l'ensemble des travaux présentés dans ce chapitre, à travers trois expérimentations : l'exploration exhaustive d'applications MPI extraites de la suite de tests de MPICH, avec réduction basée sur la détection des états visités, l'exploration exhaustive de protocoles cycliques finis ou infinis et l'étude de la terminaison de programmes issus de la compétition de vérification SV-COMP (`http://sv-comp.sosy-lab.org`). Malgré les limites précédemment définies, les résultats obtenus à travers ces expériences démontrent toutefois l'efficacité de notre approche.

Chapitre 3

Évaluation expérimentale

Dans ce chapitre, nous évaluons l'approche d'analyse sémantique dynamique d'un état système par introspection mémoire, présentée dans le chapitre précédent. Pour cela, nous mettons en œuvre trois types d'expériences.

Nous commençons par réaliser une exploration exhaustive d'applications MPI, c'est-à-dire sans propriété à vérifier, dont l'exécution est finie. Ces applications sont issues des tests d'intégration de MPICH. Nous évaluons sur celles-ci la réduction de l'espace d'états basée sur la détection d'états sémantiquement identiques.

La seconde expérience évalue la détection d'états identiques dans le contexte de protocoles cycliques dont l'exécution est finie ou infinie. Il s'agit donc ici d'identifier dans un premier temps les cycles pour réduire l'espace d'états dans le cas d'exécutions finies. Dans le cas d'exécutions infinies, cette détection permet de garantir une exploration exhaustive de l'ensemble des états sémantiquement différents.

Enfin, nous réalisons en troisième partie la vérification de la terminaison de programmes à travers la détection de cycles de non-progression dans des applications réelles issues du *benchmark* de la compétition de vérification SV-COMP (`http://sv-comp.sosy-lab.org`).

1 Réduction de l'espace d'états

Dans cette première partie de l'évaluation, nous nous intéressons à des applications MPI issues des tests d'intégration de MPICH. MPICH (MPI CHameleon) est l'une des implémentations les plus populaires du standard MPI, à l'origine notamment de IBM MPI, Intel MPI, Cray MPI ou Microsoft MPI. Sa version actuelle MPICH3 implémente le standard MPI-3. Au sein de ce développement, un ensemble de tests d'intégration est proposé afin d'évaluer si toute implémentation donnée respecte le standard MPI. Il s'agit de petites applications (quelques centaines de lignes de code chacune) écrites en C ou Fortran qui permettent d'évaluer précisément chaque fonctionnalité du standard.

Nous avons sélectionné parmi ces applications plusieurs d'entre elles issues de différentes catégories de tests : les communications point-à-point (leur création et leur échange), les collectives (en C et en Fortran) et la création de groupes. Celles-ci sont intégrées sans modification au sein de SimGrid et peuvent être exécutées grâce à SMPI. Pour chaque application, nous effectuons une exploration exhaustive, avec et sans détection des états identiques, ainsi qu'avec et sans compression mémoire. Aucune propriété particulière n'est spécifiée, seules les détections de *deadlocks* et *cycles de non-progression* sont automatiquement réalisées. Ces expériences ont été réalisées sur une machine avec un processeur Intel(R) Xeon(R) CPU E7540 @ 2.00GHz, RAM

512GiB, 48 cœurs, et un système d'exploitation Debian wheezy environnement 3.2.0-4-amd64. Les résultats sont présentés dans la table 3.1.

Application (langage)	**# P**	**Sans réduction**			**Avec réduction**			**Avec réduction et compression mémoire**		
		# États	**Temps**	**Mémoire**	**# États**	**Temps**	**Mémoire**	**# États**	**Temps**	**Mémoire**
bcasttest (C)	3	> 1 million	> 24 h	-	4 823	18 min 31 s	37 Go	4 823	25 min 23 s	1.01 Go
bcastzerotype (C)	5	12 135 948	1 h 22 min	0.35 Go	4 734	6 min 35 s	5.83 Go	4 734	6 min 50 s	0.84 Go
	6	> 263 millions	> 24 h	-	56 054	9 h 03 min	62.4 Go	56 054	10 h 02 min	7.16 Go
commcreate1 (C)	4	102 289	44 s	0.35 Go	1 556	1 min 19 s	2.48 Go	1 556	1 min 28 s	0.49 Go
	5	12 710 034	1 h 23 min	0.35 Go	8 359	23 min 22 s	10.56 Go	8 359	25 min	1.48 Go
	6	> 274 millions	> 24 h	-	99 235	23 h 14 min	108.36 Go	99 235	24 h 55 min	14.18 Go
dup (C)	2	907	2 s	0.35 Go	81	2 s	0.45 Go	81	2 s	0.35 Go
	3	138 678	43 s	0.35 Go	405	5 s	0.77 Go	405	7 s	0.36 Go
	4	78 082 843	7 h 36 min	0.35 Go	2 352	1 min 04 s	2.99 Go	2 352	1 min 16 s	0.62 Go
	5	> 276 millions	> 24 h	-	39 263	6 h 32 min	47.63 Go	39 263	7 h 06 min	5.83 Go
groupcreate (C)	4	102 289	31 s	0.35 Go	1 205	40 s	1.86 Go	1 205	44 s	0.44 Go
	5	12 710 034	1 h 22 min	0.35 Go	6 237	11 min	7.62 Go	6 237	11 min 21 s	1.21 Go
	6	> 272 millions	> 24 h	-	80 878	16 h 15 min	89.31 Go	80 878	17 h 35 min	11.47 Go
inplacef (Fortran)	3	> 182 millions	> 24 h	-	2 941	1 min 07 s	3.87 Go	2 941	1 min 15 s	0.73 Go
op_commutative (C)	3	358	2 s	0.35 Go	94	2 s	0.46 Go	94	2 s	0.35 Go
	4	102 289	31 s	0.35 Go	1 545	1 min 16 s	2.47 Go	1 545	1 min 20 s	0.48 Go
	5	12 710 034	1 h 23 min	0.35 Go	10 998	48 min 42 s	14.25 Go	10 998	53 min 29 s	1.79 Go
sendrecv2 (C)	2	> 156 millions	> 24 h	-	1 877	28 s	3.25 Go	1 877	30 s	0.49 Go

Table 3.1 – Exploration exhaustive d'applications MPI issues des tests d'intégration de MPICH3.

Deux faits importants peuvent être observés d'après les résultats de la table 3.1. Tout d'abord, bien que les applications étudiées soient de petites applications avec très peu de processus, le nombre d'états lors d'une exploration exhaustive sans réduction augmente de façon exponentielle, au fur et à mesure que le nombre de processus augmente, atteignant plusieurs millions d'états à chaque fois, voire des milliards pour certaines. Toutefois, si l'on observe le nombre d'états sémantiquement différents dans cet ensemble, calculé grâce à une exploration avec détection des états identiques, cette valeur chute à chaque fois à quelques dizaines de milliers d'états. Il est ainsi possible de garantir une exploration, et donc une vérification exhaustive de plusieurs applications, en quelques minutes ou heures contre plusieurs heures voire plusieurs jours sans réduction.

Par exemple, l'application `dup` nécessite 7 heures et 36 minutes pour explorer entièrement l'espace d'états avec 4 processus sans réduction contre 1 minute avec réduction. C'est ainsi qu'une exploration sur 5 processus a pu être effectuée en 6 h et 32 minutes grâce à la réduction alors qu'au moins 24 heures sont nécessaires sans réduction (expérience stoppée après 24 heures). Dans le cas de l'application `sendrecv2`, nous n'avons pas augmenté le nombre de processus car son comportement est déterminé uniquement pour 2 processus. Cependant, même avec seulement 2 processus, une exploration exhaustive n'a pu être réalisée sans réduction en moins de 24 heures,

contre quelques secondes avec la détection des états identiques. Cette analyse est également présente dans le cas d'applications Fortran avec notamment l'application `inplacef`.

Concernant la consommation mémoire, pour chacune des expériences, le nombre d'états sauvegardés était uniquement limité par la quantité de mémoire disponible sur la machine utilisée, en l'occurrence 512 Go. L'ensemble des expériences réalisées nécessite moins de 16 Go de mémoire grâce à la compression mémoire, rendant ainsi l'exploration réalisable sur une machine personnelle classique. Sans cette compression mémoire, la consommation mémoire varie de quelques Go à plusieurs dizaines voire centaines dans le cas de l'application `commcreate1`. La limite de ces expériences était dans un premier temps basée sur le temps mais si celui-ci n'est plus limité à 24 heures, certaines explorations ne peuvent être réalisée faute de mémoire suffisante. La compression mémoire permet donc de renforcer la réduction, avec une augmentation du temps de vérification de seulement 10% en moyenne, mais également de rendre possible certaines explorations en un temps et une quantité de mémoire limités.

Du point de vue des performances, si la sauvegarde mais surtout la comparaison des états réduit considérablement le nombre d'états explorés en un temps donné, certaines expériences (`commcreate1`, `groupcreate` et `op_commutative` à 4 processus) nécessitent un temps de vérification légèrement supérieur par rapport à une exploration sans réduction. La gestion des états sauvegardés ainsi que leur comparaison restent coûteuses en temps si bien qu'il existe un seuil en dessous duquel la réduction devient moins efficace en terme de temps de vérification. Toutefois, la réduction de l'espace d'états reste suffisamment importante dans la grande majorité des cas pour rendre l'approche efficace voire indispensable.

Nous démontrons donc par ces expériences l'importance de la détection des états identiques pour la réduction de l'espace d'états mais également l'efficacité de l'approche présentée dans le chapitre 2 de cette partie II.

Dans les expériences suivantes, nous nous intéressons à des applications aux exécutions infinies mais dont le comportement est cyclique et démontrons comment la détection des états identiques permet une exploration exhaustive de celles-ci.

2 Exploration exhaustive de protocoles cycliques finis ou infinis

Parmi les applications distribuées issues des systèmes distribués, il existe des applications ou protocoles tels que Chord ou Pastry dont le comportement est cyclique, soumis à des évènements périodiques. Certaines de ces applications peuvent s'exécuter de façon infinie, même si aucun nouvel évènement n'affecte l'état du système. Dans le cas d'applications dont l'exécution est cyclique mais infinie, il n'est pas possible de garantir une vérification exhaustive dès lors que ces cycles ne sont pas clairement identifiés. En l'absence de détection de ces cycles, l'exploration d'un seul chemin sera elle aussi infinie.

La première approche pour vérifier ce type d'application est de modifier le comportement initial de celle-ci en rendant l'exécution finie. Ainsi, il est garanti que l'exploration se terminera un jour sous conditions de ressources suffisantes. Pour cela, il est possible de fixer un nombre limite de cycles ainsi qu'un nombre fini d'évènements périodiques par exemple. Si cette opération peut être facilement réalisée pour certaines applications, ce n'est pas toujours possible. Il est nécessaire de connaître en détails le comportement de l'application, de déterminer le début et la fin d'un cycle et ensuite de modifier l'implémentation réelle de l'application. L'un des objectifs de la vérification dynamique formelle étant de pouvoir travailler directement sur l'implémentation

réelle courante d'une application sans la modifier, cette approche ne constitue donc pas une piste à privilégier.

Une autre approche est de définir une profondeur maximale d'exploration. Tout comme l'approche précédente, le calcul de cette profondeur est établi d'après le comportement de l'application. Elle doit notamment garantir qu'au moins un cycle complet a été soumis à la vérification pour commencer à prétendre une vérification exhaustive. Si ce calcul peut être facilement réalisé sur de petites applications au comportement simple, ceci est généralement très complexe face à des applications distribuées réelles classiques. Cette approche limite donc le type d'applications pouvant être vérifiées grâce à celle-ci.

Notre approche consiste ainsi à travailler sur l'application réelle, sans modification de son comportement, en préservant son exécution infinie le cas échéant. Grâce à la détection d'états systèmes identiques, nous pouvons détecter automatiquement les cycles au sein de ce type d'applications et donc stopper l'exploration d'un chemin courant en cas de cycles. Une vérification exhaustive est alors garantie même en cas d'exécution infinie.

Pour évaluer cette approche, nous avons effectué une exploration exhaustive d'une application MPI dont l'exécution est infinie et qui implémente l'algorithme d'exclusion mutuelle centralisée. L'exclusion mutuelle est une primitive de synchronisation pour éviter que des ressources partagées d'un système ne soient utilisées en même temps. On parle de section critique pour la partie de l'exécution durant laquelle des ressources communes sont manipulées. Le principe de cet algorithme repose sur deux types de processus : un coordinateur et des clients. Le coordinateur est en charge de la gestion de la section critique. Il reçoit des requêtes des processus clients et distribue infiniment souvent la section critique à l'un d'entre eux selon sa disponibilité et l'ordre des requêtes. Ainsi chaque client demande infiniment souvent la section critique au coordinateur, attend celle-ci, l'obtient puis la rend au coordinateur et la demande à nouveau. Dans cette exemple, un cycle est donc constitué des évènements suivants : le client envoie au coordinateur une requête de section critique, le coordinateur reçoit cette requête, il envoie la section au client dès que celle-ci est disponible, le client utilise la section puis la renvoie au coordinateur, qui la distribue alors au client dont la requête suit juste après. Enfin le client effectue une nouvelle requête auprès du coordinateur.

Les résultats de cette exploration sont présentés dans la table 3.2. Tout comme pour l'exploration des tests MPICH, nous présentons dans ce tableau une exploration avec et sans l'approche de compression mémoire. Grâce à cette dernière, nous avons pu réaliser une exploration exhaustive de cette algorithme mettant en œuvre 4 processus. Dans le cas de 5 processus, nous avons stoppé l'expérience après 6 jours d'exploration et une consommation mémoire dépassant les 120˜Go.

3 Terminaison de programmes

Pour la dernière expérience de cette partie, nous nous sommes intéressés au problème de la terminaison de programme. Ce problème est indécidable dans le cas général mais il est possible, dans certains cas, de vérifier de façon formelle que l'exécution d'un programme ne se terminera pas en détectant des cycles de non-progression. Un cycle de non-progression signifie que l'exécution d'un système avance dans le temps mais revient toujours sur un état précédemment rencontré. Elle ne progresse donc plus en terme d'exploration de nouveaux états.

Pour cela, nous avons extrait du *benchmark* de la compétition de vérification SV-COMP [12]

12. `https://svn.sosy-lab.org/software/sv-benchmarks/tags/`

# P	Avec réduction			Avec réduction et compression mémoire		
	# États	Temps	Mémoire	# États	Temps	Mémoire
2	75	3 s	0.47 Go	75	3 s	0.35 Go
3	8 216	3 min 55 s	11.33 Go	8 216	4 min 05 s	1.05 Go
4	-	-	> 200 Go	480 997	18 h 02 min	48.26 Go
5	-	-	-	> 10 millions	> 6 j	> 120 Go

Table 3.2 – Exploration exhaustive de l'algorithme d'exclusion mutuelle centralisée MPI en version infinie.

plusieurs exemples d'applications dont la non-terminaison est connue, de par la présence de cycles de non-progression. Pour cette compétition, la vérification est réalisée normalement à partir d'une analyse statique du code source. Nous proposons donc de réaliser cette même vérification mais dynamiquement.

Les codes sélectionnés parmi ces *benchmark* ne sont initialement pas destinés à être exécutés sur des systèmes distribués, aucune notion de communication n'étant intégrée. Nous les avons donc implémentés avec le standard MPI en mettant en œuvre à chaque fois deux processus, tel que le premier processus exécute le code correspondant à l'application initiale et informe un second processus de l'avancement de son calcul. Bien que non associés aux systèmes distribués initialement, ces exemples constituent une bonne base pour l'évaluation des outils de vérification.

Le premier exemple issu de la compétition de vérification sur lequel nous avons évalué notre outil met en œuvre une boucle infinie au sein de laquelle la valeur 2 est affectée à une variable `x`. L'implémentation correspondante est présentée en figure 3.1.

```
1  int x;
2
3  int main(int argc, char **argv) {
4    int buff, size, rank;
5    MPI_Status status;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   if (rank == 0) {
12     while (1) {
13       MPI_Recv(&buff, 1, MPI_INT, ANY_SOURCE, ANY_TAG, MPI_COMM_WORLD, &status);
14     }
15   } else {
16     while (1) {
17       x = 2;
18       MPI_Send(&x, 1, MPI_INT, 0, 42, MPI_COMM_WORLD);
19     }
20   }
21
22   MPI_Finalize();
23   return 0;
24 }
```

Figure 3.1 – Implémentation en MPI du code Madrid_false-termination.c

Pour ce premier exemple simple, un cycle de non-progression est détecté dès le second message envoyé par le second processus.

Un second exemple de code dans sa version originale est présenté en figure 3.2 et la version correspondante en utilisant le standard MPI en figure 3.3.

```
1  extern int __VERIFIER_nondet_int();
2
3  int main () {
4    int x = __VERIFIER_nondet_int();
5    int y = __VERIFIER_nondet_int();
6    while (1) {
7      int old_x = x;
8      x = -y;
9      y = old_x;
10   }
11   return 0;
12 }
```

Figure 3.2 – Code Rotation180_false-termination.c issu du *benchmark* termination-crafted de l'édition 2015.

```
1  int x = random_int();
2  int y = random_int();
3
4  int main(int argc, char **argv) {
5    int buff, size, rank;
6    MPI_Status status;
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &size);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12   if (rank == 0) {
13     while (1) {
14       MPI_Recv(&buff, 1, MPI_INT, ANY_SOURCE, ANY_TAG, MPI_COMM_WORLD, &status);
15     }
16   } else {
17     while (1) {
18       int old_x = x;
19       x = -y;
20       y = old_x;
21       MPI_Send(&x, 1, MPI_INT, 0, 42, MPI_COMM_WORLD);
22     }
23   }
24
25   MPI_Finalize();
26   return 0;
27 }
```

Figure 3.3 – Implémentation en MPI du code Rotation180_false-termination.c

Dans cet exemple, `x` et `y` sont modifiés de telle manière à effectuer une rotation de 90 degrés à chaque fois. Si l'on déroule cet algorithme, à partir des valeurs initiales `x=5` et `y=8` par exemple, nous obtenons les valeurs suivantes après chaque boucle `while` traitée :

— $S_0 = \{x = 5, y = 8, old\_x = ND\}$;
— $S_1 = \{x = -8, y = 5, old\_x = 5\}$;
— $S_2 = \{x = -5, y = -8, old\_x = -8\}$;

— $S_3 = \{x = 8, y = -5, old\_x = -5\}$;
— $S_4 = \{x = 5, y = 8, old\_x = 8\}$;
— $S_5 = \{x = -8, y = 5, old\_x = 5\}$;
— ...

Ainsi à partir de S_5, nous entrons dans un cycle de non-progression, avec S_5 identique à S_1. Le programme ne terminera donc jamais même si les valeurs des variables changent à chaque étape.

