Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**

UFR STMIA

Performance & Correctness Assessment of Distributed Systems

THÈSE

présentée et soutenue publiquement le 24/10/2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Cristian ROSA

Composition du jury

Directeurs :	Stephan MERZ Martin QUINSON
Président :	Isabelle CHRISMENT
Rapporteurs :	Ganesh GOPALAKRISHNAN Jean-François MÉHAUT
Examinateur :	Brigitte ROZOY

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

Loria

Mis en page avec la classe thloria.

a Erica, mi amor. . .

Résumé

Les systèmes distribués sont au cœur des technologies de l'information. Il est devenu classique de s'appuyer sur multiples unités distribuées pour améliorer la performance d'une application, la tolérance aux pannes, ou pour traiter problèmes dépassant les capacités d'une seule unité de traitement. La conception d'algorithmes adaptés au contexte distribué est particulièrement difficile en raison de l'asynchronisme et du non-déterminisme qui caractérisent ces systèmes. La simulation offre la possibilité d'étudier les performances des applications distribuées sans la complexité et le coût des plates-formes d'exécution réelles. Par ailleurs, le model checking permet d'évaluer la correction de ces systèmes de manière entièrement automatique.

Dans cette thèse, nous explorons l'idée d'intégrer au sein d'un même outil un model checker et un simulateur de systèmes distribués. Nous souhaitons ainsi pouvoir évaluer la performance et la correction des applications distribuées. Pour faire face au problème de l'explosion combinatoire des états, nous présentons un algorithme de réduction dynamique par ordre partiel (DPOR), qui effectue une exploration basée sur un ensemble réduit de primitives de réseau. Cette approche permet de vérifier les programmes écrits avec n'importe laquelle des interfaces de communication proposées par le simulateur. Nous avons pour cela développé une spécification formelle complète de la sémantique de ces primitives réseau qui permet de raisonner sur l'indépendance des actions de communication nécessaire à la DPOR. Nous montrons au travers de résultats expérimentaux que notre approche est capable de traiter des programmes C non triviaux et non modifiés, écrits pour le simulateur SimGrid.

Par ailleurs, nous proposons une solution au problème du passage à l'échelle des simulations limitées pour le CPU, ce qui permet d'envisager la simulation d'applications pair-à-pair comportant plusieurs millions de nœuds. Contrairement aux approches classiques de parallélisation, nous proposons une parallélisation des étapes internes de la simulation, tout en gardant l'ensemble du processus séquentiel. Nous présentons une analyse de la complexité de l'algorithme de simulation parallèle, et nous la comparons à l'algorithme classique séquentiel pour obtenir un critère qui caractérise les situations où un gain de performances peut être attendu avec notre approche. Un résultat important est l'observation de la relation entre la précision numérique des modèles utilisés pour

simuler les ressources matérielles, avec le degré potentiel de parallélisation atteignables avec cette approche. Nous présentons plusieurs cas d'étude bénéficiant de la simulation parallèle, et nous détaillons les résultats d'une simulation à une échelle sans précédent du protocole pair-à-pair Chord avec deux millions de nœuds, exécutée sur une seule machine avec un modèle précis du réseau.

Mots-clés: model, checking, simulation, distribuées, verification, parallèlisation.

Abstract

Distributed systems are in the mainstream of information technology. It has become standard to rely on multiple distributed units to improve the performance of the application, help tolerate component failures, or handle problems too large to fit in a single processing unit. The design of algorithms adapted to the distributed context is particularly difficult due to the asynchrony and the nondeterminism that characterize distributed systems. Simulation offers the ability to study the performance of distributed applications without the complexity and cost of the real execution platforms. On the other hand, model checking allows to assess the correctness of such systems in a fully automatic manner.

In this thesis, we explore the idea of integrating a model checker with a simulator for distributed systems in a single framework to gain performance and correctness assessment capabilities. To deal with the state explosion problem, we present a dynamic partial order reduction algorithm that performs the exploration based on a reduced set of networking primitives, that allows to verify programs written for any of the communication APIs offered by the simulator. This is only possible after the development of a full formal specification with the semantics of these networking primitives, that allows to reason about the independency of the communication actions as required by the DPOR algorithm. We show through experimental results that our approach is capable of dealing with non trivial unmodified C programs written for the SimGrid simulator.

Moreover, we propose a solution to the problem of scalability for CPU bound simulations, envisioning the simulation of Peer-to-Peer applications with millions of participating nodes. Contrary to classical parallelization approaches, we propose parallelizing some internal steps of the simulation, while keeping the whole process sequential. We present a complexity analysis of the simulation algorithm, and we compare it to the classical sequential algorithm to obtain a criteria that describes in what situations a speed up

can be expected. An important result is the observation of the relation between the precision of the models used to simulate the hardware resources, and the potential degree of parallelization attainable with this approach. We present several case studies that benefit from the parallel simulation, and we show the results of a simulation at unprecedented scale of the Chord Peer-to-Peer protocol with two millions nodes executed in a single machine.

Keywords: model, checking, simulation, distributed, verification, parallelization

Remerciements

First of all I would like to thank Stephan and Martin who gave me the chance to work with them. They are brillant people who guided me wisely during all these 3 years, and that going beyond their duty, they helped us with our staying in France that otherwise wouldn't have been possible. I would also like to thank to all our friends that were also our family, their support was inspiring and part of this work is due to them.

Éste párrafo va en español, para mis queridos hispanoparlantes. En primer lugar quiero agradecer a mi amada esposa Erica, que dejó una gran parte de su vida de lado para acompañarme en estos 3 años a puro sacrificio, muchas gracias por el aguante, no lo voy a olvidar nunca. También quiero agradecer a mis padres, que a pesar de la distancia estuvieron siempre presentes dándonos todo el apoyo y la ayuda que necesitabamos para seguir adelante. Finalmente, quiero agradecer a todos los que no he mencionado pero que igualmente hicieron posible este logro, que si bien es personal, fue gracias a su motivación y apoyo.

Table of Contents

Table des figures

Résumé étendu

1 Résumé de l'introduction

1.1 Contexte scientifique

Les systèmes distribués sont au cœur des systèmes d'information. Il est devenu classique de s'appuyer sur de multiples unités distribuées qui, collectivement, contribuent à une application commerciale ou scientifique. Avoir plusieurs unités distribuées qui travaillent simultanément sur plusieurs parties d'un problème présente de nombreux avantages potentiels : amélioration des performances, meilleure tolérance aux défaillances des composants, ou capacité à résoudre des problèmes dont les dimensions interdisent leur résolution par une seule unité de traitement.

La massification de leur usage impose de mieux comprendre comment ces plates-formes se comportent. Cependant, l'hétérogénéité des systèmes distribués et leur échelle posent de sérieux défis méthodologiques quant à leur analyse. Le large spectre des architectures et des applications explique que différentes notions de performances coexistent, ce qui complique encore l'analyse. Par exemple, la qualité des systèmes de calcul à haute performance (HPC) est évaluée par le temps nécessaire pour réaliser tous les calculs assignés au système (makespan) tandis que les systèmes pair-à-pair (P2P) sont évalués uniquement par le décompte des messages échangés en vue d'atteindre l'objectif fixé. Les systèmes de calcul dans les nuages (Clouds) introduisent une notion de performance encore différente, basée sur des variables économiques comme le coût d'exécution du programme sur la plate-forme louée pour l'occasion.

Les algorithmes distribués posent par ailleurs des défis spécifiques mais communs à tous leurs domaines d'application. La nature distribuée des plates-formes utilisées augmente grandement le risque de panne d'une machine distante ou de perte de message, problèmes pour lesquels il n'existe pas de méthode de détection parfaite. Ces facteurs externes, hors du contrôle du programmeur mais dont l'occurrence conditionnent le com-

portement du système compliquent la mise au point de systèmes distribués.

De plus, la distribution empêche également d'avoir une vue complète de l'état du système puisque les processus s'exécutent dans leur propre espace mémoire. Chacun n'a qu'une vision potentiellement obsolète du système dans sa globalité. L'obtention d'une vue cohérente du système, par exemple pour permettre la reprise sur erreur, constitue un problème très difficile ayant donné lieu à une étude très fournie dans la littérature.

Une autre difficulté inhérente aux systèmes distribués est l'absence d'horloge centralisée, ce qui complique fondamentalement la synchronisation entre les processus ou l'ordonnancement d'événements ayant eu lieu sur des machines différentes.

De manière générale, l'asynchronie et la vue seulement partielle du système sont la source de nombreuses difficultés pour les concepteurs. De plus, la nature éminemment parallèle des systèmes distribués fait qu'ils présentent également les difficultés classiques des systèmes multi-threadés centralisés, telles que les conditions de compétitions (race condition), les interblocages (dead-locks) ainsi que les famines (live-locks).

1.2 Motivation

Avec la complexité grandissante des programmes distribués, il devient nécessaire développer de nouvelles méthodologies et outils pour aider le développeur à mieux comprendre le comportement de ces systèmes. Malgré leurs différences apparentes, tous les systèmes distribués peuvent être considérés comme des ensembles de processus indépendants dotés d'un état local privé, et interagissant par échanges de messages. Cette abstraction permet d'utiliser des technique comparables pour étudier n'importe lequel de ces systèmes.

Il est possible de grouper les différentes propriétés observables d'un système distribué en deux catégories distinctes : les propriétés liées à la performance et celles concernant la correction.

Pour évaluer les performances des systèmes distribués, la simulation constitue une approche classique offrant la possibilité d'étudier l'application au travers d'une plateforme simulée sans imposer l'usage du matériel réel. Le code s'exécute dans un environnement contrôlé, offrant plus de simplicité et de confort expérimental que les systèmes réels. Cela permet de plus d'analyser les performances du système sur des plates-formes différentes de celles auxquelles l'expérimentateur a réellement accès, permettant ainsi des campagnes de tests plus complètes. L'un de ses défaut est l'absence de certains biais expérimentaux réels, tels que le bruit de fond d'une liaison réseau. Il est cependant

généralement possible de régler la précision des modèles décrivant les ressources pour les adapter aux besoins des utilisateurs.

En ce qui concerne la vérification de la correction des systèmes distribués, plusieurs techniques de vérification formelle ont gagné du terrain ces dernières années. La correction d'un système est alors assurée par rapport à la spécification (constituée d'un ensemble de propriétés) pour lequel il doit être vérifié. Le model checking est une technique visant à établir si un modèle répond à sa spécification. Une caractéristique intéressante du model checking est sa nature totalement automatique, qui n'impose pas d'expertise spécifique pour être mise en œuvre, au contraire par exemple des assistants de preuve, accessibles aux seuls experts. Un model checker de propriétés de sûreté fonctionne en explorant l'espace d'état du modèle à la recherche d'états non valides, c'est à dire ne répondant pas aux spécifications. La recherche progresse jusqu'à la découverte d'un état qui violant une propriété de correction, ou jusqu'à l'exploration complète de l'espace d'état, ou encore jusqu'à l'épuisement des ressources allouées au model checker (comme la mémoire, ou le temps disponible pour réaliser l'étude). Quand un état invalide est découvert, le model checker fournit un contre-exemple qui consiste en la trace d'exécution qui y conduit. Ce caractère exhaustif fait du model checking une technique adaptée à la recherche de bugs qui auraient échappé aux tests traditionnels, qui négligent les cas limites. La limite majeure de cette approche est due au fait qu'il n'est pas toujours possible d'atteindre une conclusion à cause de la taille potentiellement exponentielle de l'espace d'état.

Du point de vue de l'utilisateur, simulation et vérification dynamique offrent des avantages méthodologiques similaire : elles sont à la fois complètement automatique et relativement simple à utiliser. Le type des propriétés qu'elles permettent d'étudier les rend également parfaitement complémentaire. La simulation est parfaitement adaptée à l'étude des performances. Concernant l'étude du comportement de l'application, la simulation permet d'étudier des scénarios de très grande taille, mais uniquement pour un ensemble de situation données. Par ailleurs, remonter des phénomènes observés sur la plate-forme à leurs causes dans l'algorithme peut s'avérer difficile. De son coté, le model checking permet évaluer la correction des systèmes de manière exhaustive, mais s'avère limité à de petites instances du problème pour éviter l'explosion de l'espace d'état. Les contre-exemples générés s'avèrent précieux pour découvrir les causalités ayant menées à des violations de propriétés.

Malgré la relation étroite entre la simulation et la vérification dynamique, les com-

munautés de recherche et les outils disponibles interagissent rarement. Un développeur désireux d'utiliser un simulateur et un model checker est souvent obligé d'écrire plusieurs prototypes, adaptés à chaque outil. Cette multiplication des modèles et des formalismes nécessaire pour étudier une application donnée complique le processus et augmente les coûts de développement. Par ailleurs, il n'existe aucune garantie que l'application réelle se comporte conformément aux modèles si la traduction d'un formalisme dans un autre se fait de manière manuelle. Nous pensons qu'une approche attractive pour combler l'écart entre la simulation et le model checking est d'unifier les deux méthodologies dans un même cadre. Un tel outil unifié peut grandement simplifier le développement de systèmes distribués corrects, en éliminant le coût de l'écriture des modèles multiples pour la même application.

1.3 Objectifs de la Thèse

L'objectif général de cette thèse est développer la théorie et les outils requis pour fournir un cadre unifié d'évaluation des aspects de la performance ainsi que des propriétés de correction directement sur programmes distribués exécutables.

Pour parvenir à une telle solution, cette thèse développe deux idées majeures :

1. Du côté de la correction, nous explorons l'idée d'intégrer un simulateur et un model checker dans un outil unique. Nous pensons que disposer des deux fonctionnalités au sein du même environnement facilite et encourage l'utilisation de ces techniques complémentaires à tout moment du processus de développement. Le travail de mise en œuvre est basée sur l'outil de simulation SimGrid. Nous proposons tout d'abord une analyse de l'architecture actuelle de SimGrid à la lumière des fonctionnalités à intégrer en vue de la vérification dynamique. Une fois les différences architecturales explicitées, nous présentons un travail de *refactoring* pour simplifier l'intégration du model checker. Le principal défi de ce travail est de limiter autant que possible l'impact de ces modifications sur les performances des simulations.

   Le succès du model checker repose sur sa capacité à faire face au nombre énorme d'ordonnancement possible des événements générés par les programmes distribués. En effet, l'entrelacement des histoires locales pour former un séquencement global des événements donne lieu à une explosion combinatoire. Généralement, beaucoup de ces entrelacements sont équivalents au sens où ils conduisent à des états globaux indiscernables. Pour attaquer ce problème, nous explorons une nouvelle

architecture permettant d'appliquer des techniques de réduction de l'espace d'état de façon générique, indépendante du schéma de communication de l'application. Enfin, nous validons nos résultats avec plusieurs cas de études en utilisant de petits programmes dans lesquels des bugs ont été intentionnellement ajoutés, mais également en utilisant des exemples plus réalistes.

2. À propos de la correction, nous visons des simulations P2P rapides, comportant des millions de processus s'exécutant sur des plates-formes de type Internet. À cette fin, nous essayons d'accélérer les simulations qui sont limités par le CPU, en exploitant la puissance des machines multi-core classiques. Pour cela, nous explorons l'idée de paralléliser l'exécution de la application pendant la simulation. Nous avons comparons d'abord les coûts de calcul de l'approche parallèle contre l'exécution séquentielle classique afin de comprendre dans quels scénarios une amélioration de la vitesse de simulation peut être attendu de ce parallélisme. Cette meilleure compréhension de l'exécution parallèle acquise, nous nous concentrons sur l'optimisation des performances du simulateur. Ensuite, nous proposons une validation expérimentale de l'approche avec plusieurs cas de études couvrant différents types de systèmes distribués (HPC et Grid). Afin de mesurer l'extensibilité et les performances de SimGrid sur des scénarios P2P, nous le comparons à un autre simulateur bien connu du domaine.

1.4 Structure la thèse

Ce manuscrit de thèse est organisé de la manière suivante :

Le chapitre 1 (non traduit en français) présente l'état de l'art sur les sujets principaux abordés dans ce travail. Cela comprend une introduction générale aux systèmes distribués (l'objet d'étude principal de cette thèse) ; une vue d'ensemble du domaine de la simulation, en particulier de la simulation de systèmes distribués ; une vue d'ensemble du model checking, et de la vérification dynamique de logicielle ; une description détaillée de SimGrid, qui constitue le contexte technique de nos travaux.

Le chapitre 2 introduit la première contribution de cette thèse, qui consiste en la réunion de la simulation et du model checking dans un environnement unique. Nous étudions les difficultés posées par l'architecture précédente de SimGridlors de l'ajout de fonctionnalité de model checking et de parallélisation de la boucle principale de simula-

tion. Nous présentons également un nouveau design, inspiré de concepts empruntés aux systèmes d'exploitation. Il sert de base au reste des travaux présentés dans les chapitres suivants.

Le chapitre 3 présente SimGridMC, le model checker intégré à l'environnement de simulation SimGrid. Il constitue la seconde contribution de cette thèse. Nous discutons les choix de design ainsi que les difficultés d'implémentation, puis nous décrivons en détail les techniques mises en œuvre pour luter contre l'explosion combinatoire de l'espace d'état. De plus, nous présentent divers résultats expérimentaux montrant l'efficacité de l'approche.

Le chapitre 4 s'attaque au problème des simulations dont le facteur limitant est la puissance de calcul en parallélisant une partie de la simulation. Cela constitue la troisième contribution de cette thèse. Nous détaillons l'architecture de la version parallèle de la boucle de simulation, et nous comparons sa complexité à celle de la version séquentielle. Nous présentons également le travail d'optimisation mis en œuvre pour réduire le surcoût dû au parallélisme à son minimum. Enfin, nous présentons des résultats expérimentaux justifiant l'analyse de complexité et montrant l'extensibilité de SimGridvis-à-vis des grands scénarios simulés, ainsi que les bénéfices du parallélisme dans ce contexte.

2 Résumé du chapitre 2 : réunir simulation et vérification efficaces

La première étape vers un cadre unifié pour l'étude des performances et de la correction d'un système distribué est d'analyser l'architecture préexistante de SimGridau regard des nouvelles exigences introduites par le model checker et par la parallélisation du simulateur. Dans ce chapitre, nous montrons qu'avec l'architecture actuelle, il est presque impossible de mettre en œuvre les nouvelles fonctionnalités de manière appropriée et efficace. Le problème principal de l'architecture SimGridréside dans la façon dont l'état de la simulation est manipulé, qui n'est pas compatible avec les nouvelles exigences. Par conséquent, nous proposons un nouveau design, qui part de l'observation que les services offerts par la couche virtualisation de SimGridsont similaires à ceux fournies par un système d'exploitation (OS). Cette constatation permet de concevoir un système à même d'exécuter en parallèle les processus utilisateur pendant les simulations, et per-

mettant des explorations efficaces du comportement non déterministe des programmes par model checking.

2.1 L'importance de l'état partagé

L'état partagé d'un système est une notion centrale à la fois pour l'exploration de l'espace d'état ainsi que pour l'exécution parallèle de la simulation.

Le model checker génère systématiquement toutes les exécutions possibles du programme pour vérifier le respect des propriétés à vérifier dans tous les entrelacements possibles des processus du système. Pour ce faire, il doit contrôler l'état du réseau, qui constitue le seul état partagé entre les processus. Comme le montre la figure 2.1, page 70, l'ordre dans lequel les processus effectuent les communications, en affectant l'état du réseau partagé, induit une trace donnée, correspondant à un ordre partiel particulier entre les événements. Pour pouvoir piloter l'exécution du programme à volonté, un model checker doit donc disposer d'un mécanisme d'interception des communications avant qu'elles ne modifient l'état du réseau et les retarder à après l'expression des communications prévues par tous les processus.

L'état partagé est également important pour l'exécution parallèle des processus de l'utilisateur pendant la simulation. Ces processus peuvent interagir avec la plate-forme simulée à n'importe quel point de leur exécution, ce qui implique la modification de portions partagée de l'état de la simulation. Par conséquent, l'exécution parallèle des processus nécessite que toutes les opérations sur l'état partagé soient atomiques afin de maintenir la cohérence des simulations. Par ailleurs, ces opérations doivent toujours se produire dans le même ordre afin de garantir la reproductibilité des simulations. Dans le cas contraire, des entrelacements différents des actions au niveau du simulateur pourrait changer la correspondance de la source et destination des messages et donc conduire à exécutions différentes de la simulation.