À partir de l'implémentation MPI de ce code, nous pouvons détecter ce cycle de non-progression grâce à la détection de ces états identiques. Le résultat est présenté dans la figure 3.4. Pour aider la compréhension, nous avons intégré les valeurs de `x, y` et `old_x` en commentaires dans la trace à chaque étape de l'exploration.

```
x = 5, y = 8
Non-progressive cycle : state 3 -> state 15
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>> NON-PROGRESSIVE CYCLE DETECTED >>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Execution trace:
State 1 : [(0)] iRecv(dst=(0))       /* x=5,   y=8,   old_x=undefined */
State 2 : [(1)] iSend(src=(1))       /* x=-8,  y=5,   old_x=5 */
State 3 : [(0)] Wait((1) -> (0))     /* x=-8,  y=5,   old_x=5 */
State 4 : [(0)] iRecv(dst=(0)))      /* x=-8,  y=5,   old_x=5 */
State 5 : [(1)] iSend(src=(1)))      /* x=-5,  y=-8,  old_x=-8 */
State 6 : [(0)] Wait((1) -> (0))     /* x=-5,  y=-8,  old_x=-8 */
State 7 : [(0)] iRecv(dst=(0)))      /* x=-5,  y=-8,  old_x=-8 */
State 8 : [(1)] iSend(src=(1)))      /* x=8,   y=-5,  old_x=5 */
State 9 : [(0)] Wait((1) -> (0))     /* x=8,   y=-5,  old_x=5 */
State 10 : [(0)] iRecv(dst=(0)))     /* x=8,   y=-5,  old_x=5 */
State 11 : [(1)] iSend(src=(1)))     /* x=5,   y=8,   old_x=8 */
State 12 : [(0)] Wait((1) -> (0))    /* x=5,   y=8,   old_x=8 */
State 13 : [(0)] iRecv(dst=(0)))     /* x=5,   y=8,   old_x=8 */
State 14 : [(1)] iSend(src=(1)))     /* x=-8,  y=5,   old_x=5 */
>> Expanded states = 15
>> Visited states = 14
>> Executed transitions = 14
```

Figure 3.4 – Résultat de la détection de cycle de non-progression sur le code Rotation180_false-termination.c

Pour ces deux exemples, malgré l'affectation de valeurs initiales aléatoires pour les variables `x` et `y` mises en jeu, un cycle de non-progression existe quelles que soient ces valeurs. C'est ainsi que nous détectons immédiatement un cycle sur le premier chemin d'exécution exploré, d'où une vérification rapide. Dans la version actuelle de notre outil, l'exploration est stoppée dès qu'un cycle est détecté mais il est facilement envisageable de poursuivre l'exploration afin de comptabiliser le nombre de chemins d'exécution contenant un cycle de non-progression.

Nous avons ensuite travaillé sur un exemple plus complexe, pour lequel il n'existe pas un cycle de non-progression sur tous les chemins mais uniquement dans certains cas. Cet exemple, présenté dans la figure 3.5, correspond au code `NonTerminationSimple5_falsetermination.c` du *benchmark* de la compétition.

```
1  int x = 20;
2
3  int main(int argc, char **argv) {
4
5    int buff = 1, size, rank;
6    MPI_Status status;
7
8    MPI_Init(&argc, &argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &size);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12   if (rank==0) {
13     while (buff >= 0) {
14       MPI_Recv(&buff, 1, MPI_INT, ANY_SOURCE, ANY_TAG, MPI_COMM_WORLD, &status);
15     }
16   }else{
17     while (x >= 0) {
18       if (MC_random(0,1) == 0)
19         x -= 1;
20       else
21         x += 1;
22       printf("x=%d\n", x);
23       MPI_Send(&x, 1, MPI_INT, 0, 42, MPI_COMM_WORLD);
24     }
25   }
26
27   MPI_Finalize();
28   return 0;
29 }
```

Figure 3.5 – Implémentation en MPI du code NonTerminationSimple5_false-termination.c

À noter au sein de celui-ci la présence de la primitive `MC_random`. Celle-ci nous permet de générer tous les chemins possibles lorsqu'une valeur aléatoire est spécifiée dans le code et ainsi augmenter l'exhaustivité de la vérification en explorant les différentes exécutions pour les différentes valeurs. La particularité de ce code est qu'il met en œuvre les deux causes de non-terminaison d'un programme : soit le système se trouve sur un chemin sur lequel `MC_random` retourne toujours 1 et `x` est incrémenté indéfiniment, soit `MC_random` alterne entre 0 et 1 faisant alors toujours revenir le système sur la même valeur de `x`. Il existera alors des chemins d'exécution finis (si `MC_random` retourne toujours 0, `x` est décrémenté jusqu'à la condition d'arrêt et l'exécution s'arrête) tandis que ceux contenant des cycles de non-progression seront infinis.

Un cycle de non-progression a été détecté en moins de 2 secondes, dès le second chemin exploré. Les différentes valeurs affichées de `x` sont : 19, 18, 17, 16, 15, ..., 2, 1, 0, -1, et 1. Le premier chemin correspond au cas où pour chaque appel à `MC_random`, la valeur retournée est 0, donc `x` est décrémenté jusqu'à ne plus satisfaire la condition de boucle et l'exploration se termine après avoir affiché `x=-1`. Dès lors, un backtracking est réalisé sur le dernier état pour lequel une autre exécution était possible, il s'agit ici du dernier `MC_random` exécuté. Nous avons donc testé le cas où ce dernier retourne 1 au lieu de 0. C'est alors que la valeur de x qui était à 0 à cet instant est passée à 1 ce qui correspond à une valeur connue de `x` sur le même chemin d'exécution. Il y a donc présence d'un cycle. À noter que pour les chemins d'exécution pour lesquels `x` est toujours incrémenté, menant alors à l'autre cas d'exécution infinie, notre outil n'étant pas capable de distinguer ce cas de figure, l'exploration sera automatiquement stoppée lorsque la profondeur maximale sera atteinte. Celle-ci est actuellement de 1000 par défaut mais peut être configurée dans la ligne de commande de l'exécution de l'application.

4 Conclusion

Nous avons présenté dans ce chapitre une approche permettant l'analyse sémantique dynamique d'un état système par introspection mémoire. L'objectif principal de ces travaux était de permettre la détection d'états systèmes identiques. De par la nature et la dynamicité des informations qui caractérisent un état système, nous avons vu qu'une analyse syntaxique n'est pas adaptée à cela, car trop stricte dans la comparaison de ces données. Nous avons identifié au cours de nos recherches plusieurs verrous à l'origine de résultats de comparaison négatifs mais incorrects. Pour chacun d'entre eux, nous proposons des solutions indépendantes basées sur des outils de *debug* tels que DWARF et libunwind, qui permettent dans leur application globale de reconstruire sémantiquement un état système et ainsi de réaliser une comparaison plus fine. Une synthèse des verrous ainsi que des solutions mises en œuvre pour y répondre est présentée dans la table 3.3.

Verrou	**Solution pour le tas**	**Solution pour la pile**
Overprovisioning	`memset` 0	`memset` 0 par modification du code binaire
Octets de remplissage	`memset` 0	`memset` 0
Différences syntaxiques	Heuristique de comparaison sémantique	N/A (Accès séquentiel)
Variables non-initialisées	`memset` 0	`memset` 0 par modification du code binaire
Différences non-pertinentes	Ignorance de zones explicites	DWARF + libunwind + ignorance

Table 3.3 – Synthèse des verrous et solutions pour la détection d'états sémantiquement identiques.

Cette analyse sémantique est réalisée de façon entièrement automatique sans intervention de l'utilisateur durant sa mise en œuvre. Seuls les *dangling pointers* et les fuites mémoires nécessitent un traitement préalable par l'utilisateur afin de les supprimer pour garantir la correction de l'analyse ensuite. Ceci est facilement réalisable avec des outils tels que Valgrind pour la gestion des fuites mémoires et la mise à `NULL` de chaque pointeur après libération pour les *dangling pointers*.

Comme détaillé dans la section 4 du chapitre 2 de cette partie, l'utilisation d'une heuristique pour l'analyse de la mémoire dynamique, représentée par le tas, implique que certains cas ne peuvent être entièrement traités. Cependant, les résultats présentés dans ce chapitre d'évaluation expérimentale démontrent l'efficacité de notre approche.

À notre connaissance, les travaux présentés dans cette partie constituent la première approche de détection d'états systèmes sémantiquement identiques sur des applications réelles dont l'implémentation ne met pas en œuvre un système de ramasse-miettes. Celle-ci permet ainsi d'étendre le type d'applications pouvant être étudiées, jusqu'à présent fortement restreint aux langages de programmation orientés objets tels que Java.

Nous présentons dans la partie suivante la vérification de propriétés temporelles exprimées dans les logiques $\text{LTL}_{\text{-}\mathbf{X}}$ et CTL dont la mise en œuvre repose en partie sur les travaux précédemment exposés.

Troisième partie

Vérification dynamique formelle de propriétés temporelles

La première étape pour la vérification de propriétés sur un système donné est d'exprimer ces propriétés. Pour cela, une première version dans un langage naturel est réalisée. Cependant, le résultat est souvent imprécis rendant toute automatisation impossible, mais également trop verbeux. Une approche consiste alors à utiliser des formalismes graphiques tels que les diagrammes de séquence de messages utilisés dans les Telecom. Plus précis et concis, ces formalismes sont faciles à apprendre et à mettre en œuvre. Cependant, il arrive souvent qu'ils ne permettent pas d'exprimer des propriétés élaborées notamment à cause de leur sémantique incomplète ou parfois peu claire. C'est ainsi que Pnueli a proposé en 1977 [Pnu77] d'utiliser la logique temporelle pour exprimer ces propriétés.

Dans le contexte des systèmes distribués, la formulation de propriétés de correction en logique temporelle permet de prendre en compte leur comportement dynamique au cours de leur exécution. Cependant, leur vérification reste complexe à mettre en œuvre, particulièrement sur des applications réelles, si bien que peu d'outils permettent cette opération. Les propriétés de sûreté pouvant être exprimées à l'aide d'assertion, il est parfaitement possible de proposer certaines vérifications sans utiliser la logique temporelle. Toutefois, cela ne peut pas être réalisé dans le cas de propriétés de vivacité. C'est ainsi que des outils tels que MaceMC proposent la vérification d'un sous-ensemble de ces propriétés exprimées en logique temporelle linéaire sous la forme d'invariants. Cependant, ce sous-ensemble reste très restreint et ne permet pas de travailler sur des propriétés plus proches du comportement des systèmes distribués.

Dans cette partie, nous présentons nos travaux autour de la vérification de certaines propriétés exprimées à l'aide de logiques temporelles : les propriétés de vivacité formulées avec la logique $\text{LTL}_{-\mathbf{X}}$ dans le chapitre 1 et le déterminisme des communications dans les applications MPI exprimées à l'aide de la logique CTL dans le chapitre 2. Nous proposons pour ces deux catégories un nouvel algorithme d'exploration, basé sur l'algorithme NDFS pour les propriétés de vivacité et *ad-hoc* pour le déterminisme des communications MPI.

Chapitre 1

Vérification de propriétés de vivacité

Dans ce chapitre, nous présentons nos travaux portant sur la vérification dynamique formelle de propriétés de vivacité. Pour cela, nous commençons par spécifier les motivations d'une telle vérification, en particulier la catégorie de propriétés dans le contexte de l'étude des systèmes distribués. Nous présentons ensuite le principe général de leur vérification, notamment leur formulation à l'aide d'une logique temporelle et l'algorithme d'exploration habituellement mis en œuvre en *model checking*. Puis, nous exposons notre approche permettant de réaliser cette vérification sur des applications réelles, et non plus des modèles, avec un nouvel algorithme d'exploration. Enfin, nous terminons par une évaluation expérimentale de cette dernière.

1 Motivations

Nous avons présenté dans le chapitre 1 de la partie I de ce document une catégorisation en deux ensembles des propriétés de correction pouvant être vérifiées : les propriétés de sûreté et les propriétés de vivacité. La principale différence entre ces deux types de propriétés est la présence ou non d'une notion de temps. En effet, les propriétés de sûreté permettent de spécifier des comportements devant être respectés quelque soit l'instant de l'exécution du système étudié tandis que les propriétés de vivacité permettent de définir des périodes de temps durant lesquelles il est autorisé que le système soit instable vis-à-vis de la propriété spécifiée. C'est ainsi que les propriétés de vivacité permettent d'exprimer des comportements plus complexes et sont donc considérées comme plus intéressantes dans l'étude et la correction des systèmes distribués.

Toutefois, leur vérification est plus complexe à mettre en œuvre. En effet, la non-satisfaction des propriétés de vivacité est évaluée sur des exécutions infinies durant lesquelles un cycle est recherché. Si une exécution contient un cycle dans lequel la propriété n'est pas satisfaite, il est alors possible d'affirmer que « le bon n'arrivera jamais » et donc que la propriété de vivacité correspondante n'est pas satisfaite sur l'ensemble des exécutions du système étudié. Si cette recherche de cycle peut être facilement implémentée sur des modèles au niveau d'abstraction plus ou moins important, cette tâche est bien plus difficile lorsque la vérification est effectuée sur l'implémentation réelle du système étudié.

C'est ainsi que peu d'outils existants permettent aujourd'hui de vérifier des propriétés de vivacité, en particulier sur des implémentations réelles de systèmes. Nous avons présenté dans le chapitre 2 de la partie I les différents outils de vérification travaillant sur des implémentations réelles. Parmi ceux se targuant de réaliser une vérification de propriétés de vivacité sur du code réel, très peu effectuent réellement cette vérification sur le code initial. Seul MaceMC vérifie des propriétés de vivacité exprimées avec la logique LTL sur du code C++. Toutefois, cette

vérification se limite aux propriétés de la forme $\Box\Diamond p$, où p est une propriété sans quantificateur. Cette restriction permet de mettre en œuvre une autre approche de vérification ne nécessitant pas la détection de cycles sur des exécutions infinies. Celle-ci repose sur une heuristique qui détermine les chemins d'intérêt, c'est-à-dire ceux pour lesquels la propriété restera non satisfaite. L'ensemble des autres outils effectue toujours une extraction de modèle dans un formalisme spécifique depuis le code réel du système étudié. La vérification est ensuite réalisée sur ce modèle à l'aide des outils classiques de *model checking* tels que SPIN, ou ils implémentent eux-mêmes la vérification, mais toujours sur le modèle obtenu. Dès lors, les techniques mises en œuvre pour cette vérification sont adaptées au langage de formalisation du système et ne peuvent donc être appliquées directement au contexte de la vérification dynamique formelle de propriétés de vivacité.

C'est dans ce contexte que nous présentons dans ce chapitre la mise en œuvre de techniques existantes de vérification de propriétés de vivacité adaptées à l'étude d'implémentations réelles, mais également un nouvel algorithme d'exploration.

2 Principe de la vérification formelle des propriétés de vivacité

2.1 Formulation à l'aide de la logique LTL

Parmi les logiques temporelles présentées dans le chapitre 1 de la partie I, la Logique Temporelle Linéaire (LTL) est couramment utilisée pour la formulation des propriétés de vivacité. Également appelée Logique Temporelle Linéaire Propositionnelle (PLTL), cette logique est un sous ensemble de la logique CTL* dans laquelle le quantificateur de chemins **E** est supprimé. De plus, le quantificateur **A** est souvent omis car présent par défaut en début de formule, cette dernière devant être satisfaite pour tous les chemins explorés. Dès lors, une formule LTL, dans le contexte d'un chemin d'exécution donné, ne peut tenir compte des exécutions alternatives en cas de non-déterminisme. La vérification se fait donc sur l'ensemble des exécutions mais indépendamment les unes des autres. C'est ainsi que les formules LTL sont dites *formules de chemin*.

Bien que considérée comme inférieure par rapport à la logique temporelle arborescente (CTL) en terme de vérification algorithmique et parfois d'expressivité, la logique LTL reste toutefois plus intuitive pour la formulation de propriétés d'exécution. La logique CTL est en effet souvent considérée comme difficile à utiliser car elle nécessite de raisonner sur un arbre et non de façon linéaire, comme cela se fait naturellement, ce qui amène alors à l'utilisation de la logique LTL. Au sein même de cette logique temporelle linéaire, des sous-ensembles de logique existent selon l'utilisation ou non des opérateurs temporels de base **U** et **X**. Sont ainsi généralement nommées LTL(**X**,**U**), la logique LTL permettant l'utilisation des deux opérateurs temporels, LTL(**U**) ou $\text{LTL}_{-\mathbf{X}}$ la logique temporelle permettant uniquement l'utilisation de l'opérateur **U** et LTL(**X**) ou $\text{LTL}_{-\mathbf{U}}$ la logique LTL permettant uniquement l'utilisation de l'opérateur **X**.

2.2 Invariance sous bégaiement

Leslie Lamport argumentait en 1983 dans [Lam83] que la spécification d'un système concurrent doit être invariante sous le bégaiement, c'est-à-dire qu'elle ne doit pas distinguer une séquence d'états d'une autre séquence obtenue en remplaçant une occurrence d'un état par plusieurs copies de celui-ci. Toute propriété satisfaite sur la séquence initiale doit donc être également satisfaite sur la séquence avec bégaiement. Ceci est également valable dans le contexte

des systèmes distribués, dès lors que la spécification haut-niveau est identique, seule l'implémentation du système diffère.

Formellement, un bégaiement est une séquence d'états étiquetés de façon identique le long d'un chemin dans une structure de Kripke.

Définition 6 (Équivalence par bégaiement). Soient les séquences d'états infinies σ et ρ, telles que :

— $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots s_n \xrightarrow{\alpha_n}$

— $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \dots r_n \xrightarrow{\beta_n}$

σ et ρ sont dites *équivalentes par bégaiement*, et notées $\sigma \sim_{st} \rho$, s'il existe deux séquences infinies d'entiers $0 = i_0 < i_1 < i_2 < \dots$ et $0 = j_0 < j_1 < j_2 < \dots$, telles que :

$$\forall k \geq 0,\ L(s_{i_k}) = L(s_{i_k+1}) = \dots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \dots = L(r_{j_{k+1}-1}),$$

où L est la fonction d'étiquetage dans la structure de Kripke. Si l'on appelle *bloc* une séquence finie d'états à l'étiquette identique, alors deux chemins sont dits équivalents par bégaiement s'ils peuvent être décomposés en blocs (possiblement de longueurs différentes), tels que les états dans le $k^{\text{ième}}$ bloc du premier chemin ont la même étiquette que les états du $k^{\text{ième}}$ bloc du second chemin.

Si cette caractéristique joue un rôle majeur dans la possibilité de raffiner les spécifications et de raisonner de façon modulaire, une motivation plus forte est l'utilisation de techniques de réduction par ordre partiel. Celles-ci nécessitent en effet que cette contrainte soit respectée afin de garantir leur correction. Si la propriété à vérifier est sensible au bégaiement, il est nécessaire d'explorer le chemin avec bégaiement et celui sans bégaiement, aucune réduction n'est donc possible. Inversement, s'il y a invariance au bégaiement, une exploration exhaustive est garantie dès lors que pour chaque chemin non considéré, il existe un chemin équivalent par bégaiement qui est considéré. Se pose alors la problématique de déterminer si une propriété est invariante au bégaiement ou non.

Il a été montré dans [PWW98] que vérifier si une propriété LTL est invariante sous bégaiement ou non est un problème PSPACE-complet. Pour éviter un tel calcul tout en s'assurant que l'invariance sous bégaiement est respectée et ainsi pouvoir appliquer les techniques de réduction par ordre partiel, Peled et *al.* ont démontré dans [PW97] que l'utilisation de l'opérateur $\mathbf{X}$ dans la logique LTL peut mener à une violation de cette invariance. C'est ainsi qu'il est souvent interdit d'utiliser cet opérateur lors de la formulation de propriétés. Bien qu'il y ait une restriction dans la formulation, ils prouvent que cette interdiction n'influe pas sur l'expressivité car toute propriété invariante au bégaiement peut être formulée sans utiliser l'opérateur $\mathbf{X}$. C'est donc dans ce contexte que les travaux présentés dans ce chapitre se concentrent sur la logique $\text{LTL}_{-\mathbf{X}}$.

2.3 Représentation de la propriété

2.3.1 Automate de Büchi

La logique LTL est un langage ω-régulier, c'est-à-dire un langage régulier à mots infinis, qui permet donc de spécifier des séquences d'états infinies. Pour représenter une formule exprimée à travers cette logique, et ainsi définir l'ensemble des exécutions qui satisfont cette formule, Wolper *et al.* justifient dans [WVS83] l'utilisation d'un automate fini acceptant des mots infinis. C'est ainsi que l'utilisation d'automates de Büchi [Bü60] constitue la base de la vérification de

formules LTL par *model checking*.

Un automate de Büchi est un automate à états finis, potentiellement non-déterministe, qui reconnaît un langage de mots infinis. Un mot est accepté par l'automate si un état dit *acceptant* est parcouru infiniment souvent. Un état *acceptant* correspond à un état dit *final* dans un automate classique mais depuis lequel il est encore possible d'évoluer.

Définition 7 (Automate de Büchi). Un automate de Büchi déterministe (ABD) est un quintuplet $\mathcal{A}_D = (\Sigma, Q, \delta, q_0, F)$ tel que :
- Σ est un alphabet fini ;
- Q est l'ensemble fini des états ;
- $\delta : Q \times \Sigma \to Q$ est la fonction de transition entre les états ;
- $q_0 \in Q$ est l'état initial ;
- $F \subseteq Q$ est la condition d'acceptation, c'est-à-dire l'ensemble des états finaux, telle que $\mathcal{A}$ accepte un mot si un état $q \in F$ est parcouru infiniment souvent.

Exemple 7. L'automate de la figure 1.1 est un ABD tel que $\Sigma = \{a, b\}$, $Q = \{q_0, q_1\}$, q_0 est l'état initial et q_1 est un état final acceptant. Il accepte les mots qui contiennent au moins deux a. Ainsi, les mots *baba* et *aa* sont acceptés, tandis que *bab* ne l'est pas.

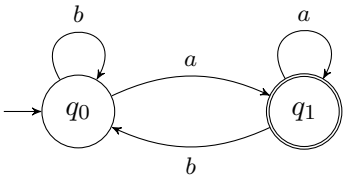

Figure 1.1 – Automate de Büchi déterministe

Dans le cas d'un automate de Büchi non-déterministe (ABND) $\mathcal{A}_{ND} = (\Sigma, Q, \Delta, I, F)$, la fonction de transition δ est remplacée par une relation de transition $\Delta : Q \times \Sigma \times Q$ qui retourne un ensemble d'états et l'état initial q_0 est remplacé par un ensemble d'états initiaux $I \subset Q$.

On note $L(\mathcal{A})$ le langage de tous les mots reconnus par l'automate de Büchi $\mathcal{A}$.

Les automates de Büchi déterministes (ABD) et non-déterministes (ABND) ne sont pas équivalents. Par exemple, il n'existe pas d'ABD qui reconnaît l'ensemble $\{a, b\}^* b^\omega$, c'est-à-dire les mots infinis sur deux lettres a et b qui ne contiennent qu'un nombre fini de lettres a, alors que cet ensemble est reconnu par un ABND à deux états présenté en figure 1.2.

C'est ainsi que les ABD sont considérés comme moins expressifs que les ABND, le non-déterminisme est donc privilégié dans la représentation des formules LTL.

2.3.2 Négation de la formule LTL$_{-\mathbf{X}}$

Une formule LTL définit l'ensemble des exécutions infinies qui satisfont une propriété donnée. Dès lors, pour vérifier une formule LTL sur un système, une approche serait de vérifier que l'ensemble des exécutions possibles du système étudié est contenu dans l'ensemble des exécutions

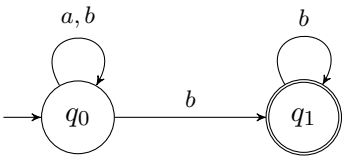

Figure 1.2 – Automate de Büchi non-déterministe qui accepte les mots infinis contenant un nombre fini de a.

acceptées. Ceci équivaut à déterminer si l'intersection entre l'ensemble des exécutions possibles du système et le complément de l'ensemble des exécutions acceptées par la formule est vide. Bien que possible, calculer ce complément est toutefois extrêmement difficile à cause des propriétés de fermeture auxquelles sont soumis les automates de Büchi [SVW85].

Une autre approche, plus simple, consiste à considérer la négation de la formule LTL. Celle-ci définit alors l'ensemble des exécutions infinies qui ne satisfont pas la propriété donnée. Il suffit, dans ce cas, de trouver une seule de ces exécutions parmi les exécutions possibles du système pour affirmer que ce dernier ne satisfait pas la propriété définie.

2.3.3 Construction de l'automate de Büchi de la négation de la formule LTL$_{-\mathbf{X}}$

Lorsque la propriété est définie, il est donc nécessaire de l'exprimer dans un premier temps en formule LTL$_{-\mathbf{X}}$. Puis, la négation de la formule est obtenue grâce à l'opérateur logique $\neg$ ajouté en début de formule. C'est à partir de cette dernière formule LTL qu'un automate de Büchi est construit.

La construction de cet automate n'est pas triviale. Bien que les formules LTL soient généralement courtes, la taille de l'automate augmente de façon exponentielle par rapport à la taille de la formule. Plus l'automate sera grand, plus le coût de la vérification sera élevé en ressources. Sachant que les ABND et les ABD ne sont pas équivalents, il n'est pas possible d'utiliser les algorithmes de minimisation existants. Déterminer la taille optimale d'un automate de Büchi étant un problème PSPACE [EH00], le processus de génération depuis la formule LTL est crucial pour limiter la taille de l'automate résultant.

2.3.3.1 Principe

La conversion d'une formule LTL en un automate de Büchi se déroule généralement en trois phases [SB00] : (1) Réécriture de la formule ; (2) Conversion de la formule en un automate de Büchi généralisé ; (3) Dé-généralisation de l'automate de Büchi généralisé.

La première phase consiste à réécrire la formule en forme normale négative. Une formule logique est dans cette forme lorsque l'opérateur de négation $\neg$ est uniquement appliqué aux variables et les seuls opérateurs booléens autorisés sont la conjonction ($AND, \wedge$) et la disjonction ($OR, \vee$). Pour obtenir cette forme, les règles de réécritures de base suivantes sont appliquées :

- $\neg\neg\phi = \phi$
- $\neg\mathbf{G}\phi = \mathbf{F}\neg\phi$
- $\neg\mathbf{F}\phi = \mathbf{G}\neg\phi$
- $\neg(\phi\mathbf{U}\psi) = (\neg\phi)\mathbf{R}(\neg\psi)$
- $\neg(\phi\mathbf{R}\psi) = (\neg\phi)\mathbf{U}(\neg\psi)$

Des règles supplémentaires permettent également de réduire la taille de la formule et donc la taille de l'automate correspondant. Cependant, celles-ci ne garantissent pas que l'automate obtenu sera le plus petit possible.

La deuxième étape de la conversion représente le cœur de la transformation. Il s'agit de générer un automate de Büchi généralisé (ABG) à partir de la forme normale négative de la formule. La différence avec un automate de Büchi réside dans la condition d'acceptation qui devient un ensemble d'ensembles d'états acceptants. Dès lors, une exécution est acceptée par un automate de Büchi généralisé si au moins un état de chaque ensemble d'états acceptants de l'ensemble des ensembles des états acceptants est parcouru infiniment souvent.

Définition 8 (Automate de Büchi généralisé). Un automate de Büchi généralisé (ABG) est un quintuplet $\mathcal{A}_G = (\Sigma, Q, \Delta, I, \{F_1, \ldots, F_n\})$ tel que :
- Σ est un alphabet fini ;
- Q est l'ensemble fini des états ;
- $\Delta : Q \times \Sigma \to 2^Q$ est la relation de transition entre les états ;
- $I \subset Q$ est l'ensemble des états initiaux ;
- $\{F_1, \ldots, F_n\}$ est la condition d'acceptation, c'est-à-dire l'ensemble des ensembles des états finaux, tel que $1 \leq i \leq n$ et $F_i \subseteq Q$.

Plusieurs algorithmes permettent cette conversion d'une formule LTL vers un automate de Büchi généralisé [WVS83, GO01, GPVW95]. Bien qu'ils diffèrent dans leur stratégie de construction, ils reposent tous sur un même principe : chaque état dans l'ABG représente un ensemble des formules LTL qui devraient être satisfaites par le mot restant soumis en entrée après parcours de l'état correspondant durant une exécution.

La dernière phase de la conversion consiste à dé-généraliser l'automate de Büchi généralisé préalablement construit. Pour cela, l'approche est, dans un premier temps, de dupliquer l'ABG autant de fois qu'il y a d'ensembles d'états acceptants. Puis, les transitions sortantes de chaque état acceptant sont redirigées vers la copie suivante. Enfin, les états non accessibles sont supprimés. Il reste alors un seul automate correspondant à l'automate de Büchi final. C'est à partir de cet automate que la vérification est ensuite réalisée.

Plusieurs outils permettent de construire automatiquement un automate de Büchi à partir d'une formule LTL donnée.

2.3.3.2 Outils de conversion automatique LTL vers Automate de Büchi

SPOT [DLP04] est une bibliothèque de *model checking* orientée objet qui offre un ensemble d'outils pouvant être lié à un autre outil de vérification. Il repose sur quatre étapes successives : génération de l'espace d'états à partir d'une description du système étudié, conversion de la formule LTL en ω-automate, calcul du produit synchronisé des deux objets, vérification du produit cartésien. Parmi les fonctionnalités disponibles en ligne indépendamment du reste de l'outil, SPOT offre un traducteur LTL vers un automate de Büchi dé-généralisé [13]. Afin d'obtenir un automate de Büchi optimal, il propose d'appliquer des règles de simplification à la formule. Il met également en œuvre plusieurs algorithmes de traduction issus de la littérature [BBDL+13,

13. `http://spot.lip6.fr/ltl2tgba.html/`

Cou99, T+06]. Une description de l'automate de Büchi produit est fournie en sortie, au format PROMELA (*PROcess* or *PROtocol MEta LAnguage*) *neverclaim*, initialement développé par SPIN.

LTL2BA [GO01] est un logiciel issu des travaux de thèse de Denis Oddoux, poursuivis par Paul Gastin. Accessible également en ligne [14] ou en téléchargement, il permet de convertir une formule LTL en un automate de Büchi. Tout comme SPOT, une description en PROMELA de l'automate de Büchi correspondant est fournie en sortie de la conversion. LTL3BA[BKŘS12], basé sur LTL2BA, offre les même fonctionnalités mais avec de meilleures performances avec notamment une approche de déterminisation.

2.4 *Emptiness Check*

Une fois la propriété construite, un algorithme d'exploration est mis en œuvre. Celui-ci détermine alors si la propriété spécifiée est satisfaite sur l'ensemble des exécutions. L'exploration se fait sur le produit cartésien du système (un modèle de celui-ci ou son implémentation réelle) et de l'automate de Büchi. Dès lors, un état correspond à une paire d'états plus précisément, composée de l'état courant du système et de l'état courant de l'automate. En construisant un automate de Büchi à partir de la négation de la formule correspondant à la propriété spécifiée, l'objectif est de déterminer s'il existe une exécution du système qui est acceptée par cet automate. Si oui, cela signifie que la négation de la propriété est satisfaite par le système, et par conséquent, que la propriété ne l'est pas. Cette exploration est appelée dans la littérature *Emptiness Check*, elle vérifie que l'ensemble des exécutions du système qui sont reconnues par l'automate de Büchi est vide.

L'algorithme *Nested Depth First Search* correspondant est présenté en algorithme 1. Proposé par Courcoubetis *et al.* dans [CVWY93] puis optimisé par Holzmann *et al.* dans [HPY96], il repose sur un double parcours en profondeur.

Un premier parcours effectue une simple exploration en profondeur depuis l'état initial du système (ligne 4), en ajoutant chaque nouvel état en cours d'exploration à la pile d'exploration (ligne 7). Celle-ci représente le chemin d'exécution en cours d'exploration et donc le contre-exemple en cas de violation de propriété. Si au cours de ce parcours un état acceptant est rencontré (ligne 11), c'est-à-dire une paire d'états dont l'état de l'automate est acceptant, celui-ci est enregistré comme tel (ligne 12) et une recherche de cycle est activée (ligne 13).

Cette recherche correspond au second parcours en profondeur. Elle est déclenchée uniquement lorsque le premier parcours en profondeur est terminé et consiste donc à revenir sur le dernier état acceptant rencontré. C'est à partir de cet état que la recherche en profondeur d'un cycle acceptant est réalisée. Si le même état est rencontré durant ce second parcours (ligne 19), cela signifie qu'un cycle acceptant a été détecté et donc qu'il y a violation de propriété. Un contre-exemple peut alors être rapporté.

Le même algorithme avec la mise en œuvre de la réduction sur les états visités est présenté dans l'algorithme 2. Afin de distinguer les états explorés lors du premier parcours en profondeur de ceux du second parcours dans l'ensemble des états visités sauvegardés, un booléen est associé à chaque état lors de sa sauvegarde. Celui-ci permet d'indiquer si l'état est exploré sans recherche de cycle (donc à 0 dans le premier parcours en profondeur) ou avec recherche de cycle (donc à 1 dans le second parcours en profondeur). Ainsi, si un état est identique à un autre état lors du

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

Algorithme 1 Algorithme *Nested Depth First Search*

```
 1: procedure DFS_INIT
 2:     ExplorationStack ← EmptyStack() ;
 3:     seed ← nil ;
 4:     DFS(initial_state) ;
 5: end procedure

 6: procedure DFS(s)
 7:     add s to ExplorationStack ;
 8:     foreach successor t of s do
 9:         DFS(t) ;
10:     end for
11:     if accepting(s) then
12:         seed ← s ;
13:         NDFS(s) ;
14:     end if
15:     delete s from ExplorationStack ;
16: end procedure

17: procedure NDFS(s)
18:     foreach successor t of s do
19:         if t = seed then
20:             « Report cycle » ;
21:         else
22:             NDFS(t) ;
23:         end if
24:     end for
25: end procedure
```

second parcours mais que ce dernier n'a pas été évalué pour la recherche de cycle, l'exploration ne sera pas stoppée.

3 Vérification dynamique formelle de propriétés de vivacité

La vérification dynamique formelle de propriétés de vivacité sur des implémentations réelles repose sur le même principe que la vérification sur des modèles. Nous présentons dans cette section les techniques que nous avons adaptées pour permettre cette vérification et proposons également un nouvel algorithme d'exploration mettant en œuvre un unique parcours en profondeur et permettant la recherche de plusieurs cycles acceptants en même temps.

3.1 Description et construction en mémoire de l'automate de Büchi

Nous avons vu dans la section 2.3 de ce chapitre qu'il existe de nombreux outils permettant de transformer une formule LTL en un automate de Büchi. Ces derniers étant accessibles en ligne gratuitement, leur correction et leur efficacité ayant été démontrées dans la littérature, nous avons fait le choix de ne pas créer notre propre outil de construction d'automate de Büchi.

Algorithme 2 Algorithme *Nested Depth First Search* avec réduction sur les états visités

```
1: procedure DFS_INIT
2:     ExplorationStack ← EmptyStack() ;
3:     VisitedStates ← EmptySet() ;
4:     seed ← nil ;
5:     DFS(initial_state) ;
6: end procedure

7: procedure DFS(s)
8:     add {s, 0} to VisitedStates ;        /* 0 = état exploré sans recherche de cycle */
9:     add s to ExplorationStack ;
10:    foreach successor t of s do
11:        if {t, 0} not in VisitedStates then
12:            DFS(t) ;
13:        end if
14:    end for
15:    if accepting(s) then
16:        seed ← s ;
17:        NDFS(s) ;
18:    end if
19:    delete s from ExplorationStack ;
20: end procedure

21: procedure NDFS(s)
22:    add {s, 1} to VisitedStates ;
23:    foreach successor t of s do
24:        if {t, 1} not in VisitedStates then      /* 1 = état exploré pendant recherche de cycle */
25:            NDFS(t) ;
26:        else if t = seed then
27:            ≪ Report cycle ≫ ;
28:        end if
29:    end for
30: end procedure
```

Nous demandons ainsi simplement à l'utilisateur de nous fournir une description en PROMELA de l'automate de Büchi généré à partir de l'un des outils présentés dans la section suivante.

La construction en mémoire de l'automate de Büchi correspondant à la négation de la formule LTL$_{-\mathbf{X}}$ qui sera vérifiée s'effectue donc à partir de sa description en PROMELA obtenue avec l'un des outils présentés précédemment. Il s'agit d'un simple fichier au format texte qui est analysé afin d'extraire les informations suivantes : le nombre d'états, les états initiaux, les états finaux, l'alphabet et les transitions entre les états. La figure 1.3 présente la description en PROMELA et la représentation de l'automate de Büchi correspondant à la formule LTL$_{-\mathbf{X}}$ $\neg(\Box(a \rightarrow \Diamond b))$.

Dans cette description, les états sont définis séparément (`T0_init` et `accept_S2`). Les états initiaux sont identifiés par le suffixe `_init` et les états acceptants par le préfixe `accept_`. Au sein de ces états, les transitions sortantes sont déclarées sous la forme :

condition de transition → `goto` *état destination*

```
never {
  T0_init:
    if
    :: ((!(b)) && (a)) -> goto accept_S2
    :: ((true)) -> goto T0_init
    fi;
  accept_S2:
    if
    :: ((!(b))) -> goto accept_S2
    fi;
}
```

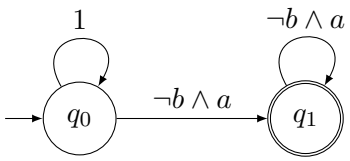

Figure 1.3 – Description en PROMELA et représentation de l'automate de Büchi correspondant à la formule $\neg(\Box(a \rightarrow \Diamond b))$.

Une transition avec la condition `true` signifie que celle-ci est toujours franchissable quelle que soit la valeur des variables de propositions.

C'est à partir de ces règles de grammaire que nous analysons la description afin de construire en mémoire l'automate de Büchi correspondant. Lors de l'analyse des conditions de transition, l'alphabet de l'automate est automatiquement extrait. Celui-ci correspond aux variables de propositions de la formule LTL initiale.

3.2 Nouvel algorithme d'exploration

Pour la vérification dynamique formelle de propriétés de vivacité exprimées à l'aide de la logique LTL$_{-\mathbf{X}}$, nous proposons un nouvel algorithme permettant de détecter au moins un cycle acceptant en un seul parcours en profondeur. Celui-ci est basé sur une unique exploration en profondeur et non deux comme dans l'algorithme original présenté dans la section 2 de ce chapitre. Il permet de rechercher plusieurs cycles acceptants différents en même temps pour un même chemin d'exécution. Cet algorithme est présenté en deux versions majeures distinctes : une récursive et une non-récursive. De plus, pour chaque version, la mise en œuvre de la réduction sur les états visités est également proposée.

3.2.1 Version récursive

Nous présentons dans un premier temps en algorithme 3 une version récursive sans réduction sur les états visités de l'exploration en profondeur avec recherche de plusieurs cycles acceptants.