2.2 État partagé et autres considérations dans SimGrid

Le principal inconvénient de la conception de SimGridpour répondre aux nouvelles exigences est l'absence d'encapsulation de l'état partagé de la simulation, qui est dispersée à travers toute la pile logicielle et donc difficile à contrôler et à manipuler.

État partagé dispersé Les structures de données composant l'état partagé de la simulation, en particulier l'état du réseau, sont dispersées dans toute la pile logicielle, comme

illustré dans la figure 1.5, page 66. L'état de la couche applicative du réseau est contenue dans les API de communication ; Le module SIMIX contient l'état des processus impliqués dans une action du réseau, les primitives de synchronisation, et l'association des actions entre d'une part les variables de condition bloquant les processus et d'autre part le noyau de simulation. Enfin, l'état des ressources, comme les processeurs ou l'état de la couche transport du réseau est géré par SURF. Dans le cas de la simulation séquentielle, cela ne pose pas de problème car à tout moment au plus un seul processus utilisateur s'exécute, et il n'y a donc aucun risque de conditions de course. Cependant, cette conception pose de sérieuses difficultés pour satisfaire à la condition d'atomicité de l'exécution parallèle. Le problème peut être clairement visualisée dans la Figure 1.4, page 65. Les contextes d'exécution du code utilisateur exécutent également les fonctions de l'API de communication (flèches USER + API + SIMIX) qui affectent les données partagées. Si nous permettons l'exécution concurrente de ces contextes, il faudrait un schéma de verrouillage traversant toute la pile logicielle afin d'éviter des conditions de compétition et maintenir la cohérence. Cela serait à la fois extrêmement difficile à réaliser correctement, et probablement prohibitif en termes de performances. Enfin, même si cette difficulté était résolue afin d'assurer la cohérence interne du simulateur, une exécution parallèle suivant ce schéma s'avérerait non reproductible puisque l'ordonnancement des exécutions pourrait varier entre les simulations, menant probablement à résultats différentes en chaque simulation d'un même système.

Absence de contrôle sur les exécutions. Un model checker doit générer systématiquement toutes les exécutions possibles du système. Cela nécessite un mécanisme d'interception des transition capable de bloquer les processus avant qu'ils ne modifient l'état du réseau. Cela permet de différer les décisions de mise en correspondance des sources et destinations des messages jusqu'à après la déclaration des intentions de chaque processus. Ce mécanisme est indispensable afin de pouvoir considérer toutes possibilités. La principale difficulté lors de la mis en œuvre d'un tel mécanisme dans SimGridse révèle être à nouveau l'absence de contrôle adéquat sur l'état du réseau. Le but d'un simulateur est de bloquer les processus impliqués dans une communication jusqu'au moment où le noyau de simulation détermine que le temps simulé correspondant est écoulé. Par conséquent, le mécanisme d'interception ne fournit qu'un moyen de retarder les processus, mais pas de modifier la correspondance entre les sources et destinations de messages, qui est plutôt déterminé par l'ordre naturel de l'émission des opérations réseau.

Code spécifique à chaque interface. SimGridpropose différentes interfaces de communication, adaptées à différents types d'applications. Malgré leur apparente diversité, l'implémentation de ces interfaces s'avère très similaire. Elles peuvent toutes être réduites à une fonction calculant l'appariement entre source et destination des messages, une variable de condition par communication pour bloquer les processus impliqués, ainsi la création et la destruction d'actions correspondantes dans le cœur de la simulation. Dans la version préexistante de SimGrid, ces interfaces ne partageait cependant aucun code, et chacune était implémentée séparément, ce qui impliquait une base de code importante et difficile à maintenir. De plus, un model checker réalisé pour l'une des interfaces devrait être entièrement réécrit pour être adapté à une autre interface.

2.3 SIMIX v2.0

Dans cette section, nous introduisons la version deux de SIMIX dont l'objectif est de simplifier l'implémentation de la simulation parallèle et du model checking au sein de SimGrid. SIMIX v2.0vise à encapsuler l'état partagé de la simulation et assurer que toute modification à cet état s'effectuera sans risque de condition de compétition. Le nouveau design de SIMIX part de la constatation que les services offerts au code utilisateur par la simulation sont très similaires à ceux offerts par un système d'exploitation classique : la notion de processus, de communication inter-processus, et de synchronisation entre processus. Il semble donc raisonnable d'encapsuler ces notions ensemble afin de répondre aux problèmes mentionnés plus haut.

Un design par couche plus strict. Le découpage en modules présenté dans la figure 1.3, page 64 n'est qu'un découpage fonctionnel et toutes les interfaces de tous les modules sont publiques (sauf vis-à-vis du code utilisateur). Il en résulte une grande quantité d'appel inter-couches qui violent le découpage. SIMIX v2.0introduit un découpage plus strict qui masque entièrement les couches basses de la simulation au regard de l'implémentation des interfaces utilisateurs de haut niveau. SIMIX v2.0agit comme un système d'exploitation virtuel et offre des notion de processus, de synchronisation et de communications inter-processus (IPC). Si les deux premières notions étaient déjà représentées dans la version préexistante de SIMIX, la prise en compte du rôle central des IPC est une nouveauté primordiale ici.

Nous donnons maintenant une vue d'ensemble des mécanismes d'IPC offerts, dont la sémantique formelle est donnée à la section 3.9. Le premier concept introduit par

SIMIX v2.0en matière d'IPC est celui de *boîte aux lettres*, qui servent de point de rendez-vous entre les processus souhaitant communiquer. Elles ont une existence propre et n'importe quel processus peut accéder à n'importe quelle boîte aux lettres à tout moment. L'interface n'offre que les quatre fonctions *Send*, *Recv*, *TestAny* et *WaitAny*. Les deux premières permettent d'ajouter une requête d'émission ou de réception dans une boite aux lettres donnée. *TestAny* permet de tester si au moins une des communications d'un ensemble est terminée tandis que *WaitAny* permet de bloquer jusqu'à la terminaison de l'une des communications d'un ensemble donné. Les requêtes de communications placées dans une boîte aux lettres particulière ne sont appariées que si leurs critères de filtrage correspondent. Par défaut tout envoi est apparié à tout réception, mais l'utilisateur peut utiliser des critères plus précis au besoin.

Malgré son apparente simplicité, ce modèle de communication est suffisant pour implémenter toutes les interfaces offertes aux utilisateurs par SimGrid. Le concept de boîte aux lettres permet des schémas de communication de groupe de type many-to-many, et les interfaces hautes peuvent renforcer les critères d'appariement pour implémenter des schémas de communication many-to-one ou one-to-many. De plus, les fonctions *Send* et *Recv* sont seulement impliquées dans l'appariement des émetteurs et récepteurs tandis que *TestAny* et *WaitAny* sont en charge de la copie effective des données échangées. Ce découpage précis des responsabilités donne un contrôle fin de l'état partagé du réseau sans compromettre la puissance sémantique. Cela a permis d'exprimer toutes les interfaces existantes de SimGrid : MSG (de type CSP), GRAS (basée sur les sockets) et un sous-ensemble de MPI. L'encapsulation de l'état partagé ainsi offerte simplifie l'écriture du model checker, qui n'a plus qu'à interagir avec un seul module. Ce nouveau fonctionnement permet également de supprimer toute duplication du code lié aux IPC, et de simplifier fortement le code des synchronisations.

Émulation du mode noyau. Après avoir assuré une meilleure encapsulation des données par une séparation par couche plus stricte, nous présentons maintenant notre proposition pour permettre des modifications concurrentes de l'état partagé. Elle s'inspire des systèmes d'exploitation modernes où les processus s'exécutent dans un espace d'adressage virtuel, séparé des autres. Pour communiquer, les processus demandent l'intervention du noyau qui réalise (après médiation) les requêtes des processus en utilisant un mode d'exécution particulier. Ce découpage clair entre le mode utilisateur et le mode noyau permet l'exécution indépendante et parallèle des processus puisque tout accès aux ressources partagées est protégé par le noyau, qui en assure ainsi la cohérence.

En suivant le même modèle, les processus simulés dans SimGridinteragissent au travers de *requêtes* qui sont traitées par un contexte d'exécution particulier nommé *maestro*. Comme l'état partagé ne peut être modifié que par le maestro, aucun risque de compétition n'est à craindre dans les interactions entre les processus et leur environnement.

L'algorithme 3 (page 78) présente la nouvelle boucle principale de simulation. La majeure différence avec celui préexistant (algorithme 2, page 65) est que les interactions des processus avec leur environnement ne sont pas réalisées immédiatement, mais retardées jusqu'à la ligne 5 et la fonction *handle_request()*, en charge de leur traitement.

Cette modification, bénigne en apparence, s'avère particulièrement intéressante lors de l'implémentation de la simulation parallèle et du model checking. Les requêtes constituent un point d'interception potentiel permettant de contrôler toute modification de l'état partagé, tandis que le traitement séquentiel des requêtes permet d'éviter les conditions de compétition dans les accès aux ressources sans nécessiter de verrouillage fin des données.

Contextes d'exécution systèmes efficaces. Nous avons vu que les processus simulés s'exécutent au sein de contextes d'exécution systèmes différents pour permettre leur séparation. Ce mécanisme peut par exemple être implémenté en utilisant les `ucontextes` spécifiés par POSIX. Les changements de contexte sont alors réalisés par la fonction `swapcontext`, offerte par le système. Bien qu'entièrement réalisée en espace utilisateur en apparence, cette fonction recèle pourtant un appel système pour permettre d'avoir un masque de signaux par contexte d'exécution. Comme cette fonctionnalité n'est pas pertinente dans notre contexte, nous avons réimplémenté directement en assembleur une version des contextes exempte de tout appel système.

Les modifications présentées dans ce chapitre ont naturellement un impact sur les performances de la simulation dans le cas standard où ni le model checking ni la simulation parallèle ne sont activés. En effet, augmenter l'encapsulation résulte classiquement en un accroissement des temps d'accès aux données. Pourtant, les résultats expérimentaux (détaillés dans la section 2.6) montrent que la nouvelle version séquentielle est plus rapide d'environ 10% par rapport à la version préexistante de SimGrid. Ce résultat légèrement contre-intuitif à première vue s'explique d'une part par la simplification du code d'IPC rendue possible par SIMIX v2.0 et d'autre part par l'efficacité des nouveaux contextes d'exécution débarrassés de tout appel système.

3 Résumé du chapitre 3 : vérification dynamique de systèmes distribués

Dans ce chapitre, nous présentons SimGridMC, un model checker de systèmes distribués intégré à l'environnement SimGrid. Cela permet aux utilisateurs d'étudier la *correction* de leurs systèmes en plus des études sur leurs *performances* permises classiquement par le simulateur. L'objectif principal de cette intégration est la simplicité d'usage : notre outil permet de vérifier dynamiquement les programmes écrits pour la simulation *sans aucune modification* par exploration exhaustive de l'espace d'états du programme.

Nous nous attachons tout d'abord à donner une vue d'ensemble de SimGridMC et montrer comment il s'intègre à l'environnement SimGrid. Suivent une description des modèles et propriétés que SimGridMC peut prendre en compte. Nous discutons ensuite les décisions de conception du mécanisme d'exploration de trace d'exécution, et comment il s'intègre au reste de l'environnement. Nous présentons enfin notre solution au problème de l'explosion combinatoire de l'espace d'états, basée sur la réduction dynamique par ordre partiel (DPOR). L'efficacité de cette méthode dépend grandement de la capacité à déterminer si deux transitions sont dépendantes ou non. Nous introduisons donc une formalisation en TLA$^+$ des transitions possibles dans SimGrid, et nous en dérivons un prédicat d'(in)dépendance. En conclusion, nous présentons des résultats expérimentaux montrant notre capacité à appliquer la méthode de DPOR à des systèmes distribués exprimés dans des interfaces différentes grâce à la couche intermédiaire introduite au chapitre précédent.

3.1 Vérification dynamique dans SimGrid

SimGridMC est un model checker permettant la vérification dynamique de systèmes distribués exprimés sous forme de programmes pour le simulateur SimGrid. Cela permet d'éviter la construction d'un modèle du programme compatible avec un model checker classique, étape souvent difficile et fastidieuse. Il s'agit donc en quelque sorte d'un *model checker sans modèle*, ou plutôt où le modèle n'est pas connu explicitement. Cette approche souffre cependant de certaines limitations par rapport au model checking classique, comme le fait qu'elle ne permet que de vérifier des systèmes finis alors que l'étude classique des systèmes distribués se fait au travers de systèmes comptant une infinité de transitions. Notre approche reste justifiée dans le cadre d'une utilisation en tant que debugguer. Nous considérons en effet qu'une séance de vérification dynamique est fructueuse

si nous trouvons un contre-exemple pour les propriétés annoncées.

D'un point de vue conceptuel, SimGridMC prend en entrée un programme SimGrid classique, avec des assertions ajoutées dans le code pour exprimer les propriétés de sûreté. Il mène alors une exploration exhaustive de l'espace d'état pour vérifier le respect des assertions dans tous les cas. Au regard de l'architecture de SimGrid, le model checker vient se placer en remplacement du module SURF, habituellement en charge du noyau de simulation. Les autres modules (SIMIX et les interfaces utilisateurs) restent inchangés. Le travail de refactoring introduit dans le chapitre 2 simplifie grandement cette intégration technique puisque l'environnement comprend déjà l'infrastructure pour influer sur l'ordonnancement des processus et pour intercepter les communications modifiant l'état partagé. La difficulté majeure consiste alors à générer tous les ordonnancements d'événements possibles au lieu de déterminer par simulation celui qui aura lieu sur la plate-forme utilisée.

Le modèle. Comme explicité dans l'introduction, notre modèle se composes d'entités autonomes interagissant par échange de messages uniquement (pas d'état global, ni d'horloge globale). L'état du système est donné par l'état local de chaque processus et l'état du réseau. Le comportement du modèle peut être décrit par un graphe d'états où les nœuds représentent différents états et les arrêtes représentent des transitions d'un état vers un autre. Dans notre cas, ces transitions correspondent aux échanges de messages entre processus.

Dans SimGrid, l'état de chaque processus est donné par ses registres sur le processeur, sa pile d'exécution et la mémoire allouée dans le tas. L'état du réseau contient les messages en transit ainsi que les requêtes de communication postées dans les boîtes aux lettres existantes. Seules les requêtes de communications influant sur l'état partagé du système, ce sont les seuls transitions considérées par notre model checker, et les changements locaux ne sont donc pas considérés. Au lieu de cela, ces changements locaux sont agrégés à la transition globale à laquelle ils répondent. On considère donc comme transition un échange de message et les réactions locales des processus impliqués. L'espace d'état est généré en calculant tous les ordonnancements possibles de transitions des processus. Cet espace est souvent infini, même pour un nombre fini de processus, à cause des libertés offertes par la gestion de la mémoire et par le traitement des processus. La figure 3.2 (page 86) présente l'architecture de SimGridMC, qui intercepte toutes les interactions processus avec le réseau au travers de l'interface de communication. Les zones en bleu représentent l'état de système étudié tandis que les zones en rouge représentent

l'état du model checker, qui contient principalement la pile d'exploration.

Les propriétés. Elles décrivent le comportement attendu du système. À l'heure actuelle, SimGridMC ne gère que les propriétés de sûreté locales, sous forme d'assertions insérées dans le code du programme. Il y a donc deux limitations majeures aux propriétés qu'il est possible d'exprimer dans SimGridMC. D'une part, il n'est pas possible d'exprimer de propriétés de vivacité qui mettrait en relation diverses étapes de l'histoire du système. D'autre part, il n'est pas possible d'exprimer d'assertion liant l'état de plusieurs processus. Ces deux limitations sont nécessaires pour assurer le bon fonctionnement de l'algorithme de réduction de l'espace d'état présenté à la section 3.7. Il est cependant possible de contourner en partie ces limitations en pratique, et la vérification d'absence de situation d'interblocage est par exemple possible dans ce cadre.

Exploration explicite. L'état global du système est extrêmement complexe à analyser dans notre cas, car il contient la pile d'exécution des processus (dont l'analyse devrait se faire au niveau système et assembleur) et car les transitions sont exprimées sous forme d'un code C arbitraire à exécuter. Le fait que le modèle soit ainsi implicite rend son étude symbolique quasi impossible. Au lieu de cela, SimGridMC mène une exploration explicite de ce modèle en exécutant le code des transitions.

Lors de cette exploration, le model checker construit une pile dont chaque entrée représente un ensemble de processus dont il faut tester tous les ordonnancements, un ensemble de processus dont les ordonnancements ont déjà été testés, et la transition effectuée pour quitter cet état. L'exploration est alors menée en profondeur d'abord, en sélectionnant une transition possible à chaque étape, puis en revenant au début de l'histoire pour explorer une autre branche lorsqu'un état final est accepté. Comme l'histoire du système peut être infinie, le processus est borné à une profondeur maximale déterminée par l'utilisateur. Cela signifie naturellement que SimGridMC ne peut détecter les erreurs ayant lieu après la limite de la recherche, mais il assure une exploration exhaustive jusqu'à ce niveau.

La figure 3.3 (page 87) illustre le principe de l'exploration de SimGridMC. Le model checker commence par une phase d'initialisation où il exécute tous les processus jusqu'à leur première action de communication (exclue). Dans l'exemple, il s'agit des flèches descendantes vertes a et b. L'état résultant S_0 est l'état initial du système, et donc poussé sur la pile d'exploration avec deux transitions possibles a et b. De plus, une sauvegarde par snapshot système est réalisée afin de pouvoir restaurer cet état du système étudié

par la suite.

Une fois cette initialisation réalisée, le model checker sélectionne et exécute l'une des transitions possibles (dans l'exemple, a). Il effectue les changements à l'état global demandés par a puis ordonnance l'exécution du processus correspondant jusqu'à leur première action de communication, exclue (ici, c). Cette exécution correspond à une transition telle que considérée par le model checker. Ensuite, le mécanisme est itéré par l'exécution de l'une des transitions possibles en S_1 (ici, b). Ce mécanisme est itéré jusqu'à ce que la limite de profondeur de l'exploration soit atteinte ou bien jusqu'à ce qu'il n'y ait plus de transition à explorer. Cette dernière situation correspond soit à une terminaison du programme ou à une situation d'interblocage.

Sauvegarde et restauration d'états. Lorsque l'exploration atteint la fin de l'histoire du système, il est nécessaire de revenir à un état antérieur du système afin d'explorer une autre branche d'exécution. Une solution serait de sauvegarder tous les états rencontrés, mais il est probable qu'elle s'avèrerait trop coûteuse en temps et en espace dans notre cas. Au lieu de cela, nous avons choisi l'approche state-less introduite par Verisoft [26]. Elle consiste à ne sauvegarder que l'état initial, et à retourner dans un état quelconque du système en revenant à cet état initial puis en re-exécutant la trajectoire menant à l'état auquel on souhaite se ramener. Cette approche évite la consommation mémoire due au stockage de tous les états rencontrés, mais en contrepartie, elle ne permet pas la détection des cycles, et peut donc mener à réexplorer des sections de l'histoire qui avaient déjà été explorés. Cette limitation n'est pas critique dans notre contexte puisque nous menons une exploration bornée pour vérifier des propriétés de sûreté, mais elle s'avèrerait critique pour la vérification de propriétés de vivacité où la prise en compte des cycles est primordiale.

3.2 Limiter l'explosion combinatoire de l'espace d'états

La taille énorme de l'espace d'états à explorer lors de la vérification dynamique du système impose la mise en place de réduction pour rendre le model checking utilisable en pratique. Toutes les techniques de réduction de l'espace d'états cherchent à exploiter des symétries dans l'espace afin de n'en explorer qu'une sous-partie sans perte de généralité. Les techniques de réduction par ordre partiel (POR) proposent de ne pas explorer tous les ordres globaux résultant de l'interclassement des ordres locaux de chaque processus. En effet, certains de ces ordres globaux sont sémantiquement équivalent entre eux, car

l'ordre d'événements locaux indépendants les uns des autres n'a aucune influence sur le comportement du système. Ces méthodes sont donc particulièrement adaptées aux systèmes distribués de par l'absence d'horloge globale qui caractérise ces systèmes. En effet, le model checker, en calculant toutes les histoires globales possibles du système, calcule donc tout les interclassements possibles entre les histoires locales des processus. Dans un contexte distribué, il est attendu que de nombreux événements locaux soient indépendants, et qu'il en résulte une forte symétrie entre les différentes histoires globales possibles.