Dans l'algorithme 3, nous sauvegardons l'ensemble des états acceptants pour chaque chemin exploré. Dès que le premier état acceptant est rencontré, la recherche de cycle est activée à travers le booléen *search_cycle* (lignes 13 et 14). Dès lors, lorsqu'un nouvel état acceptant est rencontré en poursuivant l'exploration en profondeur, nous vérifions si celui-ci correspond à l'un des états acceptants précédemment rencontrés sur le même chemin (lignes 8 et 9). Si la recherche est positive, cela signifie qu'un cycle acceptant a été détecté (ligne 10), sinon l'état est ajouté à l'ensemble AcceptanceStates (ligne 12).

Il est ainsi possible de rechercher plusieurs cycles acceptants en même temps, en traitant plusieurs états acceptants à la fois. Dans le pire des cas, tous les états du chemin d'exploration sont des états acceptants et nécessitent donc d'être tous sauvegardés. Cependant, grâce à la

Algorithme 3 Algorithme récursif d'exploration en profondeur multi-cycles acceptants

```
1: procedure DFS_Multi_Cycles_Init
2:     AcceptanceStates ← EmptySet() ;
3:     ExplorationStack ← EmptyStack() ;
4:     add initial_state to ExplorationStack ;
5:     DFS_Multi_Cycles(initial_state, 0) ;
6: end procedure

7: procedure DFS_Multi_Cycles(s, search_cycle)
8:     if accepting(s) then
9:         if search_cycle = 1 AND s in AcceptanceStates then
10:            ≪ Report cycle ≫ ;
11:        else
12:            add s to AcceptanceStates ;
13:            if search_cycle = 0 then
14:                search_cycle ← 1 ;
15:            end if
16:        end if
17:    end if
18:    foreach successor t of s do
19:        add t to ExplorationStack ;
20:        DFS_Multi_Cycles(t, search_cycle) ;
21:    end for
22:    delete s from ExplorationStack ;
23:    if accepting(s) then
24:        delete s from AcceptanceStates ;
25:    end if
26: end procedure
```

compression mémoire présentée dans la section 3 du chapitre 2 de la partie II, ceci reste possible tout en préservant la consommation mémoire jusqu'à une certaine limite.

La mise en œuvre de la réduction sur les états visités dans la version récursive de l'algorithme est présentée dans l'algorithme 4.

Bien que sous une forme identique à première vue, l'algorithme 4 nécessite une gestion plus complexe des états visités par rapport à l'algorithme 2 présenté dans la section précédente, correspondant à l'algorithme original mis en œuvre par SPIN pour la vérification de propriétés sur des modèles.

En effet, en réalisant un unique parcours en profondeur, en gérant donc à la fois la détection de cycles acceptants mais également des états visités, la principale difficulté réside dans la préservation des potentiels cycles acceptants lors d'une éventuelle réduction suite à la détection d'un état déjà visité.

Dans le cas de l'algorithme 2, ceci est géré grâce à un unique booléen associé à chaque état lorsque ce dernier est ajouté dans l'ensemble VisitedStates. Il indique si une détection de cycle est en cours lorsque l'état est exploré. Dès lors, pour un même état, deux versions de celui-ci sont contenues dans l'ensemble VisitedStates. En terme d'implémentation, afin de ne pas sauvegarder deux représentations mémoires d'un même état avec un seul booléen qui diffère, une

Algorithme 4 Algorithme récursif d'exploration en profondeur multi-cycles acceptants avec réduction sur les états visités.

```
 1: procedure DFS_Multi_Cycles_Init
 2:     AcceptanceStates ← EmptySet() ;
 3:     VisitedStates ← EmptySet() ;
 4:     SearchCycleSet ← EmptySet() ;
 5:     ExplorationStack ← EmptyStack() ;
 6:     add initial_state to ExplorationStack ;
 7:     DFS_Multi_Cycles(initial_state, SearchCycleSet) ;
 8: end procedure

 9: procedure DFS_Multi_Cycles(s, SearchCycleSet)
10:     if accepting(s) then
11:         if get_search_cycle(s, SearchCycleSet) = 1 AND s in AcceptanceStates then
12:             ≪ Report cycle ≫ ;
13:         else
14:             add s to AcceptanceStates ;
15:             add s to SearchCycleSet ;
16:             if get_search_cycle(s, SearchCycleSet) = 0 then
17:                 set_search_cycle(s, SearchCycleSet, 1) ;
18:             end if
19:         end if
20:     end if
21:     if {s, SearchCycleSet} not in VisitedStates then
22:         add {s, SearchCycleSet} to VisitedStates ;
23:         foreach successor t of s do
24:             add t to ExplorationStack ;
25:             DFS_Multi_Cycles(t, SearchCycleSet) ;
26:         end for
27:     end if
28:     delete s from ExplorationStack ;
29:     if accepting(s) then
30:         delete s from AcceptanceStates ;
31:         delete s from SearchCycleSet ;
32:     end if
33: end procedure
```

seule représentation peut être sauvegardée, accompagnée de deux booléens. Le premier indique si l'état a été visité lors de la première exploration (sans détection de cycle) et le second si l'état a été visité lors de la seconde exploration (avec détection de cycle). Cette approche est notamment mise en œuvre dans le *model checker* SPIN. La détection d'états identiques se base donc également sur la valeur de ces deux bits. Dès lors, un seul booléen global, pour indiquer si une recherche de cycle est en cours, et une seule paire de booléens, par état exploré, sont nécessaires pour l'algorithme 2 car un seul état acceptant à la fois est traité lors de l'exploration.

L'algorithme 4 traitant plusieurs états acceptants à la fois, une paire de booléens par état acceptant doit être mise en œuvre pour chaque état exploré ainsi qu'un booléen global pour

chaque état acceptant du chemin courant.

C'est ainsi que pour l'implémentation de l'algorithme 4, tout comme SPIN, nous ne sauvegardons dans un premier temps qu'une seule représentation mémoire de chaque état visité. De plus, pour chaque état exploré, un ensemble de paires de booléens est associé. Tout comme pour l'algorithme 2, chaque paire indique si l'état donné a été à la fois exploré sans recherche de cycle et avec recherche de cycle pour chaque état acceptant du chemin courant. Sachant que l'exploration se fait à la volée, la taille de cet ensemble pour chaque état exploré varie au cours de l'exploration, ce dernier étant mis à jour à chaque nouvel état acceptant rencontré. Ceci garantit ainsi que la détection des états visités préserve toute recherche de cycle en cours en intégrant ces booléens dans la comparaison des états.

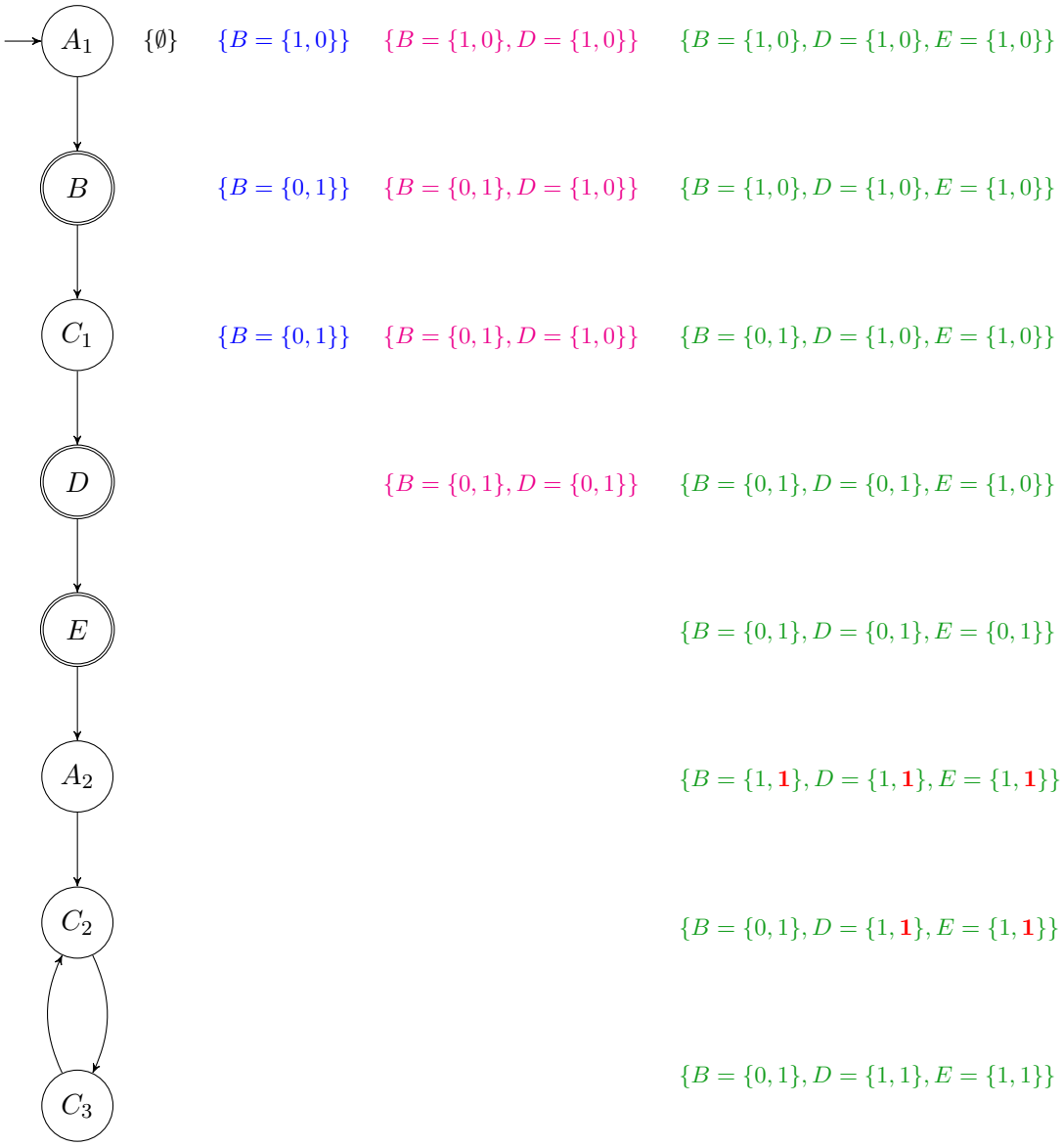

Figure 1.4 – Gestion de la détection des cycles acceptants avec réduction sur les états visités.

Un exemple d'illustration de cette implémentation est présenté en figure 1.4. Dans cet exemple, nous indiquons à chaque étape de l'exploration, l'ensemble des paires de booléens et leurs valeurs au cours de l'exploration. Un changement de couleur indique l'ajout d'un nouvel état acceptant dans chaque ensemble.

Ainsi, lorsque l'exploration commence à l'état A, cet ensemble est vide (aucun état acceptant n'a été rencontré à cet instant). Puis, en profondeur 2, l'état acceptant B est exploré. Celui-ci est donc ajouté (en bleu) à l'ensemble de paires de booléens de l'état courant mais également dans l'état précédent. Pour l'état A_1, la paire de booléens en relation avec l'état acceptant B est à {1,0} car ce premier état a été exploré alors qu'il n'y avait pas de recherche de cycle sur B. Pour l'état B, les valeurs sont inverses car une recherche de cycle est déclenchée pour ce dernier, nous considérons donc que l'état est exploré pendant une recherche de cycle sur B. À l'état suivant (C_1), seul B fait pas partie des états acceptants détectés, et C_1 est exploré pendant une recherche de cycle sur cet état acceptant, ce qui implique une paire de booléens à {0,1}. Lorsque l'exploration arrive à l'état D, tous les ensembles de paires de booléens sont mis à jour pour intégré ce nouvel état acceptant (en rose). Sachant que D n'était pas connu avant, la paire de booléen associé à celui-ci dans chaque ensemble précédent est à {1,0}. Il en est de même pour l'état acceptant suivant E (en vert). Puis l'exploration arrive sur l'état A_2 considéré comme identique avec l'état A_1. Cependant, les valeurs des booléens étant mises à jour (A est à présent exploré pendant une recherche de cycle sur B, D et E) (**en rouge**), l'exploration n'est pas stoppée car ces états sont considérés comme différents. De même pour C_2 qui n'est pas considéré comme identique avec C_1. Puis, les valeurs des paires de booléens n'étant pas modifiées pour C_3, celui-ci est considéré comme identique avec C_2 et l'exploration peut être stoppée sur ce chemin.

3.2.2 Version non-récursive

En se basant sur l'algorithme 2, nous avons donc développé dans un premier temps une version récursive de notre algorithme. Un appel récursif est effectué pour chaque nouvel état créé, c'est-à-dire, après chaque interception d'une communication de type `Send, Recv, Wait` ou `Test`. Dans le cas de la vérification de propriétés LTL, il convient également de lier le graphe d'exploration de l'application à l'automate représentant la négation de la propriété. C'est ainsi que pour un même état système, il est possible d'avoir plusieurs états de l'automate à associer, créant alors de nouveaux états supplémentaires à explorer et donc une augmentation de la récursivité.

Nous nous heurtons alors rapidement à une limite connue de la récursivité : le dépassement (ou débordement) de pile. Sans que la récursivité soit nécessairement infinie, ce problème peut survenir lorsque celle-ci est simplement trop profonde ou bien qu'il y a une allocation de variables trop grande dans la pile. C'est ainsi que nous avons constaté cette limite dans le cadre de la vérification de propriétés $\text{LTL}_{\text{-X}}$ sur de simples applications à partir de huit processus et dont la profondeur de chaque chemin d'exploration dépassait plusieurs dizaines de milliers d'états. Nous proposons alors une version non-récursive de l'algorithme présentée dans l'algorithme 5. Celui-ci intègre déjà la réduction sur les états visités.

Nous évaluons cet algorithme dans la section suivante à travers la vérification d'une propriété de vivacité exprimée à l'aide de la logique LTL sur l'implémentation réelle en MPI de l'algorithme d'exclusion mutuelle centralisée.

4 Évaluation expérimentale

Afin d'évaluer notre approche de vérification dynamique formelle de propriétés de vivacité formulées en LTL sur des applications distribuées réelles, nous avons mis en œuvre la vérification d'une propriété LTL$_{-\mathbf{X}}$ sur un exemple MPI de l'algorithme d'exclusion mutuelle centralisée dans lequel un *bug* a volontairement été introduit. L'objectif de cette expérience est de démontrer la validité de notre approche.

L'un des exemples les plus courants de propriété de vivacité exprimée avec la logique LTL est la gestion de la section critique dans l'algorithme d'exclusion mutuelle. Celle-ci est définie par : « *Chaque requête pour la section critique doit finalement être satisfaite* ». En terme de logique LTL, celle-ci correspond à : $\Box(r \rightarrow \Diamond cs)$, où r correspond à une requête et cs à l'obtention de la section critique suite à cette requête. L'automate de Büchi correspondant à la négation de cette formule est présenté en figure 1.5.

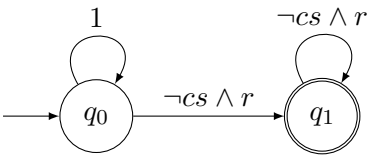

Figure 1.5 – Représentation de l'automate de Büchi correspondant à la formule $\neg(\Box(r \rightarrow \Diamond cs))$.

C'est ainsi que pour cette expérience, nous nous sommes basé sur l'implémentation de l'algorithme d'exclusion mutuelle utilisée dans les expériences de la partie II. Celle-ci a toutefois été modifiée telle que, au sein du code correspondant au coordinateur, un test sur le processus à l'origine d'une requête est effectué et le comportement diffère selon le résultat de ce test. Si la requête est issue d'un processus donné, celle-ci n'est pas satisfaite. Le processus concerné n'obtient alors jamais la section critique qu'il a demandée ce qui correspond à une violation de la propriété précédemment définie.

#P	**# États**	**Temps**	**Mémoire**	**Profondeur contre-exemple**
3	101	2,5 s	0.35 Go	94
4	287	3 s	0.37 Go	267
5	960	5,5 s	0.62 Go	890
8	68 128	18 min 40 s	5 Go	62 831
10	$>$ 200 000	$>$ 1 h	$>$ 16 Go	-

Table 1.1 – Vérification de la propriété LTL $\Box(r \rightarrow \Diamond cs)$ sur un exemple manuellement buggé d'exclusion mutuelle centralisée en MPI.

À noter qu'à chaque étape de l'exploration, lorsque plusieurs transitions sont actives et peuvent donc être exécutées, notre outil de vérification sélectionne toujours la première dans l'ordre des PID des processus à l'origine de ces transitions. Ainsi, selon si le *bug* affecte le

processus au PID le plus petit ou celui au PID le plus grand, la violation de la propriété aura lieu à des instants différents et nécessitera donc une exploration plus approfondie ou non. Les résultats présentés dans la table 1.1 sont obtenus dans le pire des cas, c'est-à-dire lorsque le *bug* touche le dernier processus dans l'ordre des PID, nécessitant alors une exploration plus approfondie pour découvrir la violation de propriété.

5 Conclusion

Nous avons présenté dans ce chapitre la mise en œuvre de la vérification dynamique formelle de propriétés de vivacité exprimées à l'aide de la logique $\text{LTL}_{-\mathbf{X}}$ sur des applications distribuées réelles. Basée sur la détection de cycles d'exécution infinis dans lesquels la propriété n'est pas satisfaite, celle-ci repose sur les travaux de détection d'états systèmes identiques présentés dans la partie précédente. Nous avons démontré la validité de notre approche à travers la vérification d'une propriété sur l'algorithme d'exclusion mutuelle centralisée.

Pour l'évaluation expérimentale de cette contribution, nous avons été confronté à la difficulté de trouver des applications réelles sur lesquelles des propriétés de vivacité peuvent être spécifiées. En effet, contrairement à la détection de *deadlock* qui ne nécessite pas de spécifier une propriété ni même de connaître le comportement de l'application, la formulation de propriétés de vivacité requiert de connaître un minimum l'application étudiée pour en extraire des propriétés pertinentes à vérifier. De plus, il subsiste une certaine barrière entre la communauté système et la communauté des méthodes formelles, que la vérification dynamique formelle tente de réduire. L'utilisation de la logique temporelle pour la formulation des propriétés à vérifier peut s'avérer complexe pour des utilisateurs peu familiers avec les méthodes formelles, si bien qu'ils ne vont pas au-delà de cette étape.

Dans le chapitre suivant, nous étendons cette vérification à une autre catégorie de propriétés, l'étude du déterminisme des communications dans les applications MPI, basée sur une formulation avec la logique CTL.