Transitions indépendantes et POR. Pour appliquer une méthode de réduction par ordre partiel, il est fondamental de parvenir à déterminer si deux transitions donnés sont indépendantes ou non. Si le prédicat chargé de déterminer ce fait est trop laxiste (i.e. qu'il déclare indépendant des transitions en réalité dépendantes), il introduit le risque d'une exploration trop partielle de l'espace d'état et celui de manquer des états fautifs dans le système. Un prédicat trop strict au contraire entrave la capacité à réduire l'espace d'états, menant à un processus de vérification trop long pour être applicable en pratique. L'importance de ce prédicat explique que nous le formalisions maintenant.

Formellement, l'indépendance est défini comme suit (d'après [25]). Soit Σ l'ensemble des états accessibles du modèle. La transition t est alors une fonction partielle $\mathrm{t} : \Sigma \to \Sigma$. Pour tout état $\sigma \in \Sigma$, l'ensemble des transitions activées à σ est défini par $\mathrm{enabled}(\sigma) = \{\mathrm{t_i} \mid \mathrm{t_i}(\sigma) \in \Sigma\}$. En d'autres mots, $\mathrm{enabled}(\sigma)$ est l'ensemble des transitions qui peuvent avoir lieu lorsque le système est dans l'état σ.

Definition 1. *Deux transitions* $\mathrm{t_i}$ *et* $\mathrm{t_j}$ *sont indépendantes (ce qui est noté* $\mathrm{I}(\mathrm{t_i}, \mathrm{t_j})$*) si et seulement si :*

$$\begin{aligned}\forall \sigma \in \Sigma : \mathrm{t_i}, \mathrm{t_j} \in \mathrm{enabled}(\sigma) \Rightarrow \wedge\ & (\mathrm{t_i} \in \mathrm{enabled}(\mathrm{t_j}(\sigma)) \\ & \wedge\ (\mathrm{t_j} \in \mathrm{enabled}(\mathrm{t_i}(\sigma)) \\ & \wedge\ \mathrm{t_i}(\mathrm{t_j}(\sigma)) = \mathrm{t_j}(\mathrm{t_i}(\sigma)))\end{aligned}$$

Deux transitions sont donc déclarées indépendantes si l'état atteint après avoir appliqué ces deux transitions ne dépend pas de l'ordre dans lequel elles ont été appliqué. Il faut bien sûr également que l'application de l'une des transitions n'empêche pas l'application de l'autre transition.

La détermination parfaite de l'indépendance entre deux transitions est très difficile car elle demande d'évaluer les effets des deux transitions étudiées *depuis tous les états accessibles du système*, ce qui est naturellement impossible en pratique. Il est donc courant

d'utiliser des mécanismes d'approximation, qui se doivent d'être conservatifs pour éviter de remettre en cause l'exhaustivité de l'étude par model checking. Deux transitions sont donc considérées dépendantes, sauf dans les cas où l'on peut prouver qu'elles sont indépendantes.

Les méthodes classiques de POR s'appuient sur une étude statique du modèle visant à déterminer des ensembles de transitions indépendantes avant même de commencer l'exploration de l'espace d'états. Cette approche est malheureusement mal adaptée au cas de la vérification de code source car de nombreuses (in)dépendances entre transitions ne peuvent être établies qu'à l'exécution, quand les valeurs des variables sont connues. Cette approche mène donc à trop peu de réduction dans notre cas.

Réduction dynamique par ordre partiel (DPOR). L'idée centrale de cette approche est qu'il est possible de déterminer l'indépendance entre les transitions une fois qu'elles ont été exécutées, c'est à dire lors que l'exploration atteint la fin d'une histoire explorée, et que le model checker se prépare à rétablir l'état initial afin d'explorer une autre histoire. En effet, les valeurs des variables influant sur les transitions ainsi que les effets observés sur l'état global sont alors connus. Cette analyse doit cependant être menée intelligemment pour que son coût puisse effectivement être contrebalancé par la réduction ainsi rendue possible.

L'algorithme de DPOR utilisé dans SimGridMC se base sur celui introduit par Palmer et Al. dans [49] tout en le simplifiant : puisque nous ne cherchons qu'à vérifier des assertions locales et des absences d'interblocage, nous ne détectons pas les cycles dans le graphe d'états. Le pseudo-code de notre algorithme est donné page 92. À chaque étape q de l'exploration est associé un ensemble *interleave*(q) de transitions activables à q et dont les successeurs seront explorés. Initialement, une transition p arbitraire est sélectionnée puis explorée. À chaque pas de l'exploration, le model checker considère l'étape q qui se trouve au sommet de la pile d'exploration. S'il reste au moins une transition dans q qui n'ait pas été explorée, et si la limite de profondeur de recherche n'a pas été atteinte, l'une des transitions t de q est effectuée, résultant en une nouvelle étape q'. Le model checker pousse q' sur la pile, identifie les transitions à explorer depuis cet état, et itère le processus.

Au moment du backtracking, l'algorithme itère sur toutes les étapes de la pile d'exploration, du sommet à la base de la pile. Pour chacune, il cherche l'étape $\mathrm{s_j}$ la plus récente de la pile telle que la transition *tran*$(\mathrm{s_j})$ exécutée pour générer $\mathrm{s_j}$ est dépendante avec la transition *tran*(q) de l'état au sommet de la pile. Si une telle étape existe, la transi-

tion *tran*(q) est ajoutée à la liste des transitions à explorer depuis l'étape précédent s_j (si elle n'a pas encore été explorée). Cet algorithme est plus simple que la présentation originale de DPOR dans [19] car il suppose que les transitions restent activables jusqu'à être exécutées, conformément à l'état de fait dans SimGrid. La preuve de la correction de cet algorithme, donnée en section 3.8, n'est pas reproduite dans ce résumé par soucis de concision.

Dépendance entre transitions dans SimGrid. Il serait fastidieux de déterminer manuellement la dépendance entre les actions réalisables au sein des interfaces offertes par SimGrid (MSG, GRAS ou MPI), car elles ne disposent d'aucune spécification formelle. La formalisation d'un sous-ensemble substantiel de MPI proposé dans [49] donne par exemple lieu à plus de 100 pages de spécification en TLA⁺. Au lieu de cela, nous avons choisi de formaliser le noyau de 4 primitives de communication offertes par SIMIX au chapitre précédent. Ces fonctions étant les seuls moyens de modifier l'état partagé, il est suffisant de baser le calcul de dépendance à ce niveau puisque toutes les opérations complexes des interfaces utilisateurs résultent de combinaisons de ces fonctions. Cela simplifie grandement le travail de formalisation nécessaire pour calculer le prédicat d'indépendance. Trois pages de spécification TLA⁺ sont suffisantes pour cela. La figure 3.5, page 95, présente la formalisation des données de notre modèle tandis que la figure 3.6, pages 97 et 98 présente la formalisation des primitives offertes par le noyau de communication. En combinant ces formalisations des primitives et la définition formelle de l'indépendance donnée page 100, nous pouvons déduire les théorèmes d'indépendance suivants (dont les preuves sont données dans la version complète de ce document, en anglais) :

Théorème 2. *Toute transition* Send *est indépendante avec toute transition* Recv *(et réciproquement).*

$$\begin{aligned}&\forall p_1, p_2 \in \mathrm{Proc}, rdv_1, rdv_2 \in \mathrm{RdV}, d_1, d_2 \in \mathrm{Addr}, c_1, c_2 \in \mathrm{Addr} :\\ &\quad \mathrm{I}(\mathrm{Send}(p_1, rdv_1, d_1, c_1), \mathrm{Recv}(p_2, rdv_2, d_2, c_2))\end{aligned}$$

Ce résultat peut sembler contre-intuitif, mais il vient du fait que les données ne sont échangées que lors d'un Wait ou un Test. Si le Send et le Recv concerne des boîtes aux lettres différentes, ils sont trivialement indépendant. Dans le cas contraire, l'ordre dans lequel ces requêtes sont postées n'influe pas sur le résultat, ce qui les rend également indépendant.

Théorème 3. *Deux* Send *ou deux* Recv *réalisés par des processus différents et postés sur des boîtes aux lettres différentes sont indépendants.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{rdv}_1, \mathrm{rdv}_2 \in \mathrm{RdV}, \mathrm{d}_1, \mathrm{d}_2 \in \mathrm{Addr}, \mathrm{c}_1, \mathrm{c}_2 \in \mathrm{Addr} :\\ \mathrm{p}_1 \neq \mathrm{p}_2 \wedge \mathrm{rdv}_1 \neq \mathrm{rdv}_2 \Rightarrow \wedge\, \mathrm{I}(\mathrm{Send}(\mathrm{p}_1, \mathrm{rdv}_1, \mathrm{d}_1, \mathrm{c}_1), \mathrm{Send}(\mathrm{p}_2, \mathrm{rdv}_2, \mathrm{d}_2, \mathrm{c}_2))\\ \wedge\, \mathrm{I}(\mathrm{Recv}(\mathrm{p}_1, \mathrm{rdv}_1, \mathrm{d}_1, \mathrm{c}_1), \mathrm{Recv}(\mathrm{p}_2, \mathrm{rdv}_2, \mathrm{d}_2, \mathrm{c}_2))\end{aligned}$$

Théorème 4. *Des opérations* Wait *ou* Test *concernant la même communication sont indépendantes.*

$$\begin{aligned}&\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c} \in \mathrm{Addr} : \mathrm{I}(\mathrm{Wait}(\mathrm{p}_1, \{\mathrm{c}\}), \mathrm{Wait}(\mathrm{p}_2, \{\mathrm{c}\}))\\ &\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c}, \mathrm{r}_1, \mathrm{r}_2 \in \mathrm{Addr} : \mathrm{I}(\mathrm{Test}(\mathrm{p}_1, \mathrm{c}, \mathrm{r}_1), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}_2))\\ &\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c}, \mathrm{r}_2 \in \mathrm{Addr} : \mathrm{I}(\mathrm{Wait}(\mathrm{p}_1, \{\mathrm{c}\}), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}_2))\end{aligned}$$

Théorème 5. *Des actions locales de processus différents sont indépendantes.*

$$\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc} : \mathrm{p}_1 \neq \mathrm{p}_2 \Rightarrow \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Local}(\mathrm{p}_2))$$

Théorème 6. *Toute action locale est indépendante avec toute transition* Send *ou* Recv *activable en même temps qu'elle.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{rdv} \in \mathrm{RdV}, \mathrm{d} \in \mathrm{Addr}, \mathrm{c} \in \mathrm{Addr} :\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Send}(\mathrm{p}_2, \mathrm{rdv}, \mathrm{d}, \mathrm{c}))\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Recv}(\mathrm{p}_2, \mathrm{rdv}, \mathrm{d}, \mathrm{c}))\end{aligned}$$

Théorème 7. *Toute action locale est indépendante avec toute transition* Wait *ou* Test *activable en même temps qu'elle.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{comm} \in \textsc{subset}\ \mathrm{Addr}, \mathrm{c}, \mathrm{r} \in \mathrm{Addr} :\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Wait}(\mathrm{p}_2, \mathrm{comm}))\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}))\end{aligned}$$

Ces théorèmes nous permettent naturellement d'atteindre notre objectif de déterminer un prédicat de dépendance conservateur (c'est à dire considérant les transitions comme dépendantes, sauf à pouvoir démontrer leur indépendance). Il suffit d'établir une disjonction de toutes les closes ainsi obtenues.

3.3 Résultats expérimentaux

Cette section présente des résultats expérimentaux basés sur la vérification dynamique de programmes utilisant deux interfaces utilisateur de SimGrid : MSG et SMPI. Nous démontrons ainsi la capacité de notre approche à appliquer DPOR à différentes interfaces de communication grâce au niveau intermédiaire d'abstraction. Chaque expérience tend à évaluer l'efficacité de la réduction obtenue par DPOR.

Expériences basées sur SMPI. Dans cette première expérience, nous avons utilisé deux exemples simples de programmes MPI, construits spécifiquement pour l'occasion. Le code de ces programmes est présenté page 106. Le premier d'entre eux est une communication all-to-one où tous les processus sauf le premier envoient un message au premier. La propriété que nous avons demandé au model checker de vérifier est que le dernier message reçu provient du processus de rang N. Cette propriété est bien évidement fausse dans le cas général. La table 3.1a (page 107) présente les chronométrages et nombres d'états visités avant de trouver un contre-exemple, en fonction du nombre de processus (entre 3 et 5) et selon l'algorithme d'exploration utilisé (DPOR ou exploration en profondeur d'abord sans optimisation). L'impact du nombre de processus apparaît comme relativement limité dans cette expérience car le contre-exemple est découvert assez rapidement, mais le bénéfice de DPOR apparaît malgré tout clairement ici. La table 3.1b présente les résultats obtenus lorsqu'on désactive l'assertion, forçant ainsi le model checker à parcourir tout l'espace d'états. DPOR offre dans ce cas un gain de performance d'un ordre de grandeur.

Le second exemple est construit pour mettre en valeur l'intérêt des calculs d'indépendance. Chaque processus dont le rang est un multiple de trois attend les messages des deux messages suivants. Il est clair que chaque groupe de 3 processus est indépendant des autres groupes, mais des méthodes classiques de POR ne pourraient le déterminer puisque le rang du processus n'est connu qu'à l'exécution. La table 3.1c présente les résultats obtenus pour une exploration complète de l'espace d"états, c'est à dire sans que la découverte d'un contre-exemple ne coupe court à l'exploration. La recherche en profondeur a été interrompue après une heure de calcul, sans avoir réussi à terminer. Ceci est à comparer au fait que DPOR a réussi à terminer l'exploration complète en une demi-seconde.

Vérification du protocole Chord. Notre seconde expérience vise à vérifier un algorithme plus réaliste puisqu'il s'agit d'une implémentation de du protocole P2P bien connu Chord [59], en utilisant l'interface MSG de SimGrid. Le code simplifié de cet algorithme est donné en page 109.

En exécutant cet algorithme au sein du simulateur, nous avons constaté qu'il arrivait qu'une tâche lue en ligne 14 pouvait s'avérer parfois invalide, menant à une erreur de segmentation du programme. À cause de l'ordonnancement déterministe du simulateur, nous ne sommes pas parvenu à reproduire ce problème avec moins de 90 nœuds dans le système, ce qui rendait l'analyse du problème extrêmement complexe.

Nous avons donc décidé d'étudier le problème par model checking en étudiant un scénario ne contenant que deux processus, en insérant $task == update\_task$ comme seule propriété à vérifier, en ligne 16 du programme. En quelques secondes, le model checker nous a offert la trace de contre-exemple reproduite dans le listing 3.6 (page 110), qui nous a permis de comprendre l'origine du problème. Cette trace se lit de haut en bas, et les événements sont indentés pour les aligner en fonction du processus concerné. La cause du problème est donc que la tâche de notification envoyée par le nœud 1 à la ligne 22 est confondue par le nœud 2 avec une réponse à une requête de mise à jour émise en ligne 14. Ceci est dû à une erreur d'implémentation en ligne 12 : le code réutilise la variable *rcv_comm* en s'appuyant sur la supposition erronée qu'il est sûr de le faire grâce aux conditions de garde de cette branche. Le fait est que la condition peut changer après que la garde soit évaluée.

L'implémentation de Chord que nous avons vérifié comptait 563 lignes de code, et le model checker a trouvé le bug après avoir visité seulement 478 états (en 0,28s) avec DPOR. Sans cette optimisation, il doit visiter 15600 états (en 24s) avant de trouver le bug. Dans les deux cas, la consommation mémoire est de l'ordre de 72Mb.

4 Résumé du chapitre 4 : parallélisation de la boucle de simulation

Les systèmes à simuler étant toujours plus massifs, l'extensibilité constitue un défi d'importance pour la simulation. Le facteur limitant d'une simulation peut être soit la mémoire, soit le temps de calcul (soit les deux). La contrainte mémoire est liée au nombre de processus simulés (et leur consommation mémoire propre) ainsi qu'à la taille de la plate-forme simulée (nombre de nœuds de calcul, de liens réseau, etc.). Le temps de

calcul de la simulation est quant à lui lié aux calculs menés par les processus simulés eux-mêmes ainsi qu'à la quantité d'interactions entre les entités du système. Ainsi, les processus d'un système HPC réaliseront sans doute beaucoup de calcul par eux-mêmes, mais n'auront que peu d'interactions entre eux tandis que la situation sera complètement en P2P où chaque processus se contentera de calculs assez léger, mais où de très nombreuses interactions entre processus auront lieu.

Concernant la contrainte mémoire, il est toujours possible de contourner le problème en ajoutant des barrettes mémoire dans l'ordinateur, mais cette solution n'est pas applicable pour la contrainte liée au temps de calcul tant que la simulation reste séquentielle. Les approches classiques pour paralléliser une simulation par décomposition spatiale de la plate-forme sont difficiles à mettre en œuvre dans le cas de la simulation de systèmes distribués : L'efficacité des techniques dites pessimistes dépend grandement des latences constatées sur la plate-forme. Le lookahead maximal utilisable sur une plate-forme où les latences sont faibles serait trop petit pour permettre une exécution parallèle efficace. À l'inverse, les techniques dites optimistes ne posent pas de restriction sur le degré de parallélisme observé dans le système, mais elles demandent de pouvoir revenir en arrière quand une incohérence temporelle est constatée dans les messages reçus des autres nœuds. Cette correction est une opération complexe, dont le coût peut facilement dépasser les gains du parallélisme quand le nombre de messages augmente. Enfin, les approches de décomposition temporelle nécessite de pouvoir prédire un état futur probable pour le système, ce qui s'avère extrêmement difficile dans notre cas puisque l'état des processus est une pile d'exécution système.

4.1 Exécution parallèle lors de la simulation

Dans ce chapitre, nous proposons un nouveau schéma de parallélisation basé sur la constatation que les processus simulés ne modifiant que leur état local, ils sont intrinsèquement parallèles. Notre approche est donc de garder la simulation séquentielle, mais de paralléliser l'exécution des processus utilisateurs. L'avantage majeur de cette approche est qu'elle évite complètement le problème classique de la simulation parallèle, à savoir le maintien de la cohérence entre les évolutions temporelles réalisées par chaque élément de la simulation. Au lieu de cela, une seule évolution temporelle est utilisée, en exploitant le parallélisme potentiel existant en son sein.

L'algorithme de simulation utilisé dans SimGrid avant les travaux présentés ici est rappelé page 115. L'idée centrale est que le temps simulé n'avance que lors des appels

au noyau de simulation en ligne 6. Cela signifie que toutes les actions effectuées par U_1 et U_2 se déroulent à l'exact même instant simulé, et qu'il n'y a donc aucune différence observable selon l'ordre dans lequel ces actions sont calculées. D'un point de vue formel, toutes les transitions de $\mathrm{P}_{\mathrm{time}}$ sont parfaitement concurrentes. Il n'y a donc aucun risque d'exécution désordonnée de ces actions, qui peuvent donc être exécutées en parallèle. Le nouvel algorithme de simulation est présenté page 116. La seule différence est que les processus simulés sont exécutés en parallèle, et le reste de l'algorithme reste inchangé par ailleurs. Il convient bien entendu, de noter que ceci n'est rendu possible que par le travail de refactoring présenté au chapitre 2. Il assure en effet que l'état partagé n'est pas modifié directement par les contextes d'exécution des processus utilisateurs, mais uniquement par le maestro de la simulation lors du traitement des requêtes posées par les processus.

En ce qui concerne l'implémentation de ce schéma de parallélisation, plusieurs choix s'offrent à nous. Comme nous l'avons vu précédemment, SimGrid exécute chaque processus simulé au sein d'un contexte d'exécution `ucontext`, tel que standardisé par POSIX. Ce mécanisme peut être vu comme une extension de `setjump` et `longjmp` permettant de sauvegarder la pile système en plus des registres du processeur. Une première approche pour permettre l'exécution parallèle de ces contextes est de les faire évoluer en des threads à proprement parlé, mais ce n'est pas adapté à notre contexte pour plusieurs raisons. Tout d'abord, le nombre de threads que les systèmes d'exploitation actuels est de l'ordre de quelques dizaines de milliers là où nous visons des simulations de plusieurs millions de processus. De plus, même lorsque cette limitation n'est pas atteinte, cette implémentation reste inefficace car le nombre d'entité de calcul du processeur est bien inférieur au nombre de contextes à exécuter. Le système d'exploitation doit alors gérer la contention de ressources entre les contextes, au moyen de changements de contextes inutiles.

Une approche plus intéressante consiste à utiliser autant de threads que d'unités de calcul dans la machine, et à répartir les différents contextes à exécuter entre eux, comme représenté dans la figure 4.2b, page 116. De cette façon, aucune contention entre les threads système n'est à craindre tout en permettant une exécution parallèle. L'implémentation est plus complexe que dans le cas précédent, mais reste possible en combinant des threads classiques et des `ucontext`, qui permettent de manipuler les contextes d'exécution comme des valeurs du premier ordre à la façon des continuations. La figure 4.3 (page 118) représente une exécution parallèle comprenant 2 threads T_1 et T_2 pour 4

contextes utilisateurs U_1, …, U_4. La charge de travail est répartie entre les threads tout en limitant au maximum les besoins de synchronisation entre eux.

Analyse des coûts de la simulation. Il est courant de penser que la parallélisation constitue une panacée pour les performances en temps. La simulation constitue cependant un problème intrinsèquement séquentiel, et il est donc primordial de comprendre les différents coûts en jeu afin de comprendre les compromis réalisés lors de l'implémentation, ainsi que pour prédire les scénarios dans lesquels le parallélisme est le plus prometteur. La figure 4.4 (page 118) présente une répartition temporelle de l'exécution séquentielle et de l'exécution parallèle. Le nombre d'itération de la simulation reste inchangé après parallélisation, ainsi que les coûts dus au noyau de simulation SURF ou à la couche de virtualisation SIMIX. La différence apparaît au niveau de l'exécution des processus utilisateur (dont le temps passe de la somme de chacun d'entre eux au max des sous-sommes correspondantes à chaque thread worker), et aux coûts de syncronisation entre threads worker naturellement absents dans la version séquentielle. Cela permet de déduire simplement les scénarios où la parallélisation va s'avérer bénéfique : il s'agit des cas où les coûts de synchronisation entre threads vont être compensés par les gains résultant de l'exécution parallèle.

La taille du code utilisateur a donc un impact primordial sur les bénéfices potentiels de cette approche. Dans une simulation composée de peu de processus effectuant des calculs relativement lourds (comme c'est attendu en HPC), la parallélisation sera trivialement gagnante tandis que dans une simulation composée d'un très grand nombres de processus effectuant des calculs simples (comme attendu en P2P), il sera très difficile d'amortir les coûts du parallélisme grâce aux gains ainsi obtenus.

Une autre façon de favoriser le parallélisme est de jouer sur la précision numérique de la simulation [1]. Comme le temps est discrétisé dans une simulation à événements discrets, la précision numérique indique le nombre de timestamp existant sur un intervalle de temps. Si le nombre de processus vient à grandir énormément, il est attendu que les événements soient répartis sur moins d'étapes de simulation quand la précision numérique décroît (puisque la quantité de timestamp possibles décroît). Les coûts de synchronisation seront donc rentabilisés plus facilement par la plus grande abondance

1. Le lien entre réalisme de la simulation et précision numérique utilisée pour les calculs est moins important qu'on pourrait le penser à première vue. En effet, les modèles utilisés pour la simulation se révèlent en général relativement stable numériquement. Le réalisme dépend donc plutôt du modèle utilisé et de la finesse de la modélisation résultante que de la seule précision numérique

de parallélisme potentiel à chaque étape de simulation.

4.2 Implémentation efficace

Au regard de cette caractérisation, il convient donc d'une part de réduire au maximum les coûts de synchronisations entre threads, et d'autre part de veiller à ce que la charge de travail entre les threads soit aussi équilibrée que possible. Nous présentons maintenant une solution à chacun de ces problèmes.

Groupe de workers efficace. Différents schémas sont possibles pour permettre au maestro de signifier aux worker threads de commencer leur traitement, ainsi que permettre inversement aux workers de signifier au maestro que ce traitement est fini. Une première approche consiste à utiliser le mécanisme classique des barrières (une au début du traitement, et une à la fin). Mais ce mécanisme ne faisant pas partie de POSIX de manière native, nous devons la réimplémenter manuellement. La façon classique consistant à utiliser une variable de condition et un compteur s'avère cependant très coûteuse en temps car d'après le standard, tout accès à la variable de condition doit être protégé par mutex. Il en résulte une quantité d'appels systèmes prohibitifs en termes de performance.

Nous avons donc choisi de nous baser sur un nouveau schéma de synchronisation adapté à nos besoins. Il repose sur la notion d'événement, qui similaires à la fois aux variables de condition et aux barrières. Les deux seules primitives offertes sont *signal* et *wait*. Chaque événement est associé à un groupe de threads en attente d'un signal. Signaler un événement est une opération bloquante qui libère tous les threads du groupe. Ils réalisent les calculs qui leur sont dévolus, avant de bloquer à nouveau en attente du prochain signal. Lorsque tous les threads sont bloqués, le contrôleur qui les avait libéré est débloqué à son tour.

Ce mécanisme peut être implémenté de façon parfaitement portable en se limitant aux services offerts par POSIX, ou bien de façon très efficace en utilisant les futexes offerts par les noyaux Linux modernes. Dans le second cas, il est possible de n'effectuer que N+3 appels système par itération de simulation, où N est le nombre de thread worker. En particulier, cette grandeur ne dépend plus du nombre de tâches à effectuer dans la file. Nous proposons les deux implémentations (celle efficace et celle portable) afin de pouvoir nous adapter à toute situation.

Répartition de la charge entre workers. Après les travaux présentés dans le chapitre 2, la seule donnée partagée entre les workers est la liste des tâches elle-même. Une approche triviale consisterait à protéger tout accès à la liste par un mutex, mais il en résulterait une quantité d'appels systèmes contradictoire avec nos objectifs de performance. Une autre approche est de répartir statiquement les tâches entre les workers au début de chaque itération, mais les calculs effectué par le code utilisateur pouvant varier grandement d'une tâche sur l'autre, cette approche s'avère également peu efficace. Notre approche est basée sur l'usage de primitives atomiques très efficaces offertes par les processeurs modernes, dont le coût est de l'ordre de quelques cycles processeur seulement. Lorsqu'il cherche une tâche à effectuer, chaque thread incrémente de façon atomique un index indiquant la prochaine tâche à effectuer. De cette façon, aucune synchronisation n'est nécessaire entre les threads, et nous avons la garantie d'une répartition quasi optimale de la charge entre les workers : il ne peut y avoir de workers inactifs que s'il n'existe pas de tâche qui ne soit déjà commencée par un autre worker. Pour réaliser une meilleure répartition de la charge, il serait nécessaire de connaître à l'avance les coûts de chaque tâche pour s'assurer que la dernière tâche commencée, potentiellement réalisée par un seul thread tandis que les autres attendent, n'est pas la plus longue. Mais cette connaissance est impossible à obtenir dans notre contexte puisque le travail induit par chaque tâche résulte des programmes fournis par l'utilisateur.

4.3 Résultats expérimentaux

Les expérimentations présentées ci-après sont basées sur deux exemples. Le premier est un algorithme classique de produit parallèle de matrices (PMM) par double diffusion de blocs à chaque itération. Il a été sélectionné pour sa symétrie intrinsèque, ainsi que pour la grande quantité de calculs effectués par chaque processus à chaque étape. L'autre exemple est le protocole Chord, que nous avons déjà utilisé dans les expérimentations du chapitre précédent. Nous l'utilisons dans un scénario expérimental similaire à celui utilisé dans [3], où n nœuds ajoutés au système à $\mathrm{t} = 0$. Chaque nœud réalise une opération de stabilisation toutes les 20 secondes, une opération *fix_fingers* toutes les 120 secondes, et une recherche arbitraire toutes les 10 secondes. La simulation est arrêtée après 1000 secondes simulées. Afin d'assurer que les différentes expériences restent comparables dans notre cadre, nous avons réglé les paramètres (tels que la latence réseau) pour que le nombre total de messages échangés au cours de l'expérience (et donc, la charge induite sur le moteur de simulation) reste comparable d'une expérience sur l'autre. Ainsi, environ

25 millions de messages sont échangés dans le scénario à 100,000 nœuds. Ces expériences ont été réalisés sur l'une des machines de Grid'5000 [8] dotée de deux CPU AMD 12-core à 1.7 GHz avec 48 GB de mémoire. La version de SimGrid utilisée est v3.6 beta (identifiant de version git : 8d32c7).

Coûts de synchronisation. Pour évaluer précisément le coût induit par les synchronisations entre threads, nous avons comparé les temps de l'exécution séquentielle à ceux d'une exécution parallèle ne comptant qu'un seul thread worker. Le gain potentiel du parallélisme étant nul dans ce cas, cela nous permet de mesurer le surcoût de la synchronisation par simple comparaison.

Dans le cas de PMM, ce coût n'est pas mesurable car le nombre d'itérations de simulation est extrêmement réduit, ce qui tend à écraser les coûts de synchronisation au regard du temps d'exécution de la simulation. Dans le cas de Chord, nous avons mesuré un surcoût d'environ 16% (76s supplémentaire sur une simulation de 471s). Ces coûts, relativement élevé malgré le degré de sophistication apportée à notre parallélisation, montrent que l'exécution parallèle peut tout à fait ne pas être bénéfique dans certains cas.

Accélération parallèle. Nous présentons maintenant une série d'expériences visant à quantifier l'accélération offerte par le parallélisme dans différentes situations. Dans le cas de PMM, où les calculs utilisateurs dominent les coûts, le parallélisme est trivialement gagnant. Lors d'une multiplication de matrices de taille 1500 par 9 nœuds, la version séquentiel prend 31s là où la version parallèle à 4 threads ne prend que 11s. Cela représente une accélération de plus de 50%.

Le cas de Chord est plus difficile puisque ses processus échangent un grand nombre de messages (chargeant ainsi le noyau de simulation), mais réalisent des calculs relativement simples (offrant donc peu de parallélisme potentiel à notre approche). Dans ces conditions, la précision numériqueε du modèle s'avère être un paramètre précieux pour augmenter le parallélisme potentiel en réduisant le nombre de timestamps existants, et donc en augmentant le nombre de processus prêts à s'exécuter à chaque timestamp. La table 4.1 (page 126) montre clairement cet impact de ε sur le parallélisme potentiel, puisque passer ε de 10^{-5} à 10^{-1} permet de multiplier le nombre moyen de processus prêts à chaque itération de calcul de 10 à 251. Changer de modèle permet également d'augmenter le parallélisme potentiel (en réduisant encore une fois le nombre de timestamps considérés). Ainsi, le modèle constant de SimGrid permet d'avoir en moyenne

7424 processus prêts à être exécutés à chaque pa de temps de la simulation.

Les chronométrages complets sont représentés dans la figure 4.8 (page 127). La partie haute correspond au modèle constant où tout échange de message dure 0.1s, quel que soit l'émetteur et le récepteur. Dans ces conditions, une simulation de 2 millions de nœuds dure en 3h18 en séquentiel, contre 2h24 en parallèle. Les ratios représentés sous la figure montrent que le mode parallèle est préférable au mode séquentiel même pour un nombre relativement limité de processus simulés. Le speedup maximal est de 40% (pour 24 threads).

La partie base de la même figure rapporte les chronométrages observés en utilisant le modèle réaliste de SimGrid décrit dans [66]. Dans ce cas, la précision numérique a un impact majeur sur les performances. Avec la précision par défaut de $\varepsilon = 10^{-5}$, nous n'avons pas réussi à simuler plus de 300 000 nœuds en moins d'une nuit de calcul. Pour $\varepsilon = 10^{-1}$, la réduction du nombre de timestamps possibles et l'augmentation du parallélisme potentiel nous permet de simuler 2 millions de nœuds en 8h15 en séquentiel, contre 7h15 en parallèle. Le gain relatif du parallélisme est bien moindre dans ce cas que précédemment (20% au maximum), et ne devient positif que pour les scénarios comptant plus de 500 000 processus.

Un élément bien plus troublant de ces résultats est que ε impacte également les performances de la simulation séquentielle. Ce résultat est inattendu car la quantité de travail reste constante quand ε varie. Nous n'avons pas d'explication établie pour ce phénomène, même si les analyses préliminaires tendent à montrer qu'il pourrait s'agir d'une pollution du cache mémoire.

Concernant la limite mémoire, 36Gb de mémoire sont nécessaires pour simuler 2 millions de nœuds. Cela représente 18kb par nœud, dont 16kb sont attribués à la pile système du processus.

Comparaison à OverSim. La figure 4.8 (page 127) compare également les résultats de notre environnement à ceux d'OverSim [3] sur la même machine. Dans le graphique du haut, Oversim était configuré pour utiliser son modèle le plus simple, comparable à notre modèle constant. Nous ne sommes pas parvenu à simuler plus de 300 000 nœuds en moins d'une nuit de calcul avec ces réglages. Pour le graphique du bas, nous avons utilisé le modèle INET d'OverSim. Utilisant un simulateur des paquets réseau, il offre un degré de réalisme comparable à notre modèle réaliste. Malheureusement, le simulateur n'est pas parvenu à terminer l'initialisation de 30 000 processus après une nuit de calcul, montrant ainsi une extensibilité très limitée.

Ces résultats mettent clairement en valeur l'extensibilité et les performances de notre solution, y compris avec nos modèles réalistes. En particulier, il est tout à fait notable de constater que le pire des cas de SimGrid (modèle réaliste, $\varepsilon = 10^{-5}$) offre de meilleures performances que le meilleur des cas d'OverSim (modèle constant), alors que le réalisme des modèles n'est sans aucune commune mesure entre ces deux réglages.

Introduction

1 Scientific Context

Distributed systems are in the mainstream of information technology. It has become standard to rely on multiple distributed units that collectively contribute to a business or scientific application. Having multiple distributed units that work simultaneously at multiple parts of a problem can greatly improve the performance of the application, help tolerate component failures, handle problems too large to fit in a single processing unit, or accomodate with geographically distributed computing resources.

The amount of information processed by critical applications that rely on this technology increases every day, as does the need to understand how these systems behave and perform.

The heterogeneity of distributed systems and their scope of application pose serious challenges when devising a methodology to analyze them. The wide spectrum of architectures and applications translates to different notions of performance that complicates the analysis. For example, High Performance Computing (HPC) systems are evaluated by the time they take to finish processing all jobs (makespan), whereas Peer-to-Peer (P2P) networks focus on the amount of exchanged messages to fulfill the goal while remaining resilient to the dynamic nature of the network. Cloud computing introduces yet another notions of performance like economic variables such as the cost of running a program in rented CPUs in the client's side, and the amount of consumed energy or wasted resources in the provider's side.

There are also correctness concerns common to all of these systems. The design of algorithms adapted to the distributed context is particularly difficult. The distributed nature of the platforms on which these applications run are more likely to experience failures, often due to unreliable communication channels or remote host problems for which there is no perfect detection method. The design of distributed systems is a com-

plex and difficult task, because the developer has to deal with these external factors that condition the behavior of the program, but are outside of its control.

Moreover, most programmers are used to assume that the entire state of the program is available and directly accessible at any moment of the execution. This is not true in a distributed setting where the processes execute in their own local memory, and their knowledge about the others and the global state is potentially outdated. A great body of work is dedicated to the study of algorithms to obtain global consistent snapshots and it is still an active field of research.

Another complication inherent to many distributed systems is the lack of a common clock to synchronize the executing entities. This poses difficulties to ensure mutually exclusive access to shared resources, or to determine the ordering of events that happened in different processes.

In general, the developer must provide the abstractions to deal with the asynchrony and the partial view of the system state which are a common source of many errors. Finally, the inherent parallelism of a distributed scenario can also suffer from classical algorithmic issues of centralized multithread programs, like potential race conditions, deadlocks and live-locks.

2 Motivation

With the complexity of distributed programs on the rise, it becomes necessary to develop new methodologies and tools to help the developer better understand the behavior of these systems. Despite the differences of architecture and application fields, the distributed systems can all be abstracted as independent processing entities with private local states interacting through the exchange of messages. This common abstraction permits to use the same kind of techniques to study all of them.

Among the different observable properties of a distributed systems we can identify two important groups : those related to the performance and those regarding the correctness. To evaluate the performance aspects of distributed systems several approaches are available. The most natural one is the direct execution of the program over a given testbed. Its main advantage is that it exposes the real behavior of the program. Unfortunately, it also suffers from the real complexity of the platform. First, it is very time consuming since the programmer may face issues in the testbed system not directly related to its application, such as diverging compilation environment or loss of connection

to the testbed. Then, many different execution platforms exist, and it is difficult to assess the performance of the application on another platform than the one that the programmer has access to.

Another approach is the use of simulations. This is a well-known and popular methodology offers the ability to study the application using simulated platforms. The code runs in a controlled environment that is easier to setup than real executions, and permits to analyze the performance of the system in many more platforms than the ones that might be really available to the developer, allowing more comprehensive test campaigns. Its downside is the lack of the real experimental bias, like for example the background noise in a network link, however it is usually possible to adjust the precision of the models that describe the resources according to the user needs.

To evaluate the correctness of distributed systems, none of the previous approaches are well suited. The non-deterministic behavior of the real executions makes debugging near to impossible as the problematic execution paths are extremely difficult to reproduce. Simulation improves the situation, as it allows the reproduction of execution traces and thus helps in the debugging process. However, it solves only a part of the problem because the developer has to determine whether the test campaign is sufficient to cover all situations that may happen in real settings. In other words, the simulation is not exhaustive and a program that correctly behaves in a simulator can still fail when deployed in real life because of some execution paths that were not explored in the simulator.

All these reasons explain why most of the time, distributed applications are only tested on a very limited set of conditions before being used in production. To address the complexity of analyzing/verifying the correctness of distributed systems, several formal verification techniques gained momentum in recent years. One option is the use of proof assistants to verify if a model describing the behavior of the system satisfies a given specification. Generally, the user must provide the model written using a specific formalism, and then the proof is done in a semi-automatic manner, asking for user input when no progress can be made. The advantage of this technique is that on success it offers a complete guarantee of correctness no matter how big the system is. On the other hand, it requires experienced users and in general proofs take a lot of time to complete. Moreover, correctness is ensured with respect to the properties / specification for which it has been verified. There may still be omissions in the set of properties, and also the model may be incorrect with respect to the real system, which is hard to determine in a theorem-proving approach.

Model checking is another technique used to establish whether a model meets its specification. An attractive feature of model checking is its fully automatic nature that allows non expert users to verify systems with less effort than with proof assistants. Conceptually, a basic model checker for safety properties explores the state space of the model looking for invalid states that do not meet the specification. The search continues until it finds a state that violates some correctness property, or until the whole state space is explored, or it runs out of resources. In the case of an invalid state, the model checker provides a counter-example that consists of the execution trace that leads to it. Model checking is often more effective at discovering bugs than traditional testing due to its exhaustive nature, that considers even corner cases that might otherwise be overlooked. The downside is that sometimes it is not capable of producing an answer due to the potential size of the state space.

	Real Execution	**Simulation**	**Proofs**	**Model Checking**
Performance Assessment	☺☺	☺☺	☹	☹
Environmental Effects	☺☺	☺	n/a	n/a
Experimental Control	☹	☺☺	n/a	n/a
Ease of use	☹	☺☺	☹	☺☺
Correctness Assessment	☹☹	☹	☺☺	☺
Result if failed	n/a	n/a	☹	☺
Automatic	☹	☺☺	☹	☺☺

Table 1: Comparison of methodologies.

Table 1 summarizes and compares the characteristics of each approach. From a user perspective simulation and model checking offer the best methodological advantages, being both fully automatic and relatively simple to use. They are also complementary regarding the kind of properties they allow to study. Simulation is better fitted to study the performance and behavioral aspects of the application at large scale but in specific scenarios of interest. On the other hand, model checking permits to assess the correctness of the systems and provides powerful debugging capabilities thanks to the counter-examples it generates when properties are violated.

Despite the close relation between simulation and model checking, the research communities and available tools rarely interact. It is important to clarify that some model checkers also have simulation capabilities, however here the term is used to denote the ability to animate some executions, but none is capable of taking a platform as an in-

put parameter and generating the execution of the system with the timing constraints imposed by it.

A developer willing to use a simulator is often forced to write a prototype adapted to it, and there is no guarantee that the real application will behave as predicted by the simulations. Similarly, many model checkers accept only models written in specific abstract specification languages like Promela for Spin, or TLA$^+$ for TLC, that requires building a second prototype. Having different models or prototypes to study aspects of the same application complicates the process and leads to an elevated development cost.

There are model checkers that avoid writing models, instead they operate at program source level, because it is well-known that even if a model was shown correct, many bugs are introduced during the translation to code. Software model checkers for distributed systems, are often tailored for specific communication APIs (e.g. MPI), and generic solutions require the user to provide the infrastructure required to simulate a distributed setting.