Algorithme 5 Algorithme non-récursif d'exploration en profondeur multi-cycles acceptants avec réduction sur les états visités.

```
1:  procedure DFS_Multi_Cycles_Init
2:      AcceptanceStates ← EmptySet() ;
3:      VisitedStates ← EmptySet() ;
4:      SearchCycleSet ← EmptySet() ;
5:      ExplorationStack ← EmptyStack() ;
6:      add initial_state to ExplorationStack ;
7:      DFS_Multi_Cycles(SearchCycleSet) ;
8:  end procedure

9:  procedure DFS_Multi_Cycles(SearchCycleSet)
10:     while NOT is_Empty(ExplorationStack) do
11:         s = top_Stack(ExplorationStack)
12:         if enabled_Transitions(s) > 0 then
13:             if accepting(s) then
14:                 if get_search_cycle(s, SearchCycleSet) = 1 AND s in AcceptanceStates then
15:                     ≪ Report cycle ≫ ;
16:                 else
17:                     add s to AcceptanceStates ;
18:                     add s to SearchCycleSet ;
19:                     if get_search_cycle(s, SearchCycleSet) = 0 then
20:                         set_search_cycle(s, SearchCycleSet, 1) ;
21:                     end if
22:                 end if
23:             end if
24:             if {s, SearchCycleSet} not in VisitedStates then
25:                 add {s, SearchCycleSet} to VisitedStates ;
26:                 foreach successor t of s do
27:                     add t to ExplorationStack ;
28:                 end for
29:             else
30:                 goto backtracking ;
31:             end if
32:         else
33: backtracking :
34:             while enabled_Transitions(s) = 0 do
35:                 delete s from ExplorationStack ;
36:                 if accepting(s) then
37:                     delete s from AcceptanceStates ;
38:                     delete s from SearchCycleSet ;
39:                 end if
40:                 s = top_Stack(ExplorationStack) ;
41:             end while
42:         end if
43:     end while
44: end procedure
```

Chapitre 2

Vérification du déterminisme des communications dans les applications MPI

Dans ce chapitre, nous présentons nos travaux portant sur l'étude du déterminisme des communications dans le contexte des applications MPI issues des systèmes distribués HPC. Pour cela nous commençons par exposer les motivations de cette étude, notamment son intérêt dans le développement de protocoles de tolérances aux fautes. Nous détaillons ensuite les différents niveaux de déterminisme auxquels nous nous intéressons. Puis nous présentons le mécanisme mis en œuvre pour permettre une vérification automatique de ces différents déterminismes. Nous terminons ce chapitre par une évaluation expérimentale du travail implémenté.

1 Motivations

Nous avons présenté dans le chapitre 1 de la partie I de ce document une classification des systèmes distribués. Parmi les différentes catégories se trouvent les systèmes HPC (*High Performance Computing*). Grâce à l'agrégation de plusieurs milliers de machines, des superordinateurs sont créés, permettant ainsi la résolution rapide de gros calculs. Parmi les propriétés intéressantes à étudier sur ce type de système en particulier, nous nous intéressons dans ce chapitre au déterminisme des communications dans les applications MPI. Cette étude répond avant tout à un besoin pratique dans la communauté HPC à très large échelle.

En effet, le critère premier dans la qualification des systèmes HPC est la performance du système, c'est-à-dire sa capacité à réaliser le plus de calculs possibles en un temps toujours plus réduit. Il en découle alors une course à la performance parmi les acteurs du domaine. Pour cela, l'une des approches les plus efficaces consiste à augmenter le parallélisme, en ajoutant donc toujours plus de machines et processeurs. Toutefois, la conséquence immédiate de cette opération (en plus de l'augmentation des performances de calculs bien sûr) est l'augmentation de la fréquence des pannes. Chaque machine ayant une probabilité de panne non nulle, la probabilité d'avoir au moins machine en panne à un instant donné augmente lorsque le nombre de machines augmente également. Cette probabilité devient alors très importante pour le nombre de nœuds habituellement présents dans un superordinateur. Dans le contexte des systèmes distribués, les défaillances qui peuvent découler de ces pannes sont, par exemple, la perte de messages, l'arrêt total ou temporaire de processus ou l'attaque de ces derniers conduisant à un comportement différent du protocole initialement défini.

Des protocoles de tolérance aux pannes sont alors mis en œuvre pour assurer la stabilité du système malgré leur apparition. Cette tolérance peut être pour les pannes matérielles mais également les défaillances logicielles. Le rôle de ces protocoles est de sauvegarder les informations qui caractérisent l'état du système au cours de son exécution normale, de telle manière que celui-ci peut être restauré dans un état précédent en cas de panne, pour ainsi reprendre et terminer correctement un calcul malgré la défaillance. Le développement de nombreux protocoles de tolérance aux pannes a amené à leur regroupement en différentes familles : (1) les protocoles basés sur la sauvegarde coordonnée [CL85] [KT87], (2) les protocoles avec sauvegarde non-coordonnée et enregistrement des messages [AM98] [JZ87] [BRB+09] et (3) les protocoles de sauvegarde induite par les communications [QBC00].

Toutefois, chacune de ces familles de protocoles souffre d'inconvénients les rendant inadaptés aux systèmes HPC. En effet, dans le cas des protocoles basés sur l'enregistrement des messages, une diminution des performances du système peut être constatée, en particulier si aucune défaillance n'est détectée au cours de l'exécution. Pour les protocoles avec sauvegarde coordonnée, il est nécessaire de redémarrer l'ensemble des processus même en présence d'un unique processus défaillant. Enfin, en cas de sauvegarde non-coordonnée sans enregistrement des messages, le système doit être restauré à son état initial ce qui implique de recommencer le calcul depuis le début.

C'est dans ce contexte que des recherches sur le développement de nouveaux protocoles de tolérance aux pannes adaptés aux systèmes HPC ont été initiées. Une approche pour l'implémentation de meilleurs protocoles dans le contexte des systèmes HPC est notamment d'étudier leur comportement afin d'en extraire des caractéristiques pouvant être exploitées par les protocoles. Les protocoles existants se basent en particulier sur le déterminisme éventuel des applications. Ainsi, les protocoles avec sauvegarde coordonnée ou non-coordonnée sans enregistrement des messages sont particulièrement adaptés pour les applications dont les actions des processus peuvent être non-déterministes. En l'absence de l'enregistrement des messages, aucune supposition ne peut être réalisée sur l'état de l'exécution redémarrée après la défaillance, celui-ci peut être totalement différent de celui avant la défaillance. C'est pour cela que tous les processus doivent être redémarrés même si la défaillance n'affecte que l'un d'entre eux. Si un enregistrement des messages est inclus dans le protocole, celui-ci peut alors éviter ce redémarrage complet si les processus sont déterministes par morceaux. La présence de déterminisme signifie notamment que certaines communications peuvent être recréées à partir d'une seule, il n'est donc pas nécessaire de toutes les sauvegarder ce qui améliore les performances du protocole. Dès lors, s'il est garanti que l'état d'un processus peut être reconstruit à l'identique depuis un instant donné de l'exécution, il n'est plus nécessaire de reprendre l'exécution depuis le début ni même de réaliser des sauvegardes aussi régulièrement. Pouvoir caractériser formellement le comportement d'une application en terme de communications permet alors de réduire la fréquence de ces *checkpoints* mais également d'affiner les informations sauvegardées et ainsi mettre en œuvre des protocoles plus efficaces et performants.

Cappello *et al.* ont réalisé dans [CGS10] une étude sur 27 applications réelles MPI issues de différents *benchmarks* et ont identifié à partir de celles-ci différents schémas de communication caractéristiques de différents niveaux de déterminisme. Pour réaliser cette étude, les auteurs ont travaillé à partir du code source des applications et ont extrait différents schémas fréquemment utilisés dans les applications MPI. Selon la présence ou non de ces derniers, il est alors possible d'affirmer qu'une application possède ou non un niveau de déterminisme particulier, quel que soit le reste du code. Ainsi, si une application est qualifiée de déterministe dans ses communications, au lieu d'enregistrer tous les messages dans le but de reconstruire l'état d'un processus

défaillant, il est alors possible de re-générer ces derniers simplement en ré-exécutant uniquement les messages envoyés. Le déterminisme garantissant que les mêmes communications sont traitées en ré-exécutant les messages envoyés, l'état du processus défaillant sera identique après cette restauration.

Bien que des résultats ont pu être obtenus à partir de cette étude, l'un des inconvénients majeurs de cette approche est la nécessité de travailler sur le code source de l'application et d'identifier manuellement la présence ou non de certains schémas de communications. De plus, le standard MPI est suffisamment vaste pour que l'ensemble des schémas de communication possibles ne puisse être entièrement caractérisé en terme de déterminisme. Enfin, il est difficile de garantir une identification correcte et totale du déterminisme des communications face à des codes complexes mais surtout en présence de non-déterminisme dans l'exécution (l'ensemble des exécutions possibles de l'application doit être identifié et étudié).

À notre connaissance, aucun outil ne permet actuellement de vérifier formellement ce déterminisme. En effet, bien que la vérification dynamique formelle soit offerte à travers plusieurs outils pour la vérification d'applications réelles dans divers langages, l'étude des applications MPI en particulier reste peu mise en œuvre. Nous avons présenté dans la section 2.3 du chapitre 3 de la partie I différents outils permettant la vérification d'applications MPI. Parmi eux, ISP et DAMPI sont les plus proches en terme de fonctionnalités offertes actuellement par rapport à notre outil SimGridMC. Cependant, l'approche mise en œuvre par ces derniers se limite actuellement à la vérification de *deadlocks* ou corruptions mémoires. Une étude des communications et de leur interdépendance présentée dans [SGB12] permet une réduction efficace de l'espace d'états exploré au cours de la vérification mais celle-ci ne traite pas du déterminisme des communications. Du côté des protocoles de tolérance aux pannes, en dehors des travaux de Cappello *et al.*, aucune étude n'a été réalisée. L'ensemble des travaux développés dans ce domaine de recherche suppose la présence de niveaux de déterminisme sans le vérifier de façon formelle ni même automatique.

C'est ainsi que nous proposons dans ce chapitre une approche de vérification dynamique formelle automatique du déterminisme des schémas de communication sur des applications réelles MPI.

2 Déterminismes des schémas de communication MPI

La plupart des applications MPI en HPC correspondent à un programme dans lequel il y a des communications et des phases de calculs, ces communications pouvant être réalisées pendant les phases de calculs. Nous nous intéressons ici uniquement à l'étude des communications et non de l'application entière avec ses éventuels calculs.

L'évaluation du déterminisme des communications se fait pour chaque processus indépendamment des autres. Pour cela, le sous-ensemble ordonné des actions associées à chaque processus est extrait et évalué par processus. Action signifie dans ce contexte communication. Toutes les communications réalisées par un processus sont donc interceptées et sauvegardées dans l'ordre de leur exécution. Il y a déterminisme lorsque l'ensemble des communications réalisées par chaque processus est identique sur l'ensemble des exécutions possibles de l'application, c'est-à-dire si pour tout i_p compris entre 1 et le nombre total de communications du processus $p \in P$, la $i_p{}^{\text{ème}}$ communication est identique pour toutes les exécutions . Le fait qu'il puisse exister des chemins d'exécution différents durant lesquels les processus ne sont pas exécutés dans le même ordre ne fait donc pas partie de notre étude.

Une communication réalisée par un processus peut être de deux natures différentes : soit

le processus envoie un message, soit il en reçoit un. Nous évaluons donc les actions de type `Send` et/ou `Recv`. Sachant que tout appel MPI peut être décomposé en utilisant uniquement ces actions, nous interceptons donc chacune d'entre elles au cœur de chaque fonction MPI. En plus de sa nature, une communication est caractérisée par une source, une destination et un *tag* (correspond à un identifiant) qui permet à un processus de filtrer un peu plus les messages qu'il accepte de recevoir. Sachant que pour cette étude le déterminisme des calculs n'est pas évalué, nous ne traitons pas les éventuelles données associées à chaque communication. Deux communication sont donc considérées comme identiques si elles sont de même type, possèdent la même source, la même destination et le même tag.

Dans le standard MPI, une action de type `Send` peut être réalisée à travers les deux primitives suivantes :

— `MPI_Send(void *buff, int count, MPI_Datatype datatype, int dst, int tag, MPI_Comm communicator)`
— `MPI_Isend(void *buff, int count, MPI_Datatype datatype, int dst, int tag, MPI_Comm communicator, MPI_Request *request)`

`MPI_Send` permet de créer une communication bloquante et prend en paramètre l'adresse du *buffer* source, le type des données et le nombre d'éléments présents dans le *buffer*, le *rank* du processus destination, un *tag* et le communicateur. `MPI_Isend` possède les mêmes arguments plus un argument permettant de retrouver la communication créée, celui-ci permettant de créer une communication non-bloquante.

Pour la réception, il existe également deux primitives respectivement bloquantes et non-bloquantes :

— `MPI_Recv(void *buff, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm communicator, MPI_Status *status)`
— `MPI_Irecv(void *buff, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm communicator, MPI_Request *request)`

`MPI_Recv` prend en paramètre le *buffer* de réception des données, le type et le nombre d'éléments associés à la communication, le *rank* du processus à l'origine de la communication, le *tag* du message, le communicateur et le statut de la communication. Ce dernier permet d'évaluer la bonne réception du message. Tout comme pour `MPI_Isend`, `MPI_Irecv` possède un argument supplémentaire correspondant à la communication reçue.

À partir de ces informations, nous pouvons définir quatre types de déterminisme pouvant être évalués sur des applications MPI :

— le déterminisme des communications envoyées (`Send`) ;
— le déterminisme des communications reçues (`Recv`) ;
— le déterminisme de toutes les communications (`Send` et `Recv`) ;
— le non-déterminisme de toutes les communications (par déduction des autres déterminismes).

Nous détaillons chacun de ces déterminismes dans les sections 2.1 à 2.4. Pour définir formellement chaque déterminisme, nous nous appuyons sur la définition suivante d'une application : soient $\mathcal{E}$ l'ensemble des exécutions possibles d'une application et P l'ensemble des processus qui s'exécutent au sein de cette application. Un processus $p \in P$ est défini par un ensemble d'états S_p, un état initial $s_0^P \in S_p$, un ensemble d'actions A_p et une relation de transition $\mathcal{R}_p$ entre états telle que $\mathcal{R}_p : S_p \times A_p \to S_p$. Pour chaque exécution $E \in \mathcal{E}$, on note $E|p$ le sous-ensemble d'évènements associés au processus p.

2.1 Déterminisme des communications envoyées

2.1.1 Définition

Pour ce déterminisme, nous ne nous intéressons qu'aux actions de type `Send`. Pour chaque processus, les communications envoyées sont ordonnées par ordre d'exécution dans un sous-ensemble, pour chaque exécution possible de l'application. L'objectif est ensuite d'évaluer, pour chaque processus indépendamment des autres, si chaque ensemble de chaque exécution est identique pour l'ensemble des exécutions.

Définition 9 (Send-déterminisme). Une application est dite *send*-déterministe si pour tout $p \in P$ et tout $E \in \mathcal{E}$, $E|p$ contient le même sous-ensemble ordonné d'actions de type `Send` $\in A_p$.

2.1.2 Exemple de schéma de communication *send*-déterministe

Le schéma de communication suivant est *send*-déterministe :

```
1 if(rank == 0) {
2   for(i=0; i<nb_proc - 1; i++) {
3     MPI_Irecv(hostname, ..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...);
4   }
5   MPI_WaitAll(nb_proc - 1, ...);
6 }else{
7   MPI_Send(localhost, ..., 0, 0, ...);
8 }
```

L'utilisation de communications bloquantes pour l'envoi à travers `MPI_Send` ainsi que la destination fixée à 0 en paramètre permettent de garantir le déterminisme des communications envoyées.

2.2 Déterminisme des communications reçues

2.2.1 Définition

Seules les actions de type `Recv` sont étudiées dans ce déterminisme. L'objectif est donc d'évaluer si chaque processus reçoit tous ses messages dans le même ordre pour toutes les exécutions possibles de l'application.

Définition 10 (Recv-déterminisme). Une application est dite *recv*-déterministe si pour tout $p \in P$ et tout $E \in \mathcal{E}$, $E|p$ contient le même sous-ensemble ordonné d'actions de type `Recv` $\in A_p$.

2.2.2 Exemple de schéma de communication *recv*-déterministe

L'exemple de schéma de communication *send*-déterministe présenté précédemment n'est pas *recv*-déterministe. En effet, la mise en œuvre de communications non-bloquantes ainsi que l'utilisation de `MPI_WaitAll` impliquent des ordres de réceptions potentiellement différents selon les exécutions.

Pour obtenir des communications *recv*-déterministe à partir de cet exemple, il est possible de remplacer les réceptions non bloquantes par des réceptions bloquantes à travers `MPI_Recv`, en précisant en paramètre la source des communications attendues :

```
1 if(rank == 0) {
2   for(i=0; i<nb; i++) {
3     MPI_Recv(hostname, ..., i, MPI_ANY_TAG, ...);
4   }
5 }else{
6   MPI_Send(localhost, ..., 0, 0, ...);
7 }
```

Ainsi, l'ordre des réceptions est défini d'après la boucle `for`, en l'occurrence par ordre croissant de rank des processus sources.

2.3 Déterminisme de toutes les communications

2.3.1 Définition

Si une application est qualifiée à la fois de *send*-déterministe et de *recv*-déterministe, celle-ci est alors *comm*-déterministe. Ainsi, pour chaque processus, les envois et les réceptions sont identiques et dans le même ordre pour l'ensemble des exécutions de l'application. Si l'une des actions d'un processus à un instant donné est différente entre deux exécutions, quel que soit son type, l'application n'est pas *comm*-déterministe.

Définition 11 (Comm-déterminisme). Une application est dite *comm*-déterministe, si elle est à la fois *send*-déterministe et *recv*-déterministe, c'est-à-dire si pour tout $p \in P$ et tout $E \in \mathcal{E}$, $E|p$ contient le même ensemble ordonné d'actions $\in A_p$.

2.3.2 Exemple de schéma de communication *comm*-déterministe

L'exemple de schéma de communication *recv*-déterministe présenté précédemment étant également *send*-déterministe, celui-ci est par conséquent *comm*-déterministe.

2.4 Non-déterminisme de toutes les communications

2.4.1 Définition

Par déduction des précédents déterminismes, nous établissons une dernière catégorie de déterminisme : le non-déterminisme total des communications. Dans ce cas, ni les envois, ni les réceptions ne sont identiques à travers l'ensemble des exécutions de l'application. Bien que plus rare parmi les applications MPI en HPC, ce non-déterminisme peut toutefois apparaître dès lors que des paramètres tels que `MPI_ANY_SOURCE` et `MPI_ANY_TAG` sont utilisés tel que nous le présentons dans l'exemple qui suit.

2.4.2 Exemple de schéma de communication non-déterministe

```
1 if(rank == root) {
2   for(i=1; i<nb_procs; i++) {
3     MPI_Recv(x, ..., MPI_ANY_SOURCE, ..., &status);
4     MPI_Send(y, ..., status.MPI_Source);
5   }
6 }else{
7   MPI_Send(x, ..., root, ...);
8   MPI_Recv(y, ..., root, ...);
9 }
```

Dans cet exemple, le processus correspondant à `root` attend une communication depuis n'importe quel processus grâce au paramètre `MPI_ANY_SOURCE` dans `MPI_Recv`. Ceci implique donc dans un premier temps que l'application ne peut être *recv*-déterministe, l'ordre des envois pouvant différer entre les exécutions. Ensuite, ce même processus envoie à son tour un message, mais la destination de celui-ci est déterminée d'après la source du message précédemment reçu (paramètre `status.MPI_Source` dans `MPI_Send`). Ainsi, cette source pouvant changer selon les exécutions pour un instant donné, les envois de ce processus seront donc également différents selon les exécutions. Dès lors, l'application n'est ni *recv*-déterministe, ni *send*-déterministe, elle est donc non-déterministe pour l'ensemble des communications.

Cette combinaison de réceptions depuis n'importe quelle source et d'envois vers la source d'un message précédemment reçu constitue l'exemple le plus classique en terme de non-déterminisme des communications dans les applications MPI en HPC.

Nous détaillons dans la section suivante l'approche mise en œuvre pour la vérification dynamique formelle de ces différents déterminismes.