We think that an appealing approach to close the gap between simulation and model checking is to unify both methodologies in the same framework. Having such tool can greatly simplify the development of correct distributed systems, as it eliminates the cost of writing multiple models for the same application.

3 Thesis Objectives

The overall objective of this thesis is to develop the theory and tools required to provide a unified framework for the study and development of distributed computer systems, capable of assessing performance aspects as well as correctness properties directly on executable programs.

To provide such a solution, this thesis develops two main ideas :

1. On the correctness side, we explore the idea of integrating a simulator and a model checker in a single tool. We think that having a common framework with both functionalities can ease and encourage the use of these complementary techniques at any point of the development process. The implementation work is based on the SimGrid Simulation Framework. We first propose an analysis of SimGrid's current design to fulfill the new model checking functionality. Once all the architectural differences are clear, we plan a refactoring work to simplify the integration of the model checker. The key challenge of this work is to avoid affecting the performance

of the simulations. The success of the model checker relies on the ability to cope with the enormous number of interleavings that distributed programs generate. Usually, many of these interleavings are equivalent in the sense that they lead to indistinguishable global states. To tackle this problem we explore a novel architecture to apply state space reduction techniques in a generic way, independent of the communication scheme of the application. Finally, we plan to validate our results with several case studies using small programs that are intentionally bugged, and also with more realistic examples.

2. In the area of performance assessment, we envision fast simulations at P2P scales, with millions of interacting processes running in Internet-like platforms. To this end, we try to speed up the simulations that are CPU bound, by leveraging the power of current multi-core machines. For this, we explore the idea of parallelizing the execution of the application under simulation. We first plan to compare the computational costs of the parallel approach against the classical sequential execution, to understand in which scenarios a speed up can be expected from the parallelism. After having a better understanding of the parallel execution tradeoffs, we will focus into the optimization of the simulator's performance. Next, we propose an experimental validation of the approach with several case studies covering different types of distributed systems (HPC, and Grid). Moreover, we test the scalability and performance of SimGrid when dealing with P2P applications, comparing it against another established simulator in the area.

4 Structure of the Thesis

The rest of this document is organized as follow :

Chapter 1 presents the state of the art on the main topics approached in this work. This comprises a general introduction to distributed systems (the object of study in the thesis), an overview of the simulation area (in particular applied to distributed systems), an overview of model checking focused towards software verification, and finally a detailed description of SimGrid, the technical context of the work.

Chapter 2 introduces the first contribution of the thesis, that consists of bridging simulation and model checking into a single framework. We analyze the limitations of SimGrid's current architecture to fulfill the requirements introduced by the model checker

and the parallelization of the simulation loop. Moreover, we present a novel design inspired from ideas of operating system architectures that serves as the basis to the rest of the work in the next chapters.

Chapter 3 presents SimGridMC, the model checker integrated in the SimGrid simulation framework, and it is the second contribution of the thesis. We discuss the design choices, the implementation issues, and we describe in great detail the technique used to cope with the state explosion problem. In addition, we present the results of several verification experiences, that show the efficiency of the approach.

Chapter 4 attacks the problem of CPU bounded simulations through parallelization, and is the third contribution of this work. We detail the architecture of the parallel simulation loop, and we analyze its complexity compared to the sequential version. Moreover, we present the optimization work that uses the complexity analysis as a guideline to introduce the lowest possible overhead with the parallelization. Last, we present the results of several simulation experiences that justify the complexity analysis, and show the scalability of SimGrid when dealing with large simulation instances, and the benefits of parallelism.

Finally, we present the conclusions of this work and detail the future lines of research that arise from the results of this thesis.

Chapitre 1

State of The Art

1.1 Distributed Systems

Conceptually, a *Distributed System* consists of multiple autonomous computing entities connected by a computer network that interact towards the solution of a common goal [1]. The program executed by a distributed system is called a *Distributed Program*. Even if this definition suggests a physical separation of the computing entities, nowadays the term is used in a wider sense, that comprises processes executing in isolated memory spaces either in the same or different hosts and that communicate only by the exchange of messages.

The origins of distributed computer systems goes back to the early 60s, with the study of concurrent processes that communicate by message-passing in operating systems architecture [1]. Nowadays distributed systems have became the core of many business and scientific applications. There are two main reasons to rely on distributed systems. First, the natural need for information exchange among different geographical locations. For many businesses, data is produced in one location, but is needed in another location. For example, sharing computing resources among geographically distributed users. Second, for improving different aspects of the computing infrastructure. Sometimes, critical systems must be tolerant to hardware failures or software errors. Process replication across distributed computing entities is then a plausible option. Another aspect is scalability, typically when a problem is too big to be solved by a single computer, either by spatial or temporal limitations. Again a distributed platform can solve the problem by splitting into subproblems [66]that are assigned to the nodes in the system, and when done the results are gathered back to reconstruct the solution.

In this section we first present a taxonomy of distributed systems pointing out the relevant characteristics and purpose of each class of system. Next, we approach the challenging problem of designing distributed algorithms, showing what are the main issues that make developing these a complex task. Finally we present an abstract modeling for all distributed systems used within this document.

1.1.1 Distributed Systems Taxonomy

The last decade has brought tremendous changes to the taxonomy of large scale distributed computing platforms. These can be classified according to the characteristics of the computing entities and the topology of the network that interconnects them :

HPC & Grids. Science *In Silico* has become the third pillar of science through the simulation of the phenomena to study. These simulations now constitute a crucial tool in disciplines such as particle physics, cosmology, biology or material engineering. Simulations used to be carried out on large ad hoc supercomputers known as *Grids*, that were originally composed by spare machines located on different locations, usually on campuses, connected through National Research and Education Networks (NREN) that configured a rather static topology. Nowadays, in modern grids the spare machines are replaced by high end commodity clusters mainly due to efficiency reasons (i.e., sets of off-the-shelf computers interconnected by fast switches). Indeed, the technological advances driven by the home PC market have contributed to achieving high performance in commodity components. The main issues in Grid Systems relate to the applications now used by scientists in their day to day work. They are concerned by interoperability and data exchanges within virtual organizations, focusing for example on trust forwarding between virtual organizations.

On the other hand, High Performance Computing (HPC) systems comprise thousands of nodes, each of them holding several multi-core processors. For example, one of the world fastest computers, the Fujitsu K Computer system [60] at RIKEN Advanced Institute for Computational Science (Japan), contains 68544 SPARC64 VIIIfx 2000 MHz processors, each with eight cores, for a total of 548,352 cores. The issue in HPC is naturally massive performance, and a traditional quality metric is the percentage of the peak performance achieved by a solution.

It should be noted however that these two research communities are very close in practice, with numerous links between them. Beyond considerations of technical com-

patibility between the proposed solutions, accountability and trust management between virtual organizations, the classical research questions in grid systems encompass the dimensioning of the system (computational elements, networks and storage) to accept the load and ensure that all scientist jobs get handled with reasonable delay, and that all scientific data can be stored and retrieved afterward.

Peer-to-Peer Nowadays, similar hardware is used in home PC and in HPC nodes, with a time gap measured in months only. With the popularization of DSL and other high speed personal connections, it becomes tempting to leverage this large potential located at the edges of the network, by individuals. *Peer-to-Peer* (P2P) networks try to leverage the power of millions of spare hosts distributed across the world [58]. A key characteristic of P2P applications is the dynamic nature of the topology, as nodes can join/leave the system anytime during the execution.

The main challenge to address is the lack of organization in the resulting aggregated system. Adequate solutions must be completely distributed, without any central point, which are thus called Peer-to-Peer systems. Beyond the removal of any centralization points in the proposed algorithms, the classical research questions encompass the adaptation to node volatility (called churn in this context), that get off-line without notice when individuals halt their computers. Because of the experienced latencies and of the burden induced by P2P applications on the ISP networks, it is also crucial to discover and leverage the physical infrastructure underlying the P2P network (which is difficult given the churn).

Cloud The advent of high speed Internet connections can now replace the NREN networks, and similarly private data centers can replace the clusters to create *Clouds*. In this architecture the computations are sent to rented machines and returned to their owner afterward. Since it is often impossible to know where the rented machines are, the computations are said to be sent to the Cloud. This is the underlying infrastructure of many commercial Internet applications due to the low maintenance cost, flexibility, and scalability. Cloud computing is changing how computing services are constructed, as software can scale almost infinitely according with customers' elastic demands. The pay-as-you-go model often results in improved resource utilization and hence lower costs [61].

The research questions in Cloud computing are split in two main areas. From the provider point of view, the placement of virtual resources upon the physical ones in order to optimize various aspects (e.g., performance, energy consumption, resource usage,

etc), and the strategy of allocation of virtual resources to the clients are critical to the performance and profitability of cloud platforms. From the client point of view, clouds are very interesting for a wide range of applications, from web site hosting to scientific computations. Yet, clients have to select, dimension, and reserve resources by themselves. This process is often done in an empirical and sub-optimal way, while there are opportunities to design software brokers which would optimize the usage of resource with respect to the price paid from the client perspective.

Table 1.1 summarizes the classification of distributed systems according to the nature of the underlying platform.

Type	Network Type	CPU Scheme
Grids	NREN	clusters (hundreds)
Clouds	Internet	clusters (hundreds)
P2P	Internet	spare (millions)
HPC	LAN & SAN	clusters (thousands)

TABLE 1.1: A classification of distributed systems according to the nature of their resources.

1.1.2 Distributed Algorithms

Despite the different topologies, communities, and application fields, the underlying algorithmic model of the distributed systems is the same. They can all be abstracted as a set of autonomous entities that execute asynchronously and communicate by the exchange of messages. The design of algorithms adapted to this context is more complicated than developing centralized programs, as they differ in essentially three important reasons :

1. **Lack of knowledge about the global state**. In a centralized program the entire state of the process is directly observable, and thus it is possible to make control decisions based upon it. Instead, in a distributed algorithm there is no direct way of observing the global state of the system. The entities that compose it execute in their own isolated local states, and the knowledge about the state of the other entities can only be obtained through the exchange of messages. Hence, the control decisions can only be based upon the local knowledge of the state of the system,

that might be potentially outdated and thus invalid.

2. **Lack of a common time reference.** In a centralized system the events produced by the processes are totally ordered in a natural way by their temporal occurrence. For every pair of events it is always possible to determine which occurred earlier than the other. This can be used for example as a basis to implement mutually exclusive access to shared resources. In a distributed algorithm, the total order of the events cannot be observed (even if they actually happen in a certain order). The processes execute in different hosts with local clocks that are not synchronized, hence the lack of a common time reference renders the determination of the order among events of different hosts impossible.

Distributed algorithms also have a third reason of complexity know as nondeterminism, however this is suffered by all parallel platforms (centralized or not). Having multiple simultaneous execution flows leads to race conditions that require explicit synchronization schemes to maintain the coherence of the computation. These complicate the understanding of the global behavior of the application, and their incorrect usage can easily result in dead locks, live locks, or fairness issues.

The non-determinism, the lack of knowledge about the global states, and the impossibility to observe the total order of the events in the system makes designing distributed algorithms a challenging craft, and thus very error prone.

1.1.3 Model of a Distributed System

Along this document, we consider a distributed system as a set of loosely coupled message-passing processes that do not share memory nor global clock. We denote as P the set of individual processes, and these are connected through channels that we assume are reliable (error free), FIFO and unbounded.

Each $\mathrm{p} \in \mathrm{P}$ has a local state $\mathrm{s_p}$ that evolves by the executions of events (or actions). We note as E the set of all events in the system, and we denote $\mathrm{e^p}$ to refer to the events executed by the process p.

Because the total order of the events executed in a run cannot be observed in the context of distributed systems, we use an alternative ordering that captures the causality relation among them. This is known as the *happened before* relation and it was originally proposed by Lamport [35], who argued that it is all that can be observed in a run of a distributed system. Events executed by a single process are totally ordered using the natural sequential order. Then given an event $\mathrm{e_p}$ and a later event $\mathrm{f_p}$ of the same process,

we say that e_p locally precedes f_p and we note it $e_p \prec f_p$. We say that e_p remotely precedes f_q if the first corresponds to the send of a message from process p and the second to the corresponding receive of process q, and we note it $e_p \rightsquigarrow e_q$. We now state the definition of the happened-before relation from [22] :

Definition 2. *The happened before relation ($\rightarrow$) is the smallest relation that satisfies :*

1. $(e \prec f) \vee (e \rightsquigarrow f) \Rightarrow (e \rightarrow f)$, *and*
2. $\exists g : (e \rightarrow g) \wedge (g \rightarrow f) \Rightarrow (e \rightarrow f)$.

A *computation* (or run) in the happened before model is then defined as a tuple $(E, \rightarrow)$ where E is the set of all events and $\rightarrow$ is a partial order on the events in E. It is noteworthy to mention that instead of considering the partial order among events in E it is also possible to consider it over the set of local states. For local and send events, we consider the predecessor state, and for the receive event the successor. The behavior of a distributed system is formally characterized by the set of all possible relations $\rightarrow$ that it generates. Finally e and f are concurrent ($e \parallel f$) if neither $e \rightarrow f$ nor $f \rightarrow e$.

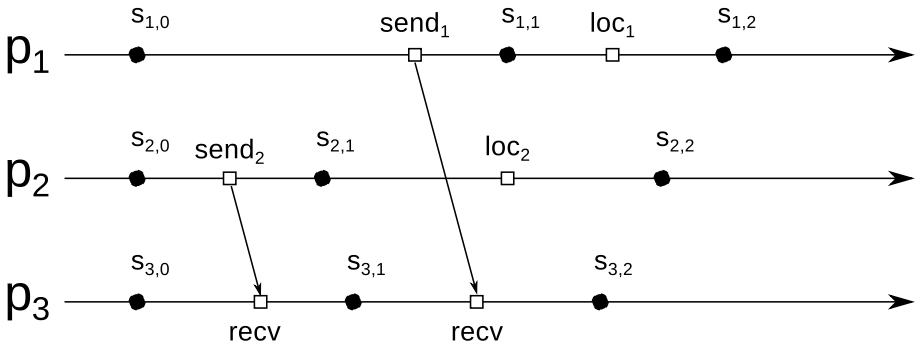


FIGURE 1.1: A run in the happened before model.

Figure 1.1 illustrates a run in the happened before model. Here $(s_{1,0} \rightarrow s_{3,2})$, $(s_{2,0} \rightarrow s_{3,2})$, but $(s_{1,0} \parallel s_{2,1})$.

1.2 Simulation

For decades, simulations have been used by physicists, mathematicians, engineers, computer scientists, and even video game designers. They offer a mean to predict the probable outcome of experiments and aid to understand large and complex systems, facilitating their performance analysis, and optimization. The distributed computer systems are a particular case of such systems, hence simulation is a classical methodology

to ease their development and analysis. It removes the need of real testbeds to test the applications together with all its operational complications.

In this section, we first summarize the area of simulation in a general sense, as a methodology that offers different approaches depending on the desired study and the phenomena under analysis. Then we focus on the particular case of simulating distributed systems, showing the methodological alternatives, their advantages, drawbacks, and common challenges. Next, we recapitulate the state of the art in simulation of distributed systems, and compare the different available solutions for each kind of system. Finally, we give a more abstract and rigorous definition of simulation that is used throughout this thesis.

1.2.1 Simulation in Science and Engineering

Simulation has taken a fundamental place in both theoretical and experimental science. It allows the study of phenomena that are too complex to be tractable by a purely theoretic approach, and permits to predict the evolution of systems without performing most of the corresponding real-world experiments.

A simulation model is an abstract representation of the physical system under study. Models can be classified according to how their state evolves [37, 18] :

Continuous-time. The state of the model changes continuously in time. These models are better suited to simulate evolutive systems like physiochemical reactions usually described by differential equations. The simulation consists in solving these equations numerically for each time interval. The application field is wide : chemical reactions, physical deformations, engineering, but also in more theoretical areas like for the numerical invalidation of a theory by comparing the experimental results with the predictions of a simulator encoding the theory.

Discrete-time. The state of the model changes instantaneously at discrete points in time. The simulation of these models is know as *discrete-event simulation* (DES) and it is better suited to study the interaction among components with discrete behavior. Conceptually, DES consists of evaluating the state of each component separately to predict the next event of the system, and then applying the effects of its execution. The application field is equally vast, for example in the industry it allows to study and optimize the interactions among the production lines, or the behavior of the clients in an environment

like a bank.

It is important to note that the choice between these two types of simulation is usually given more by the type of the envisioned study than by the type of system under analysis. The granularity of the study is equally important in this choice. Hence, the traffic in a city can be modeled like a discrete-event system, capturing the cycles of each traffic light, and the behavior of each car, or it can be described as a continuous system to study the global flow and thus predict the overall circulation conditions.

1.2.2 Parallel Discrete Event Simulation

Simulation is an inherent sequential process, whose evolution depends of the the entire state of the model. Some approaches to perform parallel discrete event simulations (PDES) exist but in general there is no golden rule to obtain an efficient parallel simulation. The most common approach consists of decomposing the simulation to compute it in multiple processors [73]. There are two main orthogonal approaches :

Space-Parallel Decomposition. The model is decomposed in several sub-models in the space domain, that can an be assigned to the different processors [21]. This decomposition is attractive for distributed systems due to the natural separation of the autonomous entities that compose these. The simulation of the sub-models can be computed in parallel, however dependencies among these might exist, hence the parallel simulation algorithm should avoid out-of-order executions to guarantee the soundness of the results. (One sub-model advanced too much in time and missed a delayed event from other sub-models).

The techniques to avoid out-of-order events can be classified in *conservative* and *optimistic*. The first, only allows the entities to advance forward in the simulation time when it is sure that no event from the past would be missed. The drawback is that these are heavily dependent on the communication latency among the entities. The second, does not restrict the entities that are allowed to advance in the simulation time freely. However, when an out-of-order event arises, the simulation is rolled-back to a consistent point and recomputed to maintain the soundness of the results. For big simulation states, these are very expensive in terms of computational costs, as they require periodic snapshots to avoid unnecessary re-computations.

Time-Parallel Decomposition. The simulated time domain is partitioned into intervals $[t_{i-1}, t_i]$ for $0 < i \leq p$, being p the number of available processors [15, 31]. Each proces-

sor computes the simulation for a given interval, and with this approach, the simulator must ensure that the state of the model at the end of interval $[t_{i-1}, t_i]$ matches the initial state of $[t_i, t_{i+1}]$. If the states do not match, then the second interval is recomputed using the correct initial state. Therefore, the efficiency of the technique depends on the ability to predict the initial state of each interval. The dynamic nature of the state of simulations that rely on executable programs makes impracticable the direct application of this approach.

1.2.3 Simulation of Distributed Systems

The simulation of distributed computer systems is a commonly used technique to analyze and optimize this kind of programs. However, there are fewer tools and standard methodologies compared to other domains. This may be explained by the relative simplicity of the platforms used until recently. When using a dozen of homogeneous computers running standard CPU bound applications, there is no real need for complex simulation frameworks. The ongoing increase of complexity of distributed computer systems explains why their study through simulation is evolving into a scientific field on its own. The simulation avoids the need of the real execution platform, instead it relies on many possible techniques that emulate the behavior of the underlying platform.

There are 4 possible approaches depending on the desired granularity of the study :

Microscopic DE. Computer Systems are inherently discrete, thus it is tempting to capture all the behaviors of all the components without any kind of abstraction. For example the network can be modeled at packet level as a sequence of events such as packet arrivals and departures at end-points and routers, the CPU at cycle-accurate level, and the behavior of a disk drive can be described by imitating the mechanics of the read/write heads and plates rotation. In general, with these models the simulation time increases in proportion to the number of events [39].

Macroscopic DE. Certain phenomena are not described in detail, but approximated numerically. For example, a classic approach is to abstract the treatment of each network packet with mathematical functions that represents the data streams as fluids in pipes [40, 47, 48]. If these functions are correctly chosen, it is possible to obtain an approximation that is precise, yet being lighter and faster than a microscopic DE simulation [66]

Constant Time. These simulations do not capture precisely the duration of the events, that are considered to cost all the same amount of time. This is useful to study the interactions among components without considering the underlying platform. This simplification allows to quickly compute the general evolution of large scale systems (like P2P).