3 Vérification dynamique formelle du déterminisme des communications MPI

Afin de vérifier formellement ces différents déterminismes sur des applications MPI réelles, nous nous appuyons naturellement sur les travaux présentés précédemment dans ce document. L'exécution réelle des applications s'appuie sur SMPI, tandis que le contrôle de l'exécution et l'exploration exhaustive des exécutions possibles sont réalisés grâce à SimGridMC.

Pour permettre l'étude du déterminisme, nous accompagnons ces mécanismes par : (1) la formulation des propriétés correspondantes avec la logique CTL et (2) l'implémentation d'un algorithme de vérification de ces propriétés. Le problème de l'explosion combinatoire étant toujours présent du fait d'une exploration exhaustive, nous proposons également (3) la mise en œuvre de la réduction sur les états visités présentée dans les parties précédentes de ce document.

3.1 Formulation avec la logique CTL

Pour évaluer le déterminisme des communications d'une application MPI, l'approche consiste à évaluer l'ensemble des exécutions possibles et à comparer les communications de chaque processus entre chacune de ces exécutions. Ainsi, s'il existe une exécution dont le schéma de communication par processus est différent par rapport aux autres exécutions, alors cette application n'est pas déterministe, selon le déterminisme étudié.

La première approche pour la formulation de cette propriété consiste à exprimer une propriété qui devra être satisfaite pour tous les chemins, en utilisant donc le quantificateur **A** disponible dans la logique LTL mais également CTL. Il s'agit de la plus intuitive vis à vis de l'étude du déterminisme de communications MPI dès lors que nous souhaitons vérifier que le même schéma de communication est présent pour toutes les exécutions. Si l'on désigne par Φ la propriété indiquant que le schéma de communication associé au processus $p \in P$ à l'instant t de l'exécution, c'est-à-dire le sous-ensemble des actions de p soumises à l'étude du déterminisme déjà exécutées, ordonnées par ordre d'exécution, est identique entre les exécutions déjà explorées, on obtient alors la formule : $\mathbf{AG}(\Phi)$.

La seconde approche consiste à raisonner de façon inverse. Tout comme pour la vérification de propriétés de vivacité exprimées avec la logique LTL, il est possible de travailler sur la négation de la propriété et de vérifier si celle-ci est satisfaite. Dans le contexte de l'étude du déterminisme, ceci consiste alors à partir de l'hypothèse que l'application étudiée est effectivement déterministe et à montrer qu'il existe un chemin d'exécution pour lequel le déterminisme n'est pas respecté. C'est dans ce contexte que la logique CTL (présentée dans le chapitre 1 de la partie 1, section 2.1.2) est utilisée. En effet, celle-ci permet de travailler sur des arbres et de spécifier des propriétés sur l'arbre des exécutions (interdépendantes) et non plus sur chaque chemin indépendamment des autres comme cela est réalisé en logique LTL. Sachant que l'étude du déterminisme repose sur la comparaison des exécutions entre elles, il est nécessaire de prendre en compte l'arbre dans son ensemble pour cette vérification.

Pour cela, la logique CTL propose le quantificateur de chemins **E** (*Exists*) qui permet de spécifier « *pour au moins un chemin* ». Il est alors possible de formuler la vérification du déterminisme des communications MPI en utilisant ce quantificateur. S'il existe deux chemins d'exécution aux schémas de communication différents l'un par rapport à l'autre, alors l'application n'est pas déterministe. La formule correspondante est alors : $\mathbf{EF}(\neg\Phi)$, où Φ correspond à la définition précédemment donnée.

La première formulation suppose de comparer les schémas de communications des exécutions deux à deux tandis que la seconde ne nécessite qu'un schéma de communication de référence avec lequel seront comparées chaque schéma de chaque exécution possible de l'application. Si une exécution possède un schéma différent, alors cela signifie que le déterminisme étudié n'est pas satisfait. C'est ainsi que nous proposons un algorithme de vérification basé sur cette seconde approche.

3.2 Algorithme de vérification

Bien que la propriété correspondant à l'étude du déterminisme des communications MPI soit exprimée à l'aide la logique CTL, l'algorithme que nous mettons en œuvre pour sa vérification est différent de celui habituellement implémenté pour la vérification de propriétés CTL. Il s'agit d'un algorithme *ad-hoc* spécialement adapté à notre étude. Celui-ci repose sur deux étapes : (1) la sauvegarde initiale d'un schéma de communication de référence et (2) la comparaison du schéma de communication de chaque exécution avec le schéma de référence.

3.2.1 Sauvegarde du schéma de communication de référence

La première étape de la vérification consiste donc à définir un schéma de communication de référence qui servira de point de comparaison avec toutes les exécutions. Si toutes les exécutions satisfont ce même schéma de communication pour chaque processus, alors tous les schémas sont identiques entre eux, le déterminisme est donc respecté.

C'est naturellement que nous avons choisi le premier chemin d'exécution exploré comme chemin de référence. C'est ainsi que nous commençons par explorer en profondeur celui-ci. Pour chaque processus, un ensemble d'actions est créé au début de l'exploration. À chaque action interceptée correspondant au déterminisme étudié (`Send` et/ou `Recv`) pendant l'exploration, celle-ci est ajoutée à l'ensemble du processus correspondant. Sachant qu'une communication s'effectue toujours en deux temps (`Send/Recv` + `Wait`), l'ensemble des informations nécessaires à l'évaluation du déterminisme pour chaque communication est mis à jour et complété au fur et à mesure de l'exploration en profondeur.

À la fin de l'exploration de ce premier chemin, nous retrouvons pour chaque processus un sous-ensemble des actions exécutées par celui-ci, ordonnées par ordre d'exécution et correspondant au déterminisme évalué. L'union de ces ensembles constitue ainsi le schéma de communication de référence qui servira de point de comparaison avec les schémas des autres exécutions.

3.2.2 Vérification sur toutes les exécutions

Une fois le schéma de communication de référence sauvegardé, nous explorons chacune des autres exécutions dans le but de comparer leur schéma avec le schéma initial.

Les communications étant sauvegardées de façon ordonnée, selon leur ordre d'exécution, il est possible d'effectuer la comparaison des schémas de communication de chaque processus à chaque nouvelle communication pendant l'exploration, sans attendre la fin de celle-ci. L'idée étant que si une seule communication diffère avec le chemin de référence, alors le déterminisme n'est pas satisfait, il n'est donc pas nécessaire de poursuivre l'exploration.

Ainsi, dès que l'ensemble des informations associées à une communication d'un processus est connu, celle-ci est soumise à comparaison avec la communication correspondante dans le schéma de communication de référence du processus. S'il s'agit de la $i^{\text{ème}}$ communication du processus p, la comparaison s'effectue avec la $i^{\text{ème}}$ communication du schéma de communication de référence de ce même processus.

Cette comparaison au fur et à mesure de l'exploration permet à la fois de stopper la vérification dès que le déterminisme n'est plus satisfait, mais également de ne pas sauvegarder toutes les communications. Dès qu'une communication a été traitée (l'ensemble de ses informations a été sauvegardé et soumis à comparaison avec le schéma de communication de référence), celle-ci n'est pas conservée en mémoire. Nous verrons dans les perspectives associées à ces travaux que ceci pourra être modifié par la suite dans le but d'étendre la vérification proposée sur le déterminisme des communications des applications MPI.

Nous présentons dans l'algorithme 6 la description simplifiée de notre algorithme.

3.3 Réduction sur les états visités

Contrairement à la vérification $\text{LTL}_{\text{-}\mathbf{X}}$, nous nous intéressons dans l'étude du déterminisme aux transitions et non pas aux états. Celles-ci correspondant à l'exécution d'une communication ou d'une action liée à l'une d'elle, nous travaillons avec ces dernières et non avec l'état du système résultant de son exécution. La mise en œuvre d'une technique de réduction de l'espace d'état exploré nécessite donc de préserver cette étude des transitions.

Ce type de vérification n'ayant jamais été réalisé dans la littérature, aucune technique de réduction spécifique à celui-ci n'est également proposée. C'est donc naturellement que nous nous sommes orientés vers la mise en œuvre de la réduction sur les états visités précédemment présentée. Sachant que des exécutions peuvent différer uniquement par une transition entre des processus différents, n'affectant donc pas le déterminisme, une réduction efficace de l'espace d'états est alors facilement envisageable. Ainsi, si nous déterminons qu'à partir d'un instant de l'exécution, nous revenons sur un état déjà visité sur une précédente exécution, l'exploration ayant continué après cet état, cela signifie que le reste du chemin exploré satisfait le déterminisme étudié (nous pouvons l'affirmer sans poursuivre l'exploration, ceci ayant déjà été vérifié pour l'exploration précédente). Le début de l'exploration ayant été évaluée au fur et à mesure, nous pouvons alors travailler sur un autre chemin d'exploration.

Algorithme 6 Algorithme de vérification du déterminisme des communications MPI

```
1: ExplorationStack ← EmptyStack() ;
2: InitialCommPattern ← EmptySet() ;
3: IncompleteComms ← EmptySet() ;
4: InitialPatternDone ← FALSE
5: add initialState to ExplorationStack ;
6: while get_size(ExplorationStack) > 0 do
7:     s = pop(ExplorationStack) ;
8:     while get_request(s) ≠ NULL do
9:         comm = get_comm(s) ;
10:        if get_comm_type(comm) = (Send || Recv) then
11:            add(comm) to IncompleteComms ;
12:        else if get_comm_type(comm) = Wait then
13:            complete_comm(IncompleteComms, comm) ;
14:        end if
15:        foreach successor t of s do
16:            add t to ExplorationStack ;
17:        end for
18:    end while
19:    delete s from ExplorationStack ;
20:    if InitialPatternDone = FALSE then
21:        InitialPatternDone ← TRUE
22:    end if
23: end while

24: procedure COMPLETE_COMM(comm)
25:    foreach currentComm in IncompleteComms do
26:        if currentComm = comm then
27:            complete_comm_information(currentComm, comm) ;
28:            if initialPatternDone = FALSE then
29:                add currentComm to InitialCommPattern ;
30:            else
31:                check_determinism(InitialCommPattern, currentComm) ;
32:            end if
33:            delete currentComm from IncompleteComms ;
34:        end if
35:    end for
36: end procedure
```

Celle-ci nécessite toutefois d'être affinée. En effet, nous avons expliqué dans la section précédente que les informations associées à une communication n'étaient pas immédiatement entièrement disponibles, la communication s'effectuant en deux temps, avec un `Wait` en plus du `Send` ou `Recv`. Or, dans sa version originale, la réduction sur les états visités peut détecter deux états comme identiques, impliquant donc de stopper l'exploration du chemin courant. Cependant, certaines communications peuvent être alors en cours de traitement (l'ensemble de leurs informations n'est pas encore connu, elles n'ont pas encore été soumises à comparaison avec le schéma de communication de référence). C'est ainsi que nous avons spécifié en plus que

toute réduction sur un état déjà visité n'est acceptée que s'il n'existe aucune communication incomplète et par conséquent non soumise à la vérification du déterminisme.

4 Évaluation expérimentale

L'étude manuelle du déterminisme dans des applications représentatives des systèmes HPC réalisée par Cappello et *al.* a amené à la conclusion que la grande majorité de ces applications est déterministe en terme de communications. Sur les 27 applications étudiées, seules 2 sont qualifiées de non-déterministes. Toutefois, n'ayant pas réussi à simplement compiler ces dernières, nous n'avons pu confirmer cette analyse sur ces exemples. Pour les applications dont une vérification a pu être réalisée à l'aide notre outil, aucune erreur n'a été trouvée. Nous estimons cependant que ceci ne permet pas vraiment de démontrer le bon fonctionnement de notre approche. Dans la même optique que le *bug finding*, l'objectif de ces travaux est d'être capable de déterminer automatiquement des cas pour lesquels certains déterminismes ne sont pas satisfaits et pour lesquels nous pouvons donc apporter une réponse différente des hypothèses habituellement émises.

C'est ainsi que nous avons mis en œuvre plusieurs exemples MPI dont l'implémentation inclut des schémas de communication identifiés comme non-déterministes. Nous avons alors réalisé une vérification dynamique formelle des différents déterminismes présentés dans la section 2 sur ces applications.

Le premier exemple vérifié est basé sur l'implémentation suivante :

```
1 if (rank % 3 == 0) {
2   MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
3   MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
4 }else{
5   MPI_Send(&rank, 1, MPI_INT, (rank / 3) * 3, 42, MPI_COMM_WORLD);
6 }
```

Dans cet exemple, le déterminisme des actions de type `Send` est garanti à la fois par l'utilisation de l'appel bloquant `MPI_Send` ainsi que par la destination spécifiée de manière fixe en paramètre.

Toutefois, les actions de type `Recv` ont comme paramètre de source `MPI_ANY_SOURCE` et un tag à `MPI_ANY_TAG`. Dès lors, le déterministe des réceptions ne peut être satisfait puisque plusieurs processus envoient à la même destination une communication avec le même tag. Le résultat de la vérification dynamique formelle confirme cette analyse. Pour cette expérience, nous avons exécuté l'application avec 3 processus.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>> Send-only-deterministic communications pattern >>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The recv communications pattern of the process 0 is different!
Different source for communication #2
>> Expanded states = 520
>> Visited states = 1476
>> Executed transitions = 1312
>> Send-deterministic : Yes
>> Recv-deterministic : No
```

Ces résultats sont obtenus en 4 secondes et sans la réduction sur les états visités. Nous affichons actuellement l'origine du non-déterminisme mais également la communication concernée. Avec réduction, le nombre d'états créés tombe à 339.

Un autre exemple est basé sur le même type d'implémentation telle que :

```
1 if (rank == 0) {
2   for(i=1; i < size; i++){
3     MPI_Irecv(&(buff[i-1]), 1, MPI_INT, MPI_ANY_SOURCE, i, MPI_COMM_WORLD, &req[i-1])
          ;
4   }
5   MPI_Waitall(size-1, req, status);
6 }else{
7   MPI_Send(&rank, 1, MPI_INT, 0, rank, MPI_COMM_WORLD);
8 }
```

Tout comme pour l'exemple précédent, le *send*-déterminisme est satisfait par la mise en œuvre d'un appel bloquant et le paramètre destination de `MPI_Send` qui est fixé à 0. Donc quelles que soient les exécutions, les processus dont le rank est différent de 0 ne font que des envois au processus 0. Les communications reçues sont elles cependant non-déterministes. En effet, car non bloquantes et complétées d'un `MPI_WaitAll` à la fin.

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>> Send-only-deterministic communications pattern >>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The recv communications pattern of the process 0 is different!
Different source for communication #2
>> Expanded states = 1263
>> Visited states = 4032
>> Executed transitions = 3584
>> Send-deterministic : Yes
>> Recv-deterministic : No
```

Nous avons vu dans les expériences précédentes que le non-déterminisme des communications reçues était dû à l'utilisation de communications non bloquantes ou du paramètre `MPI_ANY_SOURCE` associé à un `MPI_ANY_TAG`. Il s'agit donc d'une première partie du non-déterminisme dans le contexte de l'étude du déterminisme de toutes les communications.

Concernant les communications envoyées, la non satisfaction du déterminisme est souvent liée au non-déterminisme des communications reçues, avec l'utilisation, notamment du paramètre `status.MPI_Source`. Celui-ci spécifie que l'on reprend la source de la communication associée à `status`, correspondant à une communication reçue. Dès lors, si ces communications ne sont pas déterministes, les communications envoyées qui en découlent ne le sont pas également. Nous démontrons cette analyse par l'étude de l'implémentation suivante :

```
1 if (rank == 0) {
2   for(i=0; i < size - 1; i++){
3     MPI_Recv(&buff, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status)
          ;
4     MPI_Send(&buff, 1, MPI_INT, status.MPI_SOURCE, 42, MPI_COMM_WORLD);
5   }
6 }else{
7   MPI_Send(&rank, 1, MPI_INT, 0, 42, MPI_COMM_WORLD);
8   MPI_Recv(&buff, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
9 }
```

Par vérification dynamique formelle, le non-déterminisme est détecté en 1 minute sans réduction :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>> Non-deterministic communications pattern >>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The send communications pattern of the process 1 is different!
Different destination for communication #2
The recv communications pattern of the process 1 is different!
Different source for communication #1
>> Expanded states = 16028
>> Visited states = 80792
>> Executed transitions = 76040
>> Send-deterministic : No
>> Recv-deterministic : No
```

Avec l'activation de la réduction sur les états visités, le nombre d'états est divisé par 20 environ et le temps de vérification est de 15 secondes.

Enfin, nous nous sommes intéressés au schéma de communication basé sur l'implémentation suivante, extraite de [CGS10] :

```
1 for(i=0; i < size; i++){
2   MPI_Irecv(&buff[i], 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &req[2*i]);
3   MPI_Isend(&rank, 1, MPI_INT, i, 42, MPI_COMM_WORLD, &req[2*i+1]);
4 }
5 MPI_Waitall(2*size, req, status);
```

Déclaré comme *send*-déterministe par les auteurs, le schéma de communication correspondant ne l'est toutefois pas à cause du `MPI_Waitall`, comme le démontre notre outil :

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>> Non-deterministic communications pattern >>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
The send communications pattern of the process 2 is different!
Different type for communication #5
The recv communications pattern of the process 2 is different!
Different type for communication #4
>> Expanded states = 40
>> Visited states = 90
>> Executed transitions = 107
>> Send-deterministic : No
>> Recv-deterministic : No
```

5 Conclusion

Dans ce chapitre, nous avons présenté la vérification dynamique formelle du déterminisme des communications dans les applications MPI. Les protocoles de tolérance aux pannes actuels cherchent à exploiter la présence de certains déterminismes sans toutefois pouvoir garantir leur vérification. En l'absence d'une analyse complète et détaillée, un protocole plus générique est

alors mis en œuvre, au prix de performances moins bonnes. L'identification de plusieurs schémas de communications permet d'affirmer dans certains cas qu'une application est déterministe ou non en terme de communications mais cette analyse repose sur une revue manuelle du code source et peut donc être erronée, en plus d'être fastidieuse. C'est ainsi que nous avons mis en œuvre une approche permettant de vérifier dynamiquement et formellement différents déterminismes sur des applications réelles MPI. Une évaluation expérimentale sur plusieurs exemples démontre l'efficacité de notre approche.

Du fait de protocoles adaptés à des applications dont les communications sont déterministes, les applications MPI en HPC sont cependant très souvent, dans une grande majorité, déterministes (au moins *send*-déterministes). Leur développement est naturellement soumis à ces contraintes afin de mettre en œuvre facilement des protocoles de tolérance aux pannes existants et performants. Toutefois, nous avons vu que dans certains cas ce déterminisme peut avoir un impact sur les performances, si bien que certaines applications tentent de le limiter dans le but d'augmenter la concurrence, réduisant ainsi les coûts de synchronisation et améliorant l'ordonnancement. En permettant une étude automatique et détaillée de ce type d'applications, il sera alors possible de mettre en œuvre de nouveaux protocoles de tolérance aux pannes efficaces.

Quatrième partie

Conclusion et Perspectives

Chapitre 1

Conclusion

La vérification des systèmes distribués constitue un enjeu majeur dans l'amélioration de leur qualité. Certaines défaillances pouvant avoir des conséquences non négligeables d'un point de vue matériel et financier, voire humain dans certains cas, assurer la qualité d'un système par sa correction est essentiel. Cependant, les systèmes distribués se caractérisent principalement par leur hétérogénéité et une complexité de plus en plus importante à mesure que leur taille augmente. Il devient alors difficile de mettre en œuvre des approches permettant leur étude et leur vérification.