Mean Field Models. When the amount of interacting entities of the system is too big, studying them at individual scale becomes intractable. To overcome this difficulty the mean field models try to exploit the behavioral symmetries of the entities. They abstract them into groups that shows a homogeneous behavior. Examples of the application of these models are : queuing systems [43], epidemic models [10], medium access control (MAC) [5], or network congestion protocols such as TCP [2] among others. There are two approaches to analyze this models. One is to use a classical DES but at class level, where events represent the interaction among groups of entities. Another alternative, is to assume that the amount of entities goes to infinity and study the convergence of the model.

Each approach is a trade-off between the precision and the computational costs of the simulation. Microscopic DE is the most detailed, but also the most expensive in terms of CPU and memory requirements, and usually does not scale very well. On the other end, mean field models permits the maximum scalability, however the systems are represented very abstractly and thus not well suited for the scale of individual entities. It is natural to think that an increased level of precision correspond to more accuracy in the results, and thus better simulations. However, it is important to mention that perfect models can sometimes render the simulations counterproductive or even wrong. First, a very detailed description of the system can generate an enormous amount of irrelevant data that complicates the observation of the phenomena under study. Next, many low-level hardware and protocols rely on the improbability of certain phenomena to happen due to imperfections in the hardware and environmental effects of the real world. An example of this is the *Phase Effect* introduced by Floyd et al in [20]. Therefore perfect models might even yield erroneous predictions.

Another important aspect of the simulation of distributed system is how to specify the model of the distributed program. A first possibility is to use an abstract representation of the processes, usually described in a formalism like automaton, DEVS, Omnet++, or a CSP-like language. This has the advantage of being light on resources as it omits many

details and hence allows good scalability. However the models require manual translation into code in order to obtain an executable system, procedure that is known to be a very error prone.

A second approach that aims to overcome the previous shortcoming is to use *Domain Specific Languages* (DSL) that are designed to simplify the integration of the applications with the tools to study them (like a simulator, but others are also possible for example a model checker), but they also allow to automatically extract an executable implementation. This is the case of MACE [33], or GOAL [27].

A third approach is to rely directly on the executable code of the program. It allows for more realistic simulations at the expense of higher computational costs, and in some cases it permits to apply the technique to already existing applications not originally designed for the simulator. Typically, the simulator executes the program in a controlled environment and intercepts all the interaction of the program with the platform to simulate it using the models of the resources.

A fourth option is to replace the program with a trace of events previously extracted or artificially generated. This is known as *offline* simulation, and permits to decouple the execution of the systems from the simulation procedure. This provides more flexibility, and sometimes it can be used to simulate programs that are too big to fit in a single simulation run.

1.2.4 State of the Art of Distributed System Simulators

There are three classical challenges for a simulator : accuracy, speed and scalability. Attaining two of the three challenges already poses difficulties, developing a tool that accomplish all of them is a really complex and difficult task that not only requires an excellent design but also a very careful implementation. This partially explains why no general simulator yet exists for all kinds of distributed systems. Grid computing often requires precise simulations of relatively big processes, whereas on P2P systems processes are smaller but more numerous, hence simulators reduce the precision to solve the problem. Because of this, over the last decade, several simulation solutions for distributed systems emerged as references for each domain of application.

In the area of P2P computing, PeerSim [30] is a widely used simulator that was designed to be scalable and simple to adapt to the users' needs. It uses a *query-cycle* mode for maximum scalability, where the simulator's main loop iterates over every node to execute one action at each step. It was reported to simulate up to one million nodes in

this mode [30]. However, it does not simulate the CPUs, and the network uses a time-constant model where any communication takes the same amount of time. The simplifications done for sake of extensibility and scalability result in quite unrealistic simulation predictions.

OverSim [3] is another P2P simulator that tries to address the accuracy limitations of PeerSim by delegating the handling of communications to a discrete-event simulation kernel OMNet++ [65]. This consists of a packet-level simulator comparable to NS2 [41]. It was reported to simulate up to 100,000 nodes, but when OMNet++ was replaced by other internal mechanisms, however the validity of the results was never clearly demonstrated. Numerous other simulation projects were also proposed from the P2P community, such as P2PSim [23] or PlanetSim [52]. But these projects proved to be short lived and are no longer maintained by their authors.

Similarly, in the area of Grid computing numerous tools were produced. However, most of them were intended to be used only by their developers. ChicSim [55] and OptorSim [4] are specifically designed to study data replication on grids. The first uses discrete-event models to simulate the CPUs and network, whereas the second one has a hybrid approach using discrete-event models for the CPU, and analytical models for the network. OptorSim is also capable of simulating hard disk quotas. Their downside is that both mimic the flow fragmentation into packets that happens in real networks, but they do not take TCP flow management mechanisms into account. None of these simulators are actively developed any longer.

GridSim [12] is another widely used Grid simulator which was initially designed for studying Grid economy and was later used in a more general-purpose way. It is capable of simulating CPUs, network and hard disks using discrete-event models. GridSim uses a Microscopic DE modeling for networks, CPUs, and disks. The main shortcoming is that the packet level approach used to simulate network flows is simplistic, it only implements a protocol that includes some elements of UDP and allows for variable packet size. This yields unrealistic simulations as it does not consider classical TCP mechanisms for congestion control, and contention resolution. Because the accuracy of the simulations depends on the size of the packets (the smaller, the better), GridSim does not scale very well, or if it does the results of the simulations are not realistic.

SimGrid [14], the framework on which the implementation work of this thesis is based on, was conceived to study scheduling algorithms but nowadays the concept evolved to provide a general solution for the simulation of distributed systems. One of SimGrid's

main features is that it allows the user to plug in custom models of the resources. Additionally, SimGrid provides some models that are fast, yet proven realistic [66], which use a macroscopic DE approach that relies on validated analytical functions to represent data streams as flows in pipes, yielding fast but yet precise simulations. As an alternative, it can optionally use the GTNetS [57] packet-level network simulator. For the moment SimGrid does not simulate hard disks, due to the difficulty of designing accurate models. To correcly simulate disks timings, the models should describe not only the hardware aspects but also the file system layer, that has an noteworthy impact in the performance of an application.

In the field of Cloud computing, and maybe due to its recent nature, only CloudSim [13] can be considered as a reference. It is based on GridSim, but exposes specific interfaces to study Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) settings. Being based on GridSim, it is subject to the same limitations.

The Table 1.2 summarizes the characteristics of each simulator.

Simulator	Type of System	Models		
		CPU	Network	Disk
SimGrid	Grid, P2P	Coarse D.E.	Analytical or Fine Grained D.E.	N/A
GridSim	Grid	Coarse D.E.	Coarse D.E.	Fine Grained D.E.
ChicSim	Grid	Coarse D.E.	Coarse D.E.	N/A
OptorSim	Grid	Coarse D.E.	Analytical	Disk quota only
PeerSim	P2P	N/A	Constant	N/A
OverSim	P2P	N/A	Fine Grained D.E.	N/A
CloudSim	Cloud	Coarse D.E.	Coarse D.E.	Fine Grained D.E.

TABLE 1.2: Comparison of Simulators by Type of Model.

1.2.5 Elements of Simulation

The rest of this document is centered around DES. In this section we propose a more precise characterization to simplify the presentation of the ideas in the following chapters.

There are two different notions of time when we speak about simulations, one is the simulation time (real world time), i.e. the time elapsed on the host machine running the simulation, while the second one is the simulated time, that is the time elapsed in the simulated world. For example, consider the case of a simulation of the weather evolution

for one week, that takes one hour to complete. Here, the simulation time is one hour, and the simulated time is one week.

Simulation. *Given a distributed system composed of a set of processes* P*, a description of the platform* R*, and analytical models* M *of the resources instantiated in* R*, then a simulation is an algorithmic procedure to compute one possible run of* P *(the partial order* $\rightarrow$ *in the set of events* E*), with timestamps* $\mathrm{t_i}$ *associated to each of its events that represent the* simulated time *at which each event finishes. These timestamps are real numbers computed by a function* $\mathrm{T_{M,R}}$ *that uses the models* M *subject to the restrictions imposed by the resources* R*.*

$$\mathrm{S(P, R, M)} = \langle \rightarrow, \mathrm{T_{M,R}} : \mathrm{E} \rightarrow \Re \rangle$$

Note that with this definition, it is possible to have two or more events with the same timestamps if the models determine that they finish at the exact same simulated time.

Algorithm 1 illustrates the classical procedure to compute $\mathrm{S(P, R, M)}$. On each it-

Algorithm 1 Classical algorithm to compute $\mathrm{S(P, R, M)}$.

1: **while** some event may occur **do**
2: compute the time t and the set of next finishing events
3: advance simulation to time t
4: compute the reaction of the entities
5: **end while**

eration of the loop the value of t corresponds to one timestamp in the simulation. The actions that finish at time t trigger some reactions in the entities that are computed at line 4. All the actions performed by the entities during this step happen at the same simulated time t. We refer to the iterations of the loop as *scheduling rounds*, as they execute a subset of the entities each time. We call this set of these entities a *scheduling set* and it is defined as follow :

Scheduling Set. *Given a simulation* $\mathrm{S(P, R, M)} = \langle \rightarrow, \mathrm{T_{R,M}} \rangle$*, for each timestamp* t *in the range of* $\mathrm{T_{R,M}}$ *we define the* scheduling set *(*$\mathrm{P_t}$*), composed of the processes which have a blocking event that finishes at time* t*. These are the processes selected by the simulator to be executed at the simulated time* t*.*

$$\mathrm{P_t} = \{\mathrm{p} \in \mathrm{P} \mid \exists \mathrm{e} \in \mathrm{E} : \ \mathrm{T_{M,R}}(\mathrm{e^p}) = \mathrm{t}\}$$

The notion of scheduling set is important for the work on parallelization presented in Chapter 4. Conceptually, the reactions of all processes in $\mathrm{P_t}$ can be computed in parallel without the risk of any out of order interleaving as everything happens at simulated time t.

1.3 Model Checking

Developing correct distributed algorithms is a challenging task. Besides the natural algorithmic complications of the domain, such as the absence of a global clock, the lack of shared memory, and the non-determinism, the programmer has to deal with the complexity of the execution platform. Therefore, ensuring the correctness of a distributed program is not a trivial task, and cannot be achieved by classical approaches like testing or simulation.

The use of model checking to formally verify distributed systems is a plausible alternative. It proceeds in a fully automatic manner and requires less expertise to use than other alternatives such as theorem proving. In this section, we first present a general overview of the model checking verification procedure, stating is advantages and shortcomings. Then we detail how this technique can be applied to verify software, in particular distributed systems.

1.3.1 Introduction

Model checking is an algorithmic verification technique that is used to determine whether an abstract model M meets a formal specification φ expressed as a logic formula. The problem is usually formulated as

$$\mathrm{M}, \mathrm{s_0} \vdash \varphi$$

where $\mathrm{s_0}$ is the initial state of the model M. Often, M is expressed as a *Labeled Transition System* (LTS) that characterizes all possible behaviors of the system, and φ is a *temporal logic property* describing the expected behaviors of M.

In linear-time temporal logic, formulas are interpreted over sequences of states, and can be classified in two categories :

Safety properties. The validity of safety properties depends only on finite histories, they express facts of the type *"nothing bad can happen"*. Absence of deadlock, or race con-

ditions are examples of such properties. Their truth value can be determined by recording the parts of the computation history relevant to the properties, and then inspecting single states of the LTS.

Liveness properties. The validity of liveness properties depends on infinite histories, and they express facts *expected to happen* in the system. Determining the truth value of liveness properties is more complicated than for the safety case, mainly because they depend on infinite histories of computations. More precisely, the crucial aspect of a liveness property is that it can only be falsified over an infinite execution. Hence, one possibility to verify a liveness property is to determine if the LTS of the model is free of strongly connected components that contain a cycle along which the liveness property is violated.

The verification of safety properties is also called the reachability analysis problem. Given a model M, an initial state s_0, and an assertion φ, the problem is reduced to determining if it is possible to reach a state s' from s_0 where φ won't hold. The model checking algorithm in this case consists of an exhaustive exploration of the model's state space. At every visited state, the validity of the properties specified by φ is tested. The search continues until the model checker finds a state that violates the specification, or until the whole state space is explored, or when it runs out of resources. In the case of an invalid state, a model checker provides a counter example showing the trace of events from the initial state that leads to the state violating the property.

The ability to show counter examples and the automatic nature of the procedure are the features that make model checking an extremely popular tool, even outside the academia with proven cases of success in the industry.

1.3.2 The State Explosion Problem

The most problematic shortcoming of model checking is the size of the LTS that has to be explored, as it grows exponentially with the number of variables and asynchronous components of the model. To illustrate this, a system component with N variables of k possible values each yields up to $\mathrm{S} = \mathrm{k}^{\mathrm{N}}$ states. Then, a model built from P such components running asynchronously yields $\mathrm{k}^{\mathrm{N}\times\mathrm{P}}$ states. This exponential grow of the LTS is known as the state explosion problem. The exploration algorithm has to visit each state at least once to check the validity of the specification, hence it has an exponential complexity.

A great body of work is dedicated to cope with the state explosion problem. Early model checkers encoded the transition relation explicitly using adjacency lists. For systems with a small number of concurrent processes this was enough, but with bigger instances, the size of the state transition system was too large to handle this way. McMillan et al proposed using a symbolic encoding for the state transition system based on ordered binary decision diagrams (OBDDs), that resulted in a more compact representation of the transition relation. Moreover, the symbolic representation captures particularly well some of the regularity in the state space generated by circuits and protocols. Hence, symbolic model checking enabled the verification of systems with more than 10^{20} states [11].

Using model checking to verify software is more difficult, as software tends to be less structured than hardware. In addition, concurrent software is often asynchronous, as is the case of distributed programs where processes perform actions independently from the others. This leads to very big state spaces, and therefore hinders the chances of success using model checking to verify these programs. One of the most effective techniques for handling asynchronous systems is the *partial order reduction* (POR) [64, 24, 50]. The idea of POR is to reduce the state space to be explored by avoiding the exploration of traces that differ only in the ordering of events that are independent. In Chapter 3.7 we present this technique in more detail as is the approach followed in this thesis.

Despite the advantages of model checking using OBDDs, there are a number of shortcomings that restrict the size of the models that can be checked using the approach. The size of an OBDD is sensitive to the ordering of the variables, and it is known that for some boolean formulas, no space-efficient ordering exists. An alternative solution to avoid OBDDs is to rely on SAT solvers, following an approach known as Bounded Model Checking (BMC) [6]. Even though the SAT problem is NP-complete, the power of modern SAT solvers to deal with problems that occur in practice has significantly increased over the last 15 years. A BMC encodes the state as a vector of boolean variables, and the transition relation as a propositional formula over these variables. Then, all the paths of length k (k-path) can be encoded as the conjunction of k times the transition relation formula. Finally, the verification consists of using a SAT solver to determine if the conjunction of the k-paths formula with the negation of the property formula is satisfiable or not.

A last approach, is known as the Abstraction Refinement Loop. The idea is to generate an abstraction of the model that has a smaller state space, but that *simulates* it. This guarantees that if a universal CTL^* property (hence in particular LTL) holds on the abstract model, then it also holds on the concrete model. But the converse is not true, a

property valid in the concrete system may fail to hold in the abstraction. Then the model checker verifies the abstraction, and if it finds a counterexample, it has to check if it is also a counterexample in the original model. If it is not, then it is an *spurious* counterexample and it is used to improve the abstraction. This workflow is called *Counterexample Guided Abstraction Refinement* (CEGAR) [16].

1.3.3 Software Model Checking

As its name indicates model checking was initially geared towards the verification of abstract models. Later on, it was used in a wide area of industrial applications, in particular for the verification of hardware designs. There, the models described the behavior of the hardware, and model checking was able to automatically give formal guarantees about their correctness, or counterexamples in the case of not meeting their specifications. This was of extreme importance, as the designs were getting more and more complex, and traditional simulation techniques were of little help to ensure correctness. On the other hand, the verification of software is still a challenging area of research. The idea of applying model checking to actual programs originated in the late 1990s [67, 44], and it only gained momentum after the development of the state space reduction techniques presented in the previous Section. However, there are other difficulties besides the state explosion problem that complicate the use of model checking with software.

A methodological inconvenience that hinders the widespread adoption of model checking for software verification is the availability of the models themselves. Traditional model checkers require the models to be written in specification languages like TLA$^+$ for TLC [36] or Promela for SPIN [28]. In the process of hardware engineering, the development of an abstract model is a fundamental required step, and thus integrates seamlessly with model checking. However, during the software development process it is common to skip the modeling phase or do it using informal or semi-formal languages such as UML that are not well adapted to formal verification. Moreover, as most of the existing software lacks such a model, it is necessary to write one only to use a model checker. The development of a model for a software (existing or not) that exactly describes the intended behaviors, is a very long and complicated process that in most cases is too expensive to face. In the few cases where a formal model is available, it is important to note that even if the model checking process succeeds, there is no guarantee that the system is correct, because the model might abstract details of the system that are relevant to the specification and thus can go unnoticed to the model checker. Finally, even if a ver-

ified software model that correctly captures all the expected behaviors of the system is available, often many bugs are introduced during the manual translation to code by the programmers.

To overcome these limitations several recent model checkers accept source code as input, that can be seen as the most accurate model possible. Dealing with source code however is not simple, as most programming languages were not designed for model checking. One complication is to generate the state space from the program code. Typically, there are two possible approaches :

Dynamic Verification This approach consists of exploring the state space of the program by executing it under the control of the model checker. It is considered the most practical *primary* approach for real-world concurrent programs [26, 63]. Dynamic verification techniques provide the most detailed representation of the systems, where the states are the real running memory of the program, and the counterexamples point to bugs in the implementation. They also suffer from the complexity of the real running state. For example, many programs freely manipulate the stack and heap, hence it becomes difficult to determine the parts to save, and costly to actually store and retrieve them, as required for state exploration algorithms. Moreover, these also need to compare states for equality, but the dynamic nature of the heap complicates the process, as many heap configurations correspond to the same observable state of the program [29]. Finally, many programs have an infinite state space and thus it is not possible to fully verify them using a dynamic approach, as every state has to be explicitly visited. However, it is still possible to bound the exploration and verify the executions of the program up to the bound. In general, dynamic model checkers are designed more as a bug hunting tool than as a mean to ensure full correctness.

Automatic Abstraction It consists of generating an abstract model of the program automatically using static analysis, or symbolic execution, and then following a classical model checking approach on the abstraction. Because the abstraction is computed by static techniques, it is usually an approximation of the program's real behavior. The objective is to reduce the level of detail for the sake of a smaller state space, and thus a lower complexity. When combined with symbolic model checking, in some cases this approach allows to verify models with an infinite state space. The idea is to use boolean representations of the programs and then algorithms that perform the state exploration implicitly, manipulating this representations [9, 17]. However, the particular case of con-

current boolean programs is undecidable [54] and thus voids the possibility of fully verifying distributed systems. As an alternative, Qadeer et al proposed in [53] to bound the number of context switches considered by the model checker, allowing full verification up to the bound.

1.3.4 State of the Art of Software Model Checkers

In this section we describe the related work on software model checking. We focus on dynamic verification because it is the approach followed in this work. Verisoft [26] is one of the first model checkers to dynamically verify concurrent programs. It uses a *stateless* approach that does not store the visited states to avoid exhausting the memory, and thus allowing to deal with bigger program instances. The backtracking of the exploration algorithm is replay-based, it simply restarts the program from the beginning and executes the previous scheduling until the desired backtracking point.

Java PathFinder (JPF) [67, 38] is a model checker for Java bytecode developed by NASA. It uses a custom Java virtual machine that allows to control the execution of the bytecode instructions, the scheduling, and to represent and store the state in a fast and compact way. To deal with the state explosion it uses static analysis to implement POR, and program slicing with respect to the property being checked to partially cover the domain of the variables.

CMC [44] is an explicit-state model checker for C programs. It is designed (but not limited) to verify protocol implementations. To execute the programs it folds the process into threads, and to ensure their memory isolation, it provides a custom dynamic memory manager that provides disjoint heap regions for each thread. Because of the complexity of comparing the running state of C processes, it uses a canonical representation of the memory, and state hashing to avoid exhausting the system memory with the set of visited states. This last technique voids the soundness of the verification process, however CMC is designed as a bug hunting tool.

CHESS [46, 45] is a state-less bug hunting tool for multithreaded software based on the WinAPI. It takes a similar approach as Verisoft, however it refines the explored traces introducing fairness constraints on the schedules, avoiding many interleavings that would be unlikely to happen when running the program with the real OS scheduler. This allows CHESS to cope with big state spaces, and spot bugs that would be missed by traditional testing.