Nous avons présenté en partie I de ce document les différentes techniques de vérification disponibles actuellement pour l'étude des systèmes distribués. Si le test et la simulation représentent les approches les plus couramment appliquées, celles-ci souffrent toutefois d'une limite forte dans le contexte particulier des systèmes distribués : elles ne permettent pas la gestion complète du non-déterminisme des exécutions, caractéristique commune à la grande majorité de ce type de système. Ces approches sont donc insuffisantes pour garantir une vérification exhaustive. C'est dans ce contexte que les travaux présentés dans ce document portent sur une vérification formelle des systèmes distribués. Plus précisément, nous mettons en œuvre une vérification dynamique formelle des systèmes distribués en appliquant les techniques du *Software Model Checking* sur des applications distribuées réelles. L'enjeu de cette approche est de réussir à appliquer les techniques associées au *model checking* dans le cadre d'une vérification sur l'implémentation réelle d'une application et non plus sur un modèle de cette dernière. Si cette approche n'est pas nouvelle, son application à l'étude des systèmes distribués reste toutefois modérée, en particulier pour les applications MPI écrites en C, C++ et Fortran, faute de techniques et outils adaptés.

C'est ainsi que nous avons présenté dans un premier temps, dans la partie II, la mise en œuvre d'une analyse sémantique dynamique par introspection mémoire d'un état système permettant la détection d'états identiques. L'analyse des informations dynamiques qui constituent un état système au cours de son exécution est une étape préliminaire importante pour l'étude et la vérification approfondies d'applications réelles. Grâce à l'utilisation de techniques et outils de *debug*, tels que DWARF et libunwind, nous pouvons reconstruire sémantiquement l'état d'un système permettant ainsi d'évaluer l'égalité entre plusieurs états systèmes. Comme détaillé dans le chapitre 3, cette contribution permet alors l'application de la réduction sur les états visités issue du *Software model checking* traditionnel, mais également la vérification d'applications dont l'exécution est infinie ainsi que l'analyse partielle de la terminaison de programmes. À notre connaissance, il s'agit de la première approche permettant l'analyse sémantique d'états systèmes

sur des applications dont l'implémentation ne met pas en œuvre un système de ramasse-miettes, permettant ainsi de ne plus restreindre la vérification uniquement à des langages de haut niveau.

Dans la partie III, nous nous sommes intéressés à la vérification de certaines propriétés temporelles dont l'intérêt dans le contexte des systèmes distribués a été présenté dans le chapitre 1 de la partie I.

Le chapitre 1 de la partie III aborde la vérification de propriétés de vivacité exprimées à l'aide de la logique LTL$_{-\mathbf{X}}$. Basée sur la détection de cycles d'exécutions infinies dans lesquels la propriété spécifiée n'est pas satisfaite, nous nous sommes naturellement appuyés sur les travaux de la partie précédente. Nous avons associé à cela la construction automatique de l'automate de Büchi de la négation de la propriété et l'implémentation d'un nouvel algorithme d'exploration en profondeur. Celui-ci permet la recherche de plusieurs cycles acceptants en un unique parcours en profondeur en mettant également en œuvre la réduction sur les états visités. Ces travaux ont été évalués à travers la vérification d'une propriété de vivacité sur une application mettant en œuvre l'algorithme d'exclusion mutuelle centralisée dont l'exécution est infinie. Nous avons démontré alors la capacité de notre outil à détecter les cycles acceptants qui caractérisent une violation de propriété.

Enfin, nous avons présenté dans le chapitre 2 de la partie III l'étude du déterminisme des communications dans les applications MPI. Grâce à la caractérisation automatique du déterminisme des communications sur une application donnée, il est possible de mettre en œuvre des protocoles de tolérance aux pannes adaptés. Cette étude n'a jusqu'à présent fait l'objet d'aucune contribution en terme de vérification formelle automatique. Bien que les travaux sur les protocoles de tolérance aux pannes exploitent la présence de certains déterminismes dans les applications MPI étudiées, aucune vérification formelle de ce comportement n'a été réalisée dans la littérature. Après une présentation des différents déterminismes étudiés, nous avons mis en œuvre leur vérification en utilisant la logique CTL pour la formulation des propriétés. Puis, grâce à un algorithme d'exploration et de vérification *ad-hoc*, nous avons déterminé formellement et automatiquement le type de déterminisme associé à diverses applications réelles MPI.

Chapitre 2

Perspectives

Les perspectives ouvertes par les travaux présentés dans ce document sont multiples. Nous les regroupons en trois catégories principales : les perspectives basées sur un prolongement de l'existant, les perspectives de vérification d'autres types de propriétés et les perspectives d'extension du champ d'application des travaux présentés dans ce document.

1 Prolongement de l'existant

Amélioration de l'heuristique de l'analyse sémantique. L'heuristique mise en œuvre pour la détection d'états sémantiquement identiques souffre de plusieurs limites que nous avons détaillées dans la section 4 du chapitre 2 de la partie II. Ces limites sont incontournables dès lors que nous nous restreignons à une analyse dynamique de la mémoire au cours de l'exécution du système. Nous devons donc mettre en œuvre une approche permettant d'accéder à ces informations manquantes afin de renforcer notre analyse sémantique.

En ne conservant qu'une analyse dynamique, une approche serait de surcharger la bibliothèque d'allocation `malloc` afin de tracer et répertorier l'ensemble des allocations et libérations mémoires mais surtout de spécifier pour chacune le type des données contenues dans la zone mémoire correspondante. Cependant, cette approche est à la fois extrêmement intrusive et coûteuse à mettre en œuvre car elle nécessite de modifier toutes les allocations actuellement dans notre outil de vérification mais également celles dans les implémentations des systèmes étudiés. Ceci va alors à l'encontre de notre objectif à long terme qui est de proposer un outil permettant de vérifier des applications sans les modifier. L'approche envisagée est alors de combiner notre analyse dynamique avec une analyse statique. Celle-ci permettrait en effet de déterminer avant l'analyse dynamique le type des données associées à des pointeurs génériques en mémoire et donc de garantir une analyse sémantique correcte.

Implémentation d'autres réductions. Suite à l'implémentation des travaux présentés dans ce document, deux réductions sont proposées au sein de SimGridMC : DPOR et la réduction sur les états visités. La réduction dynamique par ordre partiel ne peut s'appliquer actuellement qu'à la vérification de propriétés de sûreté tandis que la réduction sur les états visités est disponible quel que soit le type de vérification. Toutefois, celles-ci ne peuvent actuellement pas être combinées. Un premier objectif serait donc de permettre l'activation de ces deux réductions lors d'une vérification. Selon si la vérification porte sur une propriété de sûreté exprimée à l'aide d'une simple assertion ou sur une propriété exprimée à l'aide la logique LTL, la mise en œuvre de cette double réduction soulève des difficultés différentes.

En effet, dans le cas d'une propriété de sûreté sous forme d'assertion, le problème réside dans la correction de DPOR. Cette réduction est actuellement basée sur une exploration en profondeur complète d'un chemin d'exécution puis d'une analyse de la dépendance des transitions précédemment exécutées sur ce chemin. Pour chaque paire de transitions dépendantes, DPOR crée un nouveau chemin à explorer. Cependant, cette analyse ne peut être réalisée que lorsque le chemin a été entièrement exploré et que chaque transition exécutée peut être caractérisée d'après les informations collectées durant l'exécution. Si nous combinons à cette réduction la réduction sur les états visités, cela signifie que l'exploration d'un chemin peut être arrêtée avant la fin du chemin (dès lors qu'un état a été détecté comme déjà visité). Sachant que nous travaillons dynamiquement et que l'espace d'états est construit à la volée au cours de l'exploration et de la vérification, nous ne connaissons pas les transitions qui devaient être exécutées après l'état sur lequel nous stoppons l'exploration. De même, toutes les transitions qui étaient actives dans un état mais qui n'ont pas été incluses dans celui-ci au moment de son exploration, ne peuvent être retrouvées sans retourner à nouveau sur cet état. Elles sont donc perdues une fois l'exploration de cet état passée. DPOR ne possède alors pas toutes les informations nécessaires pour évaluer la dépendance entre les transitions et ainsi activer de nouveaux chemins d'exécution. Nous pouvons alors ne pas voir des nouveaux chemins d'exécution à explorer, corrompant alors l'exhaustivité de la vérification et donc sa correction.

Plusieurs approches peuvent alors être mises en œuvre pour pallier ceci. La première serait de revoir le théorème d'indépendance des transitions afin de tenir compte d'une éventuelle borne d'exploration et ainsi garantir que toutes les transitions dépendantes sont effectivement explorées et vérifiées. La seconde approche consiste à appliquer la forme duale de DPOR. Actuellement, DPOR suppose initialement que toutes les transitions actives au sein d'un même état sont indépendantes entre elles. Puis, lors du *backtracking*, selon le théorème d'indépendance, toute transition déclarée comme dépendante est activée dans les états précédemment explorés sur le chemin courant, créant ainsi de nouveaux chemins à explorer. Cependant, si l'exploration est stoppée en cours, des transitions dépendantes peuvent être perdues car non exécutées et donc non analysées par DPOR. L'approche duale consiste alors à considérer dans chaque état toutes les transitions actives comme dépendantes et à désactiver celles qui peuvent être analysées par DPOR et détectées comme indépendantes. Ceci garantit que toute transition non exécutée lors de la première exploration d'un état est préservée et donc explorée ensuite si DPOR ne peut appliquer le théorème d'indépendance, faute d'informations suffisantes. Il peut donc subsister des transitions indépendantes entre elles réduisant l'efficacité de DPOR seule, mais la correction de la double réduction et donc de la vérification est garantie.

Dans le cas d'une vérification de propriété LTL, la combinaison des deux réductions peut couper des cycles dans lesquels la propriété serait violée, corrompant ainsi la correction de la vérification. En effet, DPOR, dans sa forme actuelle, ne tient pas compte dans son théorème d'indépendance de l'éventuelle modification des propositions atomiques par l'exécution d'une transition. Il peut alors déclarer deux transitions comme indépendantes ce qui amène à l'exploration d'un seul ordre d'exécution de ces transitions et ainsi supprimer l'exploration d'un chemin dans lequel les propositions atomiques sont modifiées, soulevant potentiellement une violation de propriété.

Une approche pour garantir une réduction correcte de DPOR avec la vérification de propriétés $\text{LTL}_{\setminus\mathbf{X}}$ consiste à intégrer la notion d'invisibilité des transitions. Une transition est dite invisible si son exécution depuis n'importe quel état n'affecte pas les propositions atomiques de la formule LTL. Celle-ci nécessite donc de réussir à caractériser une transition pour tous ses cas d'exécution. Sachant que l'ensemble des exécutions est découvert au fur et à mesure des

explorations, cette notion d'invisibilité ne peut être évaluée que pour certains cas évidents. Une présentation détaillée du problème est exposée dans [Cla09]. Pour évaluer cette invisibilité, une approche possible est alors de mettre en œuvre une analyse statique en plus de notre analyse dynamique afin de déterminer, dans un premier temps, les différents cas d'exécution d'une transition puis d'évaluer l'impact de son exécution sur l'état du système et ainsi mettre en œuvre la double réduction.

Enfin, nos travaux étant actuellement concentrés sur la vérification des applications MPI, une autre réduction envisagée est celle détaillée dans [VGK08] mise en œuvre par ISP. Cette réduction, basée sur DPOR, met en œuvre un algorithme POE (*Partial Order reduction avoiding Elusive interleavings*) qui exploite la complétion sémantique désordonnée des appels MPI. Par exemple, si un processus effectue deux `Send` consécutifs non-bloquants vers des processus différents, le second envoi peut se terminer (être complété) avant le premier. L'algorithme qui implémente la réduction intercepte chaque appel MPI et le retarde selon des règles d'ordonnancement établies d'après la sémantique des appels. Ceci permet alors de créer des ensembles `Send - Recv` qui seront exécutés en une transition atomique lors de l'exploration ou en plusieurs transitions, réduisant potentiellement l'espace d'états exploré. La mise en œuvre de cette réduction dans SimGrid nécessite alors d'implémenter une analyse plus approfondie de la sémantique des appels MPI.

Amélioration de l'utilisabilité. Bien que fonctionnel, notre outil de vérification SimGridMC reste difficilement abordable pour des personnes non-initiées. Si le lancement de l'outil est facile à mettre en œuvre (des options doivent être ajoutées dans la ligne de commande correspondant à l'exécution de l'application, dont la description en PROMELA de l'algorithme de Büchi correspondant à la propriété à vérifier), l'analyse du résultat de la vérification l'est moins. Une description verbeuse du chemin d'exécution contenant la violation de la propriété est fournie à l'utilisateur mais celle-ci peut être extrêmement longue. De même, pour plus de lisibilité des informations, nous limitons les données affichées dans le résultat. Il devient alors difficile d'analyser plus en détail la violation de propriété détectée par l'outil, ce qui réduit fortement l'intérêt de la vérification s'il est difficile pour l'utilisateur de corriger son application.

Pour cela, des travaux autour du développement d'un outil de visualisation dynamique ont été initiés dans le cadre d'un projet étudiant mais n'ont pour l'heure pas abouti. Ce projet a pour but de développer en Java (par exemple), un outil de visualisation dynamique de la vérification effectuée par SimGridMC. Un premier objectif serait de pouvoir visualiser dynamiquement l'ensemble des états parcourus lors d'une exploration, d'après une trace donnée. Ensuite, la visualisation obtenue après exploration pourrait être enrichie avec des informations collectées par notre outil, mais non traitées dans les traces de résultat. Nous souhaiterions par exemple pour chaque état visualiser la réduction qui a été appliquée et dans quelles conditions (quelle partie du théorème d'indépendance a été appliquée à cet instant, par exemple). Enfin, un objectif plus ambitieux serait d'obtenir une visualisation dynamique accessible durant la vérification pour ainsi évaluer en temps réel l'analyse faite par notre outil de vérification et éventuellement de guider l'exploration vers des chemins prometteurs d'après les informations disponibles dans chaque état, obtenues notamment par l'introspection mémoire d'un état système.

2 Vérification d'autres types de propriétés

***Eventual consistency* - Linéarisabilité.** La cohérence des données fait partie des propriétés de correction particulièrement intéressantes dans le contexte des systèmes distribués. Celle-ci

assure que si plus aucune modification n'est effectuée sur une donnée accessible à l'ensemble des entités du système, alors toute entité y accédant obtiendra la dernière valeur courante de cette donnée. Bien que les systèmes distribués ne partagent pas les espaces mémoires entre processus, il est toutefois courant qu'une même donnée soit exploitée par plusieurs processus. Afin de garantir que chaque processus accède à la dernière valeur de cette donnée, des communications peuvent être mises en œuvre pour transmettre cette information ou un mécanisme de réplication de données peut être inclus dans le système, réduisant ainsi le nombre de communications effectuées au cours de l'exécution impactant potentiellement favorablement les performances. Quelle que soit l'approche choisie, il reste indispensable de garantir que toute exploitation de données est effectivement réalisée à partir de la dernière valeur, dès lors que plus aucune modification par l'une des entités du système n'y est apportée.

La propriété de cohérence des données pouvant être formulée à l'aide de la logique LTL, l'ensemble des travaux présentés dans ce document constitue alors une base solide pour la mise en œuvre de cette vérification.

Cette notion de cohérence des données est également associée à la linéarisabilité dans le contexte des systèmes concurrents avec *threads*. Le principe de la linéarisabilité est de garantir que l'exécution simultanée d'opérations concurrentes sur un même objet mène au même état que lors de l'exécution séquentielle de ces mêmes opérations. Ceci garantit ainsi que si plusieurs *threads* modifient en même temps un élément partagé de la mémoire, le résultat sera identique à celui obtenu si les accès et modifications de cet élément avaient été séquentialisés entre les *threads*.

SimGridMC ne permettant actuellement pas la vérification de programmes concurrents *multi-threads*, la vérification de la linéarisabilité nécessite un travail préalable d'extension du champ d'application présenté dans la section suivante de ce chapitre.

Équité et progression des processus. L'équité des processus est une propriété portant sur la progression des processus au cours de l'exécution du système distribué étudié. Elle garantit que le choix du processus à exécuter à l'instant t est effectué de façon équitable selon des règles d'équité définies au sein de l'ordonnanceur (*scheduler*) du système étudié. Ces règles définissent des niveaux d'équité plus ou moins forts et déterminent si un processus doit être exécuté ou non, une ou plusieurs fois au cours de l'exécution. Cette notion d'équité est cruciale dans la vérification des systèmes distribués car la non-progression de certains processus peut dans certains cas cacher des erreurs au cours de l'exécution qui seront alors plus dures à détecter avec l'outil de vérification. Cependant, l'équité étant définie pour un système donné mettant en œuvre son propre mécanisme d'ordonnancement des processus à chaque instant de l'exécution, il est possible qu'une propriété soit considérée comme violée sur un système d'après ses règles d'équité mais pas sur un autre avec des règles d'équité différentes.

Actuellement, l'équité appliquée au cours de l'exploration est celle définie au sein de SimGrid ou éventuellement celle directement intégrée dans la formulation des propriétés de vivacité avec la logique LTL. Afin de garantir la vérification quelles que soient les règles d'équité définies, une perspective envisagée est de permettre avant toute vérification de définir l'équité à appliquer au cours de l'exploration. Cette équité pourrait également être modulée au cours de la vérification pour guider l'exploration vers des chemins jugés plus pertinents vis-à-vis de la propriété.

Dans le même contexte, nous avons présenté dans ce document les cycles de non-progression, définis selon l'absence de création de nouveaux états à explorer au cours de la vérification d'un chemin. Ceci arrive donc lorsque deux états sémantiquement identiques sont rencontrés sur un même chemin d'exécution. Grâce à l'analyse sémantique présentée en partie II, ces cycles de

non-progression sont automatiquement détectés par notre outil sans spécification de propriété particulière. Cependant, cette notion de cycle de non-progression fait également référence dans la littérature à la notion d'actions de progrès définie par SPIN dans [FS09]. En définissant des actions dites de progrès, il est possible de détecter des cas d'exécutions durant lesquels le système n'évolue plus d'après la définition de sa progression bien que son état global en mémoire change perpétuellement. Il s'agit donc d'une non-progression d'un point de vue macro (au niveau des actions des processus) et non plus micro (au niveau de la mémoire du système et de chaque processus).

Une perspective est donc de mettre en œuvre cette notion de progrès non plus seulement basée sur l'état du système en mémoire mais d'après la sémantique des actions au cours de l'exécution, définie par l'utilisateur. Ceci nécessite donc de formaliser la sémantique des actions des processus au sein de SimGrid à travers la logique TLA+ (*Temporal Logic of Actions*), comme cela a été fait pour le théorème d'indépendance appliqué avec DPOR par exemple, mais également d'adapter éventuellement le modèle d'atomicité mis en œuvre pour la définition des transitions de l'espace d'états.

Tolérance aux fautes. Nous avons présenté dans le chapitre 2 de la partie III l'étude du déterminisme des communications MPI dans le contexte du développement de protocoles de tolérances aux pannes adaptés aux systèmes distribués HPC. La tolérance aux pannes est un sujet de grande actualité dans le contexte HPC grâce au développement de superordinateurs toujours plus grands et plus puissants. La vérification des protocoles mettant en œuvre cette tolérance l'est tout autant. Dans [JKS+13], les auteurs soulèvent la difficulté de modéliser ces protocoles pour ensuite les soumettre à une vérification par *model checking* et proposent une approche en ce sens.

Une perspective possible est donc de permettre la vérification dynamique formelle de ces protocoles en travaillant directement sur leur implémentation réelle. Dès lors, nous pouvons imaginer qu'une troisième donnée serait fournie en entrée de notre outil : l'application étudiée, la propriété à vérifier sur celle-ci et le protocole de tolérance aux pannes. La difficulté majeure dans ce contexte réside dans la notion d'exhaustivité de la vérification. Vérifier un protocole de tolérance aux pannes signifie s'assurer que dans tous les cas possibles de pannes, le système progresse et se termine correctement. Bien que complexe à première vue, cet axe de recherche constitue un réel enjeu à venir.

3 Extension du champ d'application

Applications distribuées *multi-threads*. Le champ d'application des travaux présentés dans ce document se limite actuellement aux applications distribuées *mono-thread*. Une perspective est alors d'étendre la vérification dynamique formelle à des applications distribuées *multi-threads*. Parmi les propriétés intéressantes à évaluer sur ce type de système, nous retrouvons à nouveau la cohérence des données, dès lors que des zones mémoires sont partagées entre plusieurs entités. En effet, ce type d'application met en œuvre des communications MPI *one-sided* qui exploitent les zones mémoires partagées entre les *threads*. Basées sur une interface RMA (*Remote Memory Access*), les communications *one-sided* permettent à un unique processus MPI d'initier une communication et de la traiter à la fois du côté de la source mais également du côté de la destination, qu'il soit la source ou la destination de cette communication. L'étape de *matching* entre un `Send` et un `Recv` durant laquelle les paramètres associées à une communication et le transfert des données sont mutuellement définis par les processus inclus dans cette

communication est alors omise. Cet accès direct réduit alors toute observation possible depuis l'outil de vérification et donc potentiellement la découverte de nouveaux chemins d'exécution possibles pouvant contenir des violations de propriétés. L'une des difficultés majeures dans la vérification de ces applications est donc d'identifier ces zones mémoires à accès direct, de les analyser durant l'exécution et d'intercepter toute modification au sein de celles-ci pouvant affecter l'état global du système.

Une approche pour la gestion des zones mémoires à accès direct est de combiner analyse statique et dynamique avec une instrumentation du code. L'analyse statique permettrait dans un premier temps de déterminer les zones mémoires déclarées pour les communications *one-sided*, puis d'augmenter le code de l'application pour que les accès directs à ces zones passent par une fonction adéquate permettant à l'outil de vérification d'évaluer les interactions avec cette zone mémoire et donc d'inclure ces dernières dans la vérification. Pour cela, il peut être envisagé de protéger les zones ciblées par des accesseurs (*getter* et *setter*) explicites. Des travaux en ce sens sont initiés dans [CDT+14]. La particularité de ces travaux est que l'analyse est réalisée *offline* après la récupération d'une trace d'exécution. Un graphe orienté acyclique est construit durant l'analyse pour représenter les accès aux données partagées, dans lequel les évènements sont ordonnés d'après la relation arrivé-avant définie dans le modèle mémoire *one-sided* MPI. L'outil évalue alors les violations de cohérence qui ont eu lieu ainsi que celles qui auraient pu avoir lieu durant l'exécution d'après une table de compatibilité des opérations. La mise en œuvre de cette approche au sein de SimGridMC permettrait alors de réaliser cette vérification de façon dynamique durant l'exécution sans attendre la fin de celle-ci.

Applications non MPI. MPI est le standard le plus utilisé actuellement dans le contexte des systèmes HPC. Cependant, de nouveaux types d'applications autres que MPI émergent, nécessitant alors de nouveaux outils pour leur vérification. Parmi ces nouvelles applications, StarPU [ATNW11] constitue l'un des axes majeurs à explorer. Il s'agit d'une bibliothèque de programmation de tâches pour les architectures hybrides. Celle-ci permet l'implémentation de tâches pour CPU ou GPU et la construction de graphes de tâches associés à une optimisation des ordonnancements hétérogènes, des transferts de données et de la communication entre *clusters*.

StarPU pouvant déjà s'exécuter au dessus de SimGrid, le support technique de base est déjà disponible. Pour la partie vérification, une approche envisagée est la mise en œuvre de la technique DIR (*Dynamic Interface Reduction*) dont l'idée principale est de considérer chaque composant séparément lors de la vérification. Ceci s'applique parfaitement dans le contexte des architectures hybrides ciblées par StarPU. En définissant l'ensemble des composants qui constitue le système étudié ainsi qu'une interface associée à chacun d'eux, il est possible de déterminer le comportement de chaque composant et ainsi de le vérifier d'après une propriété donnée. Cette technique est notamment mise en œuvre par DeMeter dans [GWZ+11] dans un contexte plus général.

Applications Java. Une dernière catégorie d'applications à vérifier, envisagée en perspectives des travaux présentés dans ce document correspond aux applications Java. L'ensemble de nos travaux a été appliqué à la vérification d'applications écrites en C/C++ ou Fortran. Si ces langages sont habituellement utilisés dans le contexte des systèmes distribués en particulier en HPC, il subsiste toutefois une part non négligeable d'applications écrites en Java dans d'autres contextes (P2P, *Internet of things*, ...). Permettre leur vérification dynamique formelle constitue donc un axe de recherche à envisager. Nous avons présenté dans le chapitre 2 de la partie I JavaPathFinder, l'outil de vérification de référence pour la vérification formelle d'applications

Java. S'il permet la vérification d'applications distribuées notamment avec l'extension proposée dans [SM14], le champ d'application reste toutefois limité aux applications dont les processus communiquent à travers des *sockets* et des opérations bloquantes.

Grâce au support Java dans SimGrid, il serait envisageable d'ajouter également à SimGridMC le support pour la vérification d'applications Java. Dans un premier temps, l'objectif serait de permettre la vérification de propriétés de sûreté exprimées sous forme d'assertions puis d'étendre ensuite cette vérification aux propriétés de vivacité. Les travaux réalisés au cours de cette thèse ainsi que ceux initiés lors de la thèse précédente de Cristian Rosa constituent une base à considérer dans la mise en œuvre de cette perspective.

Les travaux présentés dans ce document contribuent au développement d'un outil de vérification unique permettant à la fois l'étude et la vérification de systèmes distribués, à travers l'étude de leurs performances et la vérification de propriétés de correction sur des implémentations réelles de ces derniers. Pour cela, il établit un rapprochement entre les techniques de simulation et celles de vérification dynamique formelle. Un tel outil permet alors de simplifier le développement de systèmes distribués au comportement correct mais également de supprimer le coût de développement de multiples modèles d'une même application habituellement nécessaire pour une vérification formelle. Face à la complexité mais également une popularité grandissantes des systèmes distribués, il devient nécessaire de développer de nouvelles approches et outils qui contribuent à une meilleure compréhension de leur comportement pour garantir une meilleure qualité.

Bibliographie

[ABC+13] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8) :1978–2001, 2013.

[AEW09] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[Ake78] Sheldon B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, 100(6) :509–516, 1978.

[AM98] Lorenzo Alvisi and Keith Marzullo. Message logging : Pessimistic, optimistic, causal, and optimal. *Software Engineering, IEEE Transactions on*, 24(2) :149–159, 1998.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later : using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2) :66–75, 2010.

[BBDL+13] Tomáš Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínskỳ, and Jan Strejček. Compositional approach to suspension and other improvements to ltl translation. In *Model Checking Software*, pages 81–98. Springer, 2013.

[BBMS13] S Bohm, Marek Behálek, Ondrej Meca, and M Surkovsky. Visual programming of mpi applications : Debugging and performance analysis. In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pages 1495–1502. IEEE, 2013.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[BCG87] Michael C Browne, Edmund M Clarke, and O Grümberg. Characterizing kripke structures in temporal logic. In *TAPSOFT'87*, pages 256–270, 1987.

[BDG+13] Paul Bédaride, Augustin Degomme, Stéphane Genaud, Arnaud Legrand, George Markomanolis, Martin Quinson, Mark Lee Stillwell, Frédéric Suter, Brice Videau,

et al. Toward better simulation of mpi applications on ethernet/tcp networks. In *PMBS13-4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2013.

[BDODF09] Thomas Bouton, Diego Caminha B De Oliveira, David Déharbe, and Pascal Fontaine. verit : an open, trustable and efficient smt-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.

[Bei03] Boris Beizer. *Software testing techniques*. 2003.

[BGRT05] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Compiler Construction*, pages 250–254, 2005.

[BHJM07] Dirk Beyer, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6) :505–525, 2007.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap : A binary analysis platform. In *Computer Aided Verification*, pages 463–469, 2011.

[Bje05] Per Bjesse. What is formal verification ? *SIGDA Newsl.*, 35(24), 2005.

[BKŘS12] Tomáš Babiak, Mojmír Křetínskỳ, Vojtěch Řehák, and Jan Strejček. Ltl to büchi automata translation : Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 95–109. Springer, 2012.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with slam. *Communications of the ACM*, 54(7) :68–76, 2011.

[BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. 1999.

[BRB+09] Aurelien Bouteiller, Thomas Ropars, George Bosilca, Christine Morin, and Jack Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–9. IEEE, 2009.

[Bü60] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik und Grundl. Math.*, 6 :66–92, 1960.

[CBDT06] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. *Best kept secrets of peer code review*. 2006.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1) :7–34, 2001.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification*, pages 334–342. Springer, 2014.

[CCG+04] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *Software Engineering, IEEE Transactions on*, 30(6) :388–402, 2004.

[CCR+03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab : an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3) :3–12, 2003.

[CDD+05] Franck Cappello, Frédéric Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, et al. Grid'5000 : a large scale, reconfigurable, controlable and monitorable grid platform. In *6th IEEE/ACM International Workshop on Grid Computing-GRID 2005*, 2005.

[CDSM10] Rosa Cristian Daniel, Merz Stephan, and Quinson Martin. A simple model of communication apis - application to dynamic partial-order reduction. *ECEASST*, 2010.

[CDT+14] Zhezhe Chen, James Dinan, Zhen Tang, Pavan Balaji, Hua Zhong, Jun Wei, Tao Huang, and Feng Qin. Mc-checker : detecting memory consistency errors in mpi one-sided applications. In *High Performance Computing, Networking, Storage and Analysis, SC14 : International Conference for*, pages 499–510. IEEE, 2014.

[CE82] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. 1982.

[CEFJ96] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2) :77–104, 1996.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[CGL+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10) :2899–2917, June 2014.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CGS10] Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8. IEEE, 2010.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol : An interpolating smt solver. In *Model Checking Software*, pages 248–254. Springer, 2012.

[CISW05] Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The comfort reasoning framework. In *Computer Aided Verification*, pages 164–169. Springer, 2005.

[CKNZ12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.

[CL85] K Mani Chandy and Leslie Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1) :63–75, 1985.

[Cla98] James M Clarke. Automated test generation from a behavioral model. In *Proceedings of Pacific Northwest Software Quality Conference. IEEE Press*, 1998.

[Cla09] Edmund Clarke. Model checking with the partial order reduction, 2009.

[CMCHG96] E Clarke, K McMillan, S Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422, 1996.

[CMM13] Katherine E Coons, Madan Musuvathi, and Kathryn S McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 833–848. ACM, 2013.

[Cou99] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *FM'99—Formal Methods*, pages 253–271. Springer, 1999.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5) :88–98, 2011.

[CVWY93] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Computer-aided Verification*, pages 129–142. Springer, 1993.

[Del11] Jean-Paul Delahaye. Du rêve à la réalité des preuves. *Pour la science*, (402) :90–95, 2011.

[DG99] Marcello D'Agostino and Dov M Gabbay. *Handbook of tableau methods*. Springer, 1999.

[DGP+97] Dingzhu Du, Jun Gu, Panos M Pardalos, et al. *Satisfiability problem : theory and applications : DIMACS Workshop, March 11-13, 1996*, volume 35. American Mathematical Soc., 1997.

[DLP04] Alexandre Duret-Lutz and Denis Poitrenaud. Spot : an extensible model checking library using transition-based generalized büchi automata. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2004)*, pages 76–83. IEEE, 2004.

[DM05] Alastair F Donaldson and Alice Miller. Automatic symmetry detection for model checking using computational group theory. In *FM 2005 : Formal Methods*, pages 481–496. Springer, 2005.

[DM06] Alastair F Donaldson and Alice Miller. Exact and approximate strategies for symmetry reduction in model checking. In *FM 2006 : Formal Methods*, pages 541–556. Springer, 2006.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. 2008.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories : introduction and applications. *Communications of the ACM*, 54(9) :69–77, 2011.

[EH00] Kousha Etessami and Gerard J Holzmann. Optimizing büchi automata. In *CONCUR 2000—Concurrency Theory*, pages 153–168. Springer, 2000.

[ES89] E Allen Emerson and Jai Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 123–172. 1989.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[FS09] David Faragó and Peter H Schmitt. Improving non-progress cycle checks. In *Model Checking Software*, pages 50–67. Springer, 2009.

[Gho10] Sukumar Ghosh. *Distributed systems : an algorithmic approach.* 2010.

[GHRF94] Dov M Gabbay, Ian Hodkinson, Mark Reynolds, and Marcelo Finger. *Temporal logic : mathematical foundations and computational aspects*, volume 1. 1994.

[GKS+11] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of mpi-based parallel programs. *Commun. ACM*, pages 82–91, 2011.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In *Computer Aided Verification*, pages 53–65. Springer, 2001.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1991.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[God05] Patrice Godefroid. Software model checking : The verisoft approach. *Formal Methods in System Design*, 26(2) :77–101, 2005.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[GvLH+96] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems : an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[GWZ+11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011.

[Ham94] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[Har13] John Harrisson. A survey of automated theorem proving, 2013.

[Hav99] Klaus Havelund. Java pathfinder, a translator from java to promela. In *SPIN*, volume 1680, page 152, 1999.

[HD01] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *CONCUR 2001—Concurrency Theory*, pages 39–58. Springer, 2001.

[HH14] Gérard Huet and Hugo Herbelin. 30 years of research and development around coq. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 249–250, 2014.

[HHS01] Gerard J Holzmann and Margaret H Smith. Software model checking : extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2) :65–79, 2001.

[HJMS02] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1) :58–70, 2002.

[HLL94] Joseph R. Horgan, Saul London, and Michael R Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9) :60–69, 1994.

[Hol04] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HPY96] Gerard J Holzmann, Doron Peled, and Mihalis Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, volume 32, pages 81–89, 1996.

[HS99] Gerard J Holzmann and Margaret H Smith. Software model checking. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497. Springer, 1999.

[Ios01] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 254–261. IEEE, 2001.

[Ios04] Radu Iosif. Symmetry reductions for model checking of concurrent dynamic software. *International Journal on Software Tools for Technology Transfer*, 6(4) :302–319, 2004.

[ISG+05] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, Malay K Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking c programs using f-soft. In *Computer Design : VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 297–308. IEEE, 2005.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5) :649–678, 2011.

[JKS+13] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards modeling and model checking fault-tolerant distributed algorithms. In *Model Checking Software*, pages 209–226. Springer, 2013.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs ? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681, 2013.

[JZ87] David B Johnson and Willy Zwaenepoel. *Sender-based message logging.* Rice University, Department of Computer Science, 1987.

[KAJV07] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition : Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 18–18, 2007.

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976.

[KK11] Manjit Kaur and Raj Kumari. Comparative study of automated testing tools : Testcomplete and quicktest pro. *International Journal of Computer Applications*, 24(1) :1–7, 2011.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, (1) :23–31, 1987.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2) :125–143, 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, July 1978.

[Lam83] Leslie Lamport. What good is temporal logic ? In *IFIP congress*, volume 83, pages 657–668, 1983.

[LV01] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Model Checking Software*, pages 80–102. Springer, 2001.

[Mat08] Aditya P Mathur. *Foundations of software testing*. 2008.

[McM04] Phil McMinn. Search-based software test data generation : a survey. *Software testing, Verification and reliability*, 14(2) :105–156, 2004.

[ML09] Mika V Mantyla and Casper Lassenius. What types of defects are really discovered in code reviews ? *Software Engineering, IEEE Transactions on*, 35(3) :430–448, 2009.

[MPC+02] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. Cmc : A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI) :75–88, 2002.

[MQ07] Madan Musuvathi and Shaz Qadeer. Chess : Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation*, pages 15–16. Springer, 2007.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. 2011.

[NCH+05] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured : type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3) :477–526, 2005.

[NK10] Anh Cuong Nguyen and Siau-Cheng Khoo. Towards automation of ltl verification for java pahtfinder. NUROP Congress, Singapore, 2010.

[NL11] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2) :11, 2011.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. 2002.

[Pel93] Doron Peled. All from one, one for all : on model checking using representatives. In *Computer Aided Verification*, pages 409–423. Springer, 1993.

[Pla14] David A Plaisted. Automated theorem proving. *Wiley Interdisciplinary Reviews : Cognitive Science*, 5(2) :115–128, 2014.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[PW97] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5) :243–246, 1997.

[PWW98] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and ω-regular languages. *Theoretical Computer Science*, 195(2) :183–203, 1998.

[QBC00] Francesco Quaglia, Roberto Baldoni, and Bruno Ciciani. On the no-z-cycle property in distributed executions. *Journal of Computer and System Sciences*, 61(3) :400–427, 2000.

[Rem05] Jason Remillard. Source code review systems. *Software, IEEE*, 22(1) :74–77, 2005.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *Computer Aided Verification*, pages 248–263. Springer, 2000.

[SBM+13] Habib Saissi, Péter Bokor, Can Arda Muftuoglu, Neeraj Suri, and Marco Serafini. Efficient verification of distributed protocols using stateful model checking. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 133–142. IEEE, 2013.

[SGB12] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. A sound reduction of persistent-sets for deadlock detection in mpi applications. In *Formal Methods : Foundations and Applications*, pages 194–209. Springer, 2012.

[Sha07] J Shattuck. Fuzz testing : explanation and useful tools, 2007.

[Sie07] Stephen F Siegel. Verifying parallel programs with mpi-spin. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14. Springer, 2007.

[SM14] Nastaran Shafiei and Peter Mehlitz. Extending jpf to verify distributed systems. *ACM SIGSOFT Software Engineering Notes*, 39(1) :1–5, 2014.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4) :149–160, 2001.

[SVW85] A Prasad Sistla, Moshe Y Vardi, and Pierre Wolper. The complementation problem for büchi automata with applications to temporal logic. In *Automata, Languages and Programming*, pages 465–474. Springer, 1985.

[SZ11] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of mpi-based parallel programs. *SIGPLAN Not.*, 2011.

[T+06] Heikki Tauriainen et al. *Automata and linear temporal logic : translations with transition-based acceptance*. Helsinki University of Technology, 2006.

[UL10] Mark Utting and Bruno Legeard. *Practical model-based testing : a tools approach*. 2010.

[VAG+10] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R De Supinski, Martin Schulz, and Greg Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10. IEEE, 2010.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, pages 491–515. Springer, 1991.

[VGK08] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification*, pages 66–79. Springer, 2008.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2) :203–232, 2003.

[VSGK08] Sarvani S Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M Kirby. Isp : a tool for model checking mpi programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 285–286. ACM, 2008.

[WVS83] Pierre Wolper, Moshe Y Vardi, and A Prasad Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 185–194. IEEE, 1983.

[YKKK09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball : Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification*, pages 385–398. Springer, 2008.

[ZWN+06] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4) :240–253, 2006.