MaceMC [32] is a model checker for the MACE [34] domain specific language. MACE

is designed to develop distributed systems, and imposes an event-driven architecture, allowing to describe the nodes' behavior as state transitions systems. Then, the MACE compiler generates a C++ implementation that can be deployed and executed as any standard distributed program. MaceMC exploits the fact that MACE programs are structured as state transition systems to perform their state space exploration, and it is capable of finding violations of safety and liveness properties. It uses a stateless approach similar to Verisoft, combined with state hashing.

CrystalBall [71] is a tool to predict and prevent inconsistencies in deployed distributed systems, implemented on top of the MACE framework. The idea is that each node continuously runs a state exploration algorithm on a recent consistent snapshot of its neighborhood and predicts possible future violations of specified safety properties. Because each node starts the exploration from actual running states of the system, it can cover relevant regions of the state space that actually happen at runtime. Additionally, if certain invalid states are detected, it can steer the execution of the system to avoid problems.

MoDist [72] is a model checker designed for transparently checking unmodified distributed systems written using the WinAPI. The transparency is achieved via an interposition layer that exposes all the interactions of the nodes with the operating system, and communicates with a centralized model checking engine that explores all relevant interleavings of these actions. This approach allows to work at binary level, without the source code of the application being verified. To cope with the state space explosion, it uses a DPOR based exploration algorithm.

In the area of HPC, ISP [69] is a verification tool for distributed programs written with the MPI library. ISP works by intercepting the MPI calls made by the program being verified. For this, the program must be compiled together with ISP's profiler. The calls to the MPI library are intercepted and notified to a special scheduler that implements a custom DPOR algorithm for MPI called Partial Order Avoiding Elusive Interleavings (POE) [63]. POE is based on a reduced execution semantics of MPI that decides when to send these calls to the MPI library, and in what orders.

1.3.5 Model Checking Distributed Programs

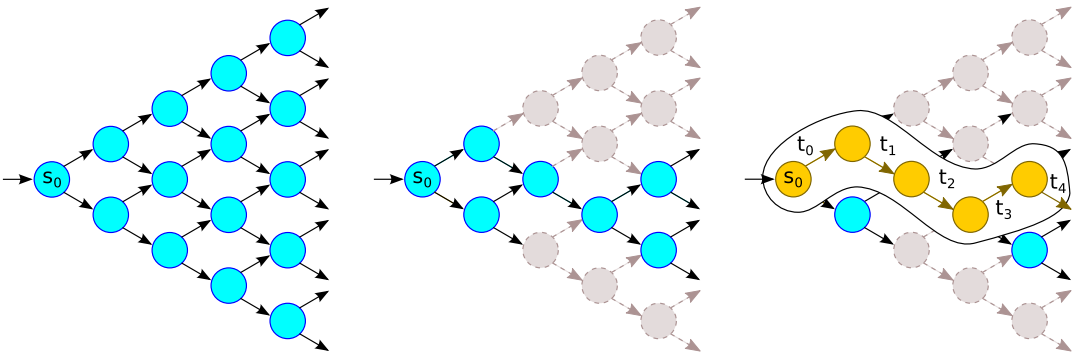
The characteristics of distributed programs make them a particularly interesting target for model checking. Assessing the correctness of a system with respect to all possible interleavings is particularly important in the distributed domain. In this regard, explicit-

state model checking works particularly well, contrary to the symbolic approach, for which there are no good symbolic representations for communication channels [7]. Even if a total verification is hard to achieve with an explicit state exploration, due to potential infinite state spaces, many non trivial bugs can be spotted using this approach, that otherwise would be hard to find and reproduce.

Another aspect of distributed systems that surprisingly benefits model checking is the lack of a global shared memory. As we previously mentioned, each process executes its computations in isolation and they can only affect each other by exchanging messages. Hence, the interleaving order of local computations cannot affect the global state. If we apply model checking as a bug hunting tool to deal with the non determinism, then we have to consider only the interleavings of the communications events, simplifying the complexity of the procedure and the implementation work.

1.3.6 Model Checking and Simulation

Simulation can be seen as a particular case of model checking, where instead of exploring the entire state space of the application, it computes a single run of it (a trace), that is consistent with the constraints imposed by the platform. An important difference is that it also computes the timings of the events in the trace, whose precision is the essence of the procedure.



(a) Complete State Space (b) Constraints From Platform (c) Execution by the Simulator

FIGURE 1.2: Model Checking vs Simulation.

Figure 1.2 outlines the differences between the techniques. The Subfigure 1.2a represents in light blue the reachable state space of a distributed system, the initial state is

labeled s_0 and the transitions among states are the communication events of the model. The Subfigure 1.2b, shows how the state space is constrained by the platform during a simulation. Here the states and transitions in gray are no longer reachable due to the restrictions imposed by the simulated platform, it acts as a bound on the possible state space on the model. Finally, Subfigure 1.2c shows the computation executed by the simulator. It illustrates the fact that from all the possible behaviors of the restricted state space, the simulator executes only one trace, here contoured and highlighted in yellow. The timestamps $t_0 \ldots t_4$ correspond to the simulated time at which the events corresponding to the transitions happened. It is important to mention that the case of simulation is simplified for the sake of presentation. Here only the communication events are represented, but a simulator can also consider local events such as computations. However, these are ignored by the model checking algorithm by the reasons exposed in the previous section.

Interception

The main task of a distributed systems simulator is to mimic the behavior of the target platform so that the application behaves as if it were actually running on that target. In most cases, the application under study has to be written using a specific API to interact with the simulator, except for tools designed for standardized APIs (i.e. such as MPI). This allows the simulator to *mediate* all the platform specific interactions such as inter-process communications, or operating system services.

The main loop of a simulator usually consists of executing each user processes in a *controlled environment* that provides mechanisms to block them when they initiate an interaction with the platform (e.g. a communication). The resource models are used to determine how long the blocking action will take to finish and what is the increment to the simulation time.

A model checker proceeds similarly. The execution of a transition is the way it has to generate the successor states. As noted previously, these transitions are the calls to the communication APIs. The main difference is that a simulator generates one possible run $\rightarrow$ of the system subject to the restrictions imposed by the models, whereas the model checkers try to explore all possible runs $\rightarrow$.

Controlled Execution Environment

In a distributed scenario, the global state of the system is composed by the local states of every process plus the state of the network. Both simulators and model checkers

require a way to inspect and manipulate the global states to operate, however it is well-known that obtaining a consistent snapshot in a distributed setting is very difficult and has been an active field of research since the 1970s. A suitable solution to this problem is relying on a virtualization of the distributed environment. Since simulation allows to emulate the entire system within a single address space, then working with global states becomes trivial.

Particularly to model checking, when the exploration leads to a final state (or hits the exploration bound), it is necessary to return to a previous execution state in order to try another execution trace. The use of a single address space greatly simplify the implementation of such a rewind-like mechanism.

1.4 The SimGrid Simulation Framework

The SimGrid framework [14] is a collection of tools for the simulation of distributed computer systems. It was designed as a scientific measurement tool, and as such it uses validated analytical models to compute realistic (and reproducible) simulations. There are already 60 external articles published that are based on SimGrid as the experimentation platform.

SimGrid is the technical context of this work, hence it is important to understand its architecture and implementation in detail. Some of the contributions of this thesis consist of the analysis and modification of SimGrid's design to enable the parallelization of certain parts of the simulation loop, and to simplify the integration of the state exploration algorithm. In this section we present SimGrid's experimentation workflow, its architecture, and the main loop of the simulation.

1.4.1 Workflow Overview

SimGrid simulates real executable implementations of distributed algorithms over virtual computing platforms. To compute a simulation $\mathrm{S}(\mathrm{P}, \mathrm{R}, \mathrm{M})$ it requires the user to provide the program, that we represent as a set of processes P, the description of the resources (platform) R, and the models for these resources M. We now provide a detailed description of each of these parameters :

The application to simulate (P). The program must use one of the communication APIs supported by the simulator, and written in a language with bindings like C or Java.

SimGrid offers three communication APIs : MSG, SMPI and GRAS. Each API is tailored to ease the use of the simulator in a given context, and they should be viewed as syntactic sugar to simplify the user experience. MSG is inspired by CSP, processes communicate through mailboxes that function as channels. SMPI stands for Simulated MPI, and implements a subset of the MPI library to enable the simulation of existing MPI programs without modifications. GRAS stands for Grid Reality and Simulation, and provides a socket-like communication interface that enforces the design of applications with an event-driven approach.

The platform ($\mathbb{R}$**).** This comprises all the experimental setup, such as the description of the resources to simulate (hosts, links, the network topology, and routing information). It also includes the external workload that the platform might experience during the experiment, such as the background traffic on the communication links, or host failures. Finally, it includes the deployment information that describes how the processes that compose the application should be started and in which nodes of the platform.

The resource models ($\mathbb{M}$**).** These are the models of the hardware resources, such as CPUs or network links. They are used to compute the sharing of the resources of the platform among the events issued by the processes of the application. The user must choose and configure one of the provided models for each kind of resource, or alternatively he/she can provide custom ones.

Provided these three parameters, the simulation is then performed by executing the distributed application in a controlled environment, where the simulated processes are associated to the hosts in the virtual platform using the deployment information, and the events of the application that interact with the platform are intercepted to be used by the models to compute the timestamps through which the simulated time advances.

1.4.2 SimGrid Architecture

The architecture of SimGrid is presented in Figure 1.3, and can be divided in three functional layers presented here from bottom to top :

1. **The simulation core (SURF).** This module has an abstract view of the simulation where there are only resources arranged in a certain topology and actions that compete to consume these. It provides the analytical models of the hardware and

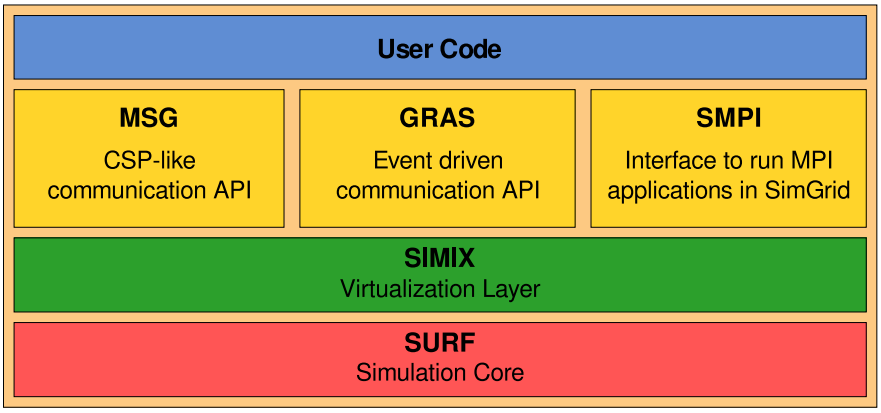


FIGURE 1.3: SimGrid's Layout.

uses them to compute the timestamps that are the result of sharing of the resources in the virtual platform among the actions.

2. **The execution environment (SIMIX).** It introduces a virtualization layer that offers services like processes, synchronization primitives and scheduling control. It is the link between the user processes and the simulated world. Both contributions of this thesis, the model checker and the parallelization work, are centered around this module.

3. **The communication APIs (MSG, GRAS, SMPI).** These modules provide user-level communication (IPC) facilities to implement the distributed algorithms. Internally, they intercept the messages exchanged by the processes and report them to the simulation core (SURF). The messages are delivered only after the core decides that simulated time advanced the right amount.

1.4.3 Simulation Algorithm

The Algorithm 2 shows the pseudocode of SimGrid's main loop located in the SIMIX module, which is a refinement of Algorithm 1 of page 52. The variable time represents the simulated time, and $\mathrm{P}_{\mathrm{time}}$ is the set of processes ready to be executed at time (the *scheduling set* corresponding to the timestamps time). Processes not ready at any given time are blocked waiting for the end of a specific action, such as a message delivery or reception. At every iteration, the virtualization layer (SIMIX) schedules the processes in the set $\mathrm{P}_{\mathrm{time}}$ (line 4). Each of them executes until an interaction with the simulated platform is required, like sending a message, at which point SIMIX blocks them and creates the respective action in the simulation core (SURF). Once all the processes are blocked,

the loop calls the simulation core (line 5) to compute and advance the simulated time to the next earliest ending actions (the next timestamp). In return, it gets the list of finished actions which it uses to compute the next scheduling set (line 6). The simulation is over when there are no more processes to run, i.e. when all processes have terminated, or when they are in a dead-lock situation.

Algorithm 2 Main Loop in SimGrid (SIMIX)

1: time $\leftarrow 0$
2: $\mathrm{P}_{\mathrm{time}} \leftarrow \mathrm{P}$
3: **while** $\mathrm{P}_{\mathrm{time}} \neq \emptyset$ **do**
4: schedule($\mathrm{P}_{\mathrm{time}}$)
5: time $\leftarrow$ surf_solve(&done_actions)
6: $\mathrm{P}_{\mathrm{time}} \leftarrow$ process_unblock(done_actions)
7: **end while**

From the implementation point of view, SimGrid runs the entire simulation as a single process in the host machine. To achieve this, the virtualization layer (SIMIX) folds the user's processes into *execution contexts* composed by a stack and storage space to save the CPU state (registers). The simulator itself runs in the default execution context of the process in the host machine, that we call *maestro*. It is responsible of executing the computations of the core (SURF), and it controls the scheduling using subroutines that swap execution contexts.

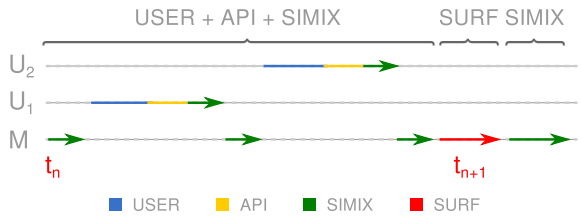


FIGURE 1.4: Simulation Main Loop in SimGrid

Figure 1.4 depicts a macroscopic description of the execution contexts of the modules in SimGrid for one iteration of Algorithm 2, where U_1, U_2 are the contexts of the user processes, and M is the maestro context. The loop schedules sequentially U_1 and then U_2 until they block waiting for their actions to finish (USER+API+SIMIX). Then, it calls SURF to share the resources and compute the time of the next earliest ending actions, in this case $\mathrm{t}_{\mathrm{n+1}}$. The iteration finishes when SIMIX determines the scheduling set of contexts to schedule in the next iteration of the loop.

It is interesting to recall the difference between wall-clock time and simulated time. In Figure 1.4 the simulated time advances discretely on each call to SURF and remains constant during the execution of the scheduling sets, while the wall-clock time advances normally.

A Detailed View of the Architecture

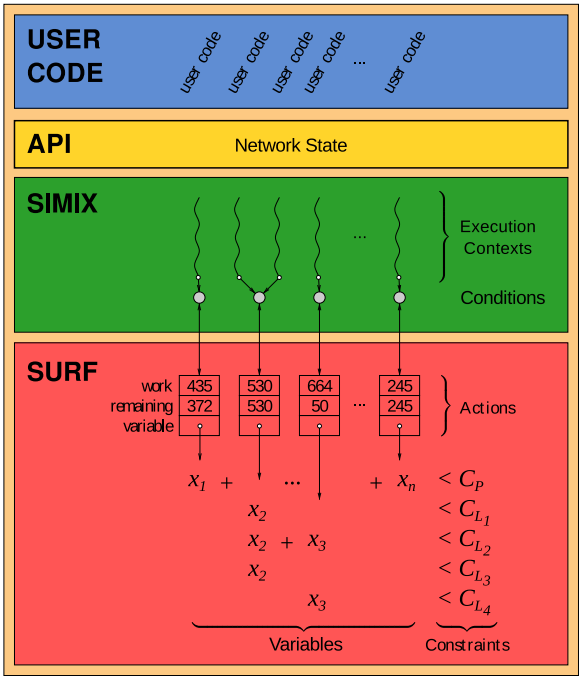


FIGURE 1.5: Internals of SimGrid

Figure 1.5 shows a detailed view of the internals of SimGrid, illustrating the location of the different elements involved in a simulation. Starting from the top, the user code relies on the APIs to interact with the simulated platform. SIMIX provides the process abstraction to run the user code using execution contexts. The conditions are the link between the user processes and the simulated world. They function as classic condition variables that block the waiting process, but instead of releasing them on signals, they are associated to the actions of the simulation core in the layer below. Each action in SURF represents an interaction of the program with the platform, and it has the information about the total amount to consume of a given resource, how much of the action remains to be done to finish, and a link to a variable in the model of the platform. SURF provides many possible abstract representations of the platform, in this example we show a Linear Max-Min (LMM) model. The LMM consist of a system of equations one for each resource

in the platform, in this case 1 CPU and 4 network links. C_P is a constraint that denotes the maximum computing power of the CPU, and the constraints $C_{L_1} \ldots C_{L_4}$ represent the bandwidth of each link. There is one variable for each action in the corresponding equation of the action's resource. Finally, the simulator advances by computing solutions of the LMM system, that correspond to possible sharing of the resources.

1.5 Detailed Objectives of This Thesis

In the previous sections of this chapter we have provided a detailed state of the art in the areas of distributed systems, simulation, and model checking. We are now in a position to recapitulate in more depth the objectives of this work, that were briefly given in the introduction. The work of this thesis is directed towards two main goals :

Correctness and performance assessment. Nowadays it is becoming equally important to attain the functional correctness of a distributed system as well as its runtime performance. Therefore, we aim to close the gap between the correctness and performance assessment of distributed systems. For this, we explore the idea of extending the SimGrid simulation framework capabilities to enable the model checking of *unmodified* distributed programs written for the framework. This would avoid the construction of separate models or prototypes for the same application just to assess the correctness or the performance, allowing to apply both techniques at any point of the development process.

In particular, we focus on exhaustively exploring the nondeterministic behavior of the systems, caused by the asynchronous execution of its components. We use a dynamic verification approach, as it is known to be well suited for dealing with source code implementations of asynchronous systems. From the technical point of view, we plan to replace the SURF module with a *model checking engine* capable of exploring all the relevant interleavings of the processes in the system being verified. The effectiveness of the model checker depends on its ability to deal with the state space explosion. In this work we use a dynamic partial order reduction method that is known to be effective in combination with dynamic verification. However, we adapted this method to be able to handle programs using any of the communication APIs in SimGrid.

Scalability of CPU bound simulations. Distributed systems are constantly increasing in size and complexity. Applications that rely on peer-to-peer architectures with millions

of interacting nodes are now common thanks to the wide availability of fast Internet access to home users. Moreover, High Performance Computing has become the third pillar of science and engineering thanks to its ability to simulate complex phenomena and thus anticipate their behavior. Optimizing these applications is of extreme importance for the real world economy.

Simulation is one of the most widely used approaches, as it allows to accurately predict the performance of a distributed system, without dealing with the complexity of the real distributed execution platform, and even if this platfrom is not available to the developer. In SimGrid, simulations can be either memory bounded or CPU bounded. In general, the spatial limitations of the simulations can be overcome by increasing the amount of memory in the workstation running the simulator, however this is not the case for CPU bounded simulations. The problem of simulating programs is inherently sequential, therefore it is very hard to make it scale with current multi-core architectures, that increase almost exclusively in parallel computing power.

To solve this problem, in this thesis we explore a parallelization approach that keeps SimGrid's simulation loop sequential, but it parallelizes one of its steps. From the technical point of view, it consist in parallelizing the execution of the user contexts shown in Figure 1.4 (USER+API), while keeping the other steps sequential. The key challenge of the approach is to reduce and encapsulate the shared state of the simulation, to minimize the synchronization costs among the user processes. Moreover, it is important to understand in what situations this can result in a speed up. Therefore we analyze and compare the complexity of the sequential loop with the parallel one, to obtain a criterion that can help the user to choose between a purely sequential simulation and a parallel one.

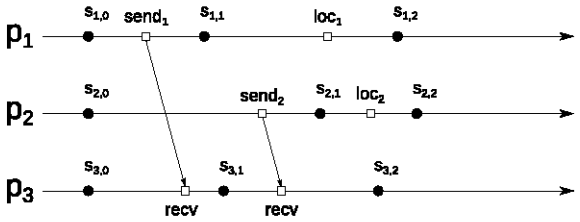
Chapitre 2

Bridging Efficient Simulation & Verification Techniques

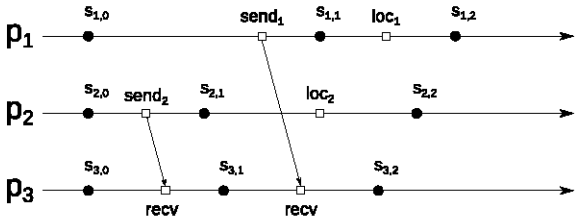
The overall objective of this thesis is to develop a unified framework for the study of both performance and correctness aspects of distributed systems. To this end, we propose two lines of work. The first towards the verification of distributed systems, that consists of extending the functionality of the SimGrid simulation framework to explore the inherent nondeterminism of the distributed programs that it simulates. The second line of work seeks to push the scalability of SimGrid to enable the simulation of systems composed of millions of processes. For this, we explore a parallelization approach to exploit the continuously increasing availability of multi-core architectures.

The first step towards a unified framework is to analyze the current design of SimGrid with respect to the new requirements introduced by the model checker and by the parallelization of the simulator. In this chapter we show that with the current architecture it is almost impossible to implement the new functionality in a proper and efficient way. The central issue with SimGrid's architecture is that it is limited by design decisions regarding the handling of the simulation state that do not correctly address the new requirements. Hence, we propose a novel design that builds from the observation that the services offered by the SIMIX module are similar to those provided by an operating systems (OS), providing a guideline to architect a system capable of simulating in parallel the user processes and performing explorations of the nondeterministic behavior of the programs.

The sections are organized as follow. Section 2.1 explains why the shared state of



(a) Run 1



(b) Run 2

FIGURE 2.1: Two possible runs of the same program.

a system is very important for both dynamic verification and the parallelization of the simulator. Then Section 2.2 presents an analysis of the limitations of SimGrid's design with respect to the shared state. Next Section 2.4 introduces SIMIX v2.0, a new virtualization layer that replaces the SIMIX module and addresses the shortcomings regarding the shared state, allowing both the parallel execution of the user code, and its verification through model checking. Finally we show that the interaction with the operating system of the host is crucial for the performance of the simulators.

2.1 Importance of the Shared State

The shared state of a system is a central notion for both state space exploration and parallelization. A model checker systematically generates runs ($\rightarrow$) of the distributed program to check all possible interleavings of the processes in the system. For this, it requires control over the state of the network, the only shared state among processes. The order in which processes perform communications, affecting the shared network state, induces a corresponding run, and therefore partial order $\rightarrow$ of events.

To illustrate this, consider a program with processes p_1,p_2, and p_3 where processes p_1 and p_2 each send a message to p_3 and then perform some local computation, and process p_3 on the other hand, expects to receive two messages from any process. The execution

of this program can lead to two different runs shown in Figure 2.1. In the first, p_1 issues its send (send_1) before p_2 (send_2) and thus it matches the first receive of p_3 (recv), forcing send_2 to match the second receive as shown in Figure 2.1a. The other alternative is shown in Figure 2.1b. Here, the order is inverted and p_2 issues send_2 before p_1, matching the first receive and forcing p_1 to match the second, resulting in a completely different run $\rightarrow$.

We can infer from this, that a requirement for the model checker is to have a mechanism to intercept the networking operations *before* they modify the state of the network. To steer the execution of the program and cover every possible interleaving, the modifications of the network state should be postponed until all the processes announced their intended communication operations, to consider all the matching possibilities.

The shared state is equally important for the parallel execution of the user processes during simulation. These can interact with the simulated platform at any point of the execution, and in many cases these actions involve modifying shared portions of the simulation state, as previously shown with the example of Figure 2.1. Therefore, to maintain the coherence of the parallel simulations, all potential operations on the shared state should be atomic. Moreover, to guarantee the reproducibility of the simulations these operations must always happen in the same order, otherwise different interleavings might change the matching of source and destination of messages and thus result in different runs. Again, the requirements concerning of the shared state become a critical point to achieve the proposed goals of this thesis.

2.2 The Shared State in SimGrid

We now analyze the current architecture of SimGrid with respect to the requirements presented in the previous section. We show that the main shortcoming of its design is the lack of encapsulation of the simulation's shared state, that is scattered across the whole software stack and thus difficult to control and manipulate. It is important to recall that according to SimGrid's original requirements this is not a problem at all. The limitations become apparent only after considering the possibility of executing the user processes in parallel, or when trying to execute all possible runs of a program as required for the model checker. In the following subsections we discuss in detail these problems.

2.2.1 Dispersed Shared State

The data structures that compose the shared state of the simulation are scattered across the whole software stack. In particular, the network state is deeply disaggregated among the simulation core, the virtualization layer, and the communication API as shown in the Figure 1.5 on page 66. The state of the network's applicative layer is contained in the communication APIs. The state of the execution contexts, the synchronization primitives (i.e the condition variables), and the association of the actions in the simulation core with the condition variables blocking the processes waiting for these are located in the SIMIX module. Finally, the state of the resources, such as the CPUs, or the state of the network's transport layer is handled by SURF.

This is also reflected in the way the communications are handled. To simulate the exchange of a message between two given processes, the networking API first updates the internal state of the network, then it creates the condition variables to block the processes participating in the communication (in SIMIX), next it creates the actions at the simulation core (SURF), associating them with the condition variables to wait until completion of the actions. Afterwards, once the simulation core determines that the actions have finished, the networking code cleans up all the associated data structures. Each of these operations corresponds to a different level of abstraction : the network, the virtualization environment, and the simulation core, however they are all performed by the API code.

Almost all the data structures that the API manipulates are shared across the entire simulation. Under sequential simulation this is not an issue as at any time only one user process is in execution, and thus there is no risk of race conditions. However, this design posses serious difficulties to fulfill the atomicity requirement of the parallel execution. The problem can be clearly visualized in Figure 1.4 in page 65. The execution contexts that run the code of the user, also run the functions of the communication APIs (USER+API+SIMIX arrows) that affect the dispersed shared data structures. If we allow the concurrent execution of these contexts it would require fine-grained locking across the entire software stack to avoid race conditions and maintain consistency. This would be both extremely difficult to get right, and prohibitively expensive in terms of performance. Moreover, each communication API implements its own logic to affect the state of the network. So each API would require its own locking mechanism with little possible logic factorization between APIs. Finally, even if these difficulties were solved to ensure the internal consistency of the simulator, the parallel execution with this design would

hinder the reproducibility of simulations, because the scheduling ordering would vary between simulations, yielding to possibly many different runs $\rightarrow$.

All these reasons explain that distributing the code modifying the simulation's shared state across the execution contexts of the user code clearly hinders the possibility of running these contexts in parallel efficiently.

2.2.2 Lack of Control Over the Executed Run

A model checker has to systematically generate every possible run $\rightarrow$ of the system. As explained in Section 2.1, this requires a transition interception mechanism capable of blocking the processes *before* they modify the network state. This permits to postpone the decisions about the matching sources and destinations of the messages, once every process announced their transitions and thus all the possibilities can be considered.

The main difficulty to implement such a mechanism in SimGrid is again the lack of proper control over the network state. This might be surprising at first, as SimGrid is a simulator and should also provide a mechanism to intercept the communication actions of the processes to simulate the network transfers. However, the problem arises from the slightly different requirement of this functionality. A simulator's goal is to block the processes involved in a communication until the simulation core determines that the corresponding simulated time has elapsed. Therefore, the interception mechanism only provides a mean to delay processes, but not to decide the matching of sources and destinations, that instead is determined by the natural issuing order of the network operations. Note that this perfectly fits the requirements of simulation, recall that the goal is to generate *one possible run* that is subject to the restrictions imposed by the platform, and the current design complies with it.

2.3 Other Considerations of SimGrid's Architecture

2.3.1 API-specific Code

SimGrid provides several communication APIs to the user, each tailored for a particular type of application. Despite the apparent external differences, internally the handshake protocol and its interception mechanism varies little from one user API to another. The differences are minor, and in essence they all consist in a procedure to determine the source and destination of the message, a condition variable to block the involved pro-

cesses, and the creation/destruction of an action in the simulation core. Nevertheless, the communication APIs re-implement the same functionality mixed with code specific to each, leading to an unnecessary complex code base that is hard to modify and maintain. With this scheme, a model checker capable of verifying programs written for one of the APIs of the simulator would require a specific transition detection mechanism.

2.3.2 Assumptions on the Scheduling Order

SimGrid is designed as a scientific measurement tool, and strives to provide deterministic and reproducible simulations. When non-deterministic executions are possible, SimGrid always takes the same choices for the sake of reproducibility. However, the lack of variation of the scheduling order has lead to an implementation that is sensitive to it. The code assumes in many places, specially in the communication APIs, that certain events always execute in the same order. This is clearly not true when executing the user code in parallel or when performing a state space exploration where all possible schedules are considered.

2.4 SIMIX v2.0

In this section we introduce SIMIX v2.0, a new virtualization layer that replaces the SIMIX module to address the requirements of dynamic verification and parallelization. SIMIX v2.0 aims to encapsulate the simulator's shared state, and provides a mechanism to guarantee that all potential modification operations on this shared state are race free. These two characteristics greatly simplify the implementation of both the parallel user code execution under simulation, and the model checking exploration algorithm. The design of SIMIX v2.0 starts from the observation that the services required by the user processes from the simulation stack are similar to those provided by any *classical* operating system (OS) : processes, inter-process communication, and synchronization primitives. Therefore it is reasonable to design a new virtualization module that incorporates and encapsulates these three services to solve the aforementioned shortcomings on the shared state.

2.4.1 Strictly Layered Design

The modules shown in Figure 1.3 exist only to group functional entities, their interfaces are public to the entire software stack (except for the user code). Because of this, the networking APIs are designed assuming that they can call any function from any module regardless of the level of abstraction.

SIMIX v2.0 follows a strict layered design that abstracts away all the details of the simulation core and the virtualization environment from the higher-level APIs. In the new architecture SIMIX v2.0 acts as a virtual operating system that mimics the services that would be available to the application if executed in the real life : processes, synchronization primitives, and inter-process communication. The first two were already part of SIMIX, but the key observation is that IPC is tightly related to these and should be integrated in the same module.

We now give a general overview of the IPC characteristics, and its full formal semantics is presented in Section 3.9. SIMIX v2.0 provides IPC through a basic set of networking primitives whose communication model is built around the concept of *mailbox* (or *rendez-vous* point). The mailboxes are meeting points to synchronize the processes participating in a communication. These have no owners, they exists as independent entities representing the network state and they can be directly accessed by any process at any time. The API provides just four operations *Send*, *Recv*, *WaitAny* and *TestAny*. The first two are the only ones that operate directly on the mailboxes, and queue a send or receive request into them, returning a communication identifier. A communication request consists of a record with a type field indicating if it is a send or receive operation, plus several fields with the source/destination buffers, data size, etc. A communication can happen only between matching requests. We say that two request *match* when they are of different types, hence a *Send* matches any *Recv* for the same mailbox, and vice versa. The operation *WaitAny* takes as argument a set of communication identifiers and blocks until *at least one* of them has been completed. Finally, *TestAny* also expects a set of communication identifiers and checks if any of these communications has already completed ; it returns a Boolean result and never blocks.

Figure 2.2 illustrates the four steps of a communication that sends data from buffer &x in process A to buffer &y in process B using the new IPC in SIMIX v2.0. The communication starts with process A issuing a $Send(\&x)$ to a mailbox (that in this example is empty). This creates a new request record $[id, \text{``ready''}, A, \_, \&x, \_]$ that is queued in the mailbox. The value *id* is a unique identification number returned to A. Next, process B

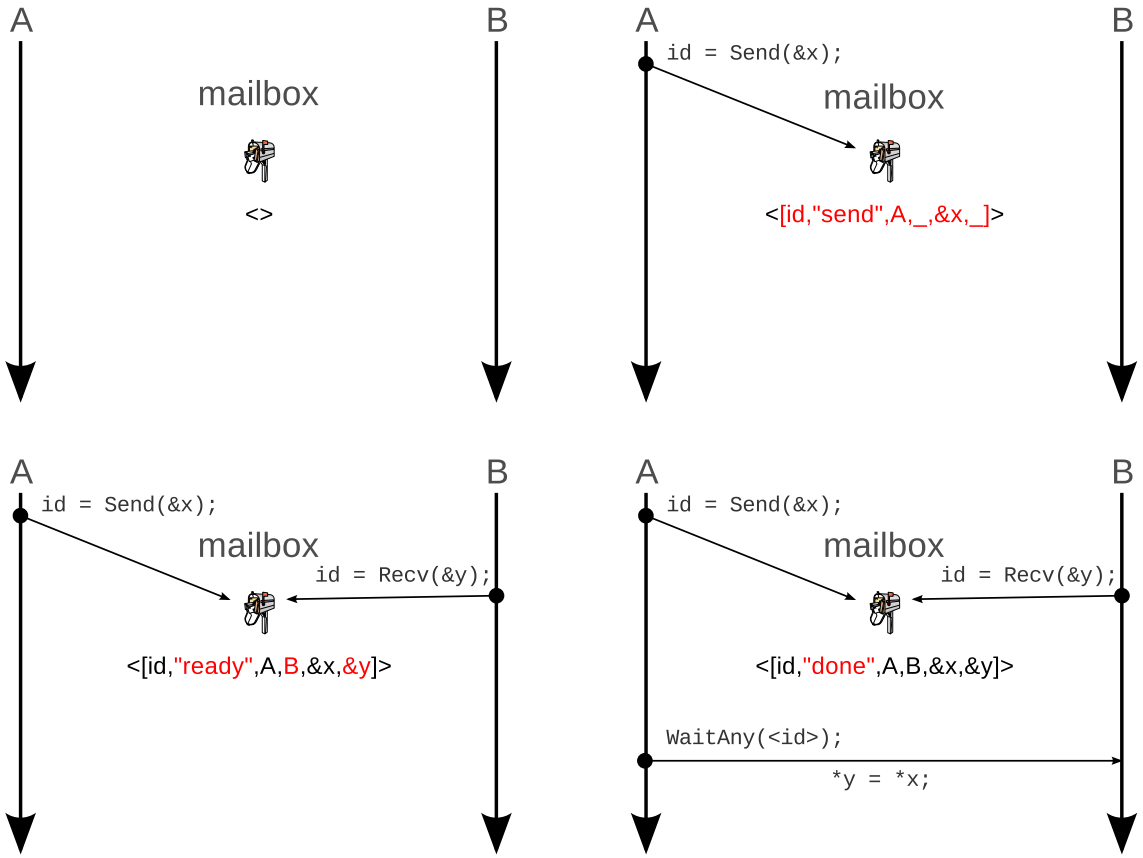


FIGURE 2.2: The four steps of a communication.

issues a *Recv*(&*y*) call to the same mailbox, that finds the matching request from process A. This results in a request that has all the required information to happen, thus the record is updated to the ‘‘ready’’ state, and the actual simulation action is created at the core. Finally, process A issues a *WaitAny*(⟨*id*⟩) that blocks it until the simulation action finishes. At that instant, the data from the source buffer &x in process A is copied to the destination buffer &y in process B.

Despite the simplicity of this networking model, it is expressive enough to implement the existing user-level communication APIs on top of it. The mailbox concept provides enough flexibility to allow many-to-many communication patterns needed for group communication, but it can also be restricted to many-to-one, or one-to-one schemes by adding restrictions in the higher-level APIs. Moreover, the *Send* and *Recv* operations are only involved with determining of the source and destination of the messages, whereas the *WaitAny* and *TestAny* are responsible for the actual data transfers. This proper split of functionality gives a fine-grained control over the network state that permits to express a CSP-like API (MSG), a socket-based API (GRAS), and a subset of the MPI library (SMPI).

The layered design of SIMIX v2.0 has other advantages over the previous architecture besides the encapsulation of the shared state into a single module. It also solves the problem of code duplication, as the hand-shaking procedure of the communications, that is almost identical in all of the user-level APIs, is now factorized out into the IPC mechanism. More importantly, the four networking primitives are the only way to modify the shared state that determine the run to execute (and the $\rightarrow$ relation), hence this simplifies the instrumentation of the exploration algorithm that now has to interact with a single module. Finally, the close integration of the IPC with the synchronization primitives and the control of the scheduling, significantly reduces the complexity of the code and therefore the confidence in its correctness, and its performance.

2.4.2 Kernel-mode Emulation

In the previous subsection we addressed the problem of the dispersed shared state by enforcing a layered design that encapsulates it. This is the first step towards an architecture flexible enough to allow both parallel simulation and state space exploration. We now address the second part of the problem, related to the concurrent access and modification of this shared state.

A distinctive characteristic of most modern operating systems is the use of a system call interface to provide services that the program does not normally have permission to run. The user processes execute in a virtual address space, isolated from the rest, and the system calls constitute the only interface with the rest of the system. To communicate with other processes, they have to request the intervention of the kernel that runs in a special supervisor mode with a complete view of the system state. This clear separation between the user process on one side, and the kernel on the other, permits the independent and parallel execution of the processes, as all the potential access to the shared resources is mediated by the kernel, which is responsible for maintaining the coherence.

Inspired by these ideas, in SIMIX v2.0 we introduced a new layer that emulates a system call interface, that we call *requests*. It completely separates the execution context of the user code from the simulator code (maestro), ensuring that the shared state of the simulation can only be accessed and modified from the maestro execution context. When a process performs an action that requires the interaction with the platform (like executing a computing task or sending a message), it issues the corresponding request through the interface, and then blocks until the answer is ready.

The main loop of SIMIX v2.0 is shown in Algorithm 3. The difference with the previous

Algorithm 3 SIMIX v2.0 Main Loop.

```
1: time ← 0
2: P_time ← P
3: while P_time ≠ ∅ do
4:    schedule(P_time)
5:    handle_requests()
6:    time ← surf_solve(&done_actions)
7:    P_time ← process_unblock(done_actions)
8: end while
```

one (Algorithm 2) is in line 5, point at which the requests issued by the processes that executed in line 4, are handled by the simulator.

Figure 2.3 outlines an iteration of the main loop from the execution context point of view, with two user contexts U_1,U_2. The loop starts in line 4, where now the scheduled contexts only execute the user code (in blue) and the user-level API (in yellow). After the changes presented in the previous subsection, the API does not modify the network state anymore and instead it relies on the IPC mechanism in SIMIX v2.0. The contexts run until an interaction with the simulated environment is required, action that issues a request holding the information about the nature of the action, (i.e if it is a send/recv communication or a computation). Once all the contexts finished their executions, the SIMIX v2.0 layer that runs in the maestro contexts, proceeds to handle all the requests sequentially (in line 5). This is a very important difference, as now it is the point where the all the modifications to the shared state happen. For example, the handlers corresponding to the networking request decide which are the matching send/recv communications, and create the simulation actions at the core. The rest of the loop remains unchanged, in line 6 SURF computes the next timestamp and the list of actions that finish on it, and finally in line 7 SIMIX v2.0 generates the scheduling set for the next iteration.

The main difference is that now, the user contexts only execute the user processes and the portions of the user APIs that do not involve modifying the shared state.

The proper split enforced by the request interface together with the strict layered architecture, overcome most of the limitations that the old design has to address the requirements of the parallelization and the state space exploration. Each request is an interception point *prior* to the modification of the shared state, and the request handlers encapsulate all the manipulations on it. Because these execute sequentially, the request mechanism removes the need for the fine-grained locking scheme, and allows a simpler

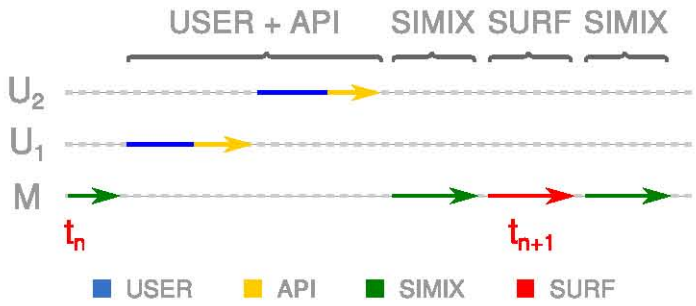


FIGURE 2.3: Simulation Main Loop in SimGrid v3.6.

parallelization of the user code.

Moreover, this simplifies the integration of both the simulator and the model checker, as each can provide different request handlers to perform different tasks according to the desired functionality. For simulations, the handlers should enforce determinism in the matching of communication requests (to produce results that are reproducible), and should create the actions in the simulation core. In the case of model checking, the handlers can determine the matchings of communications according to the state of the exploration stack, giving a precise control of the run → being executed.

2.5 Mastering the Operating System

System programming design practices are not only useful as a guideline for the architecture of a simulator, they are also important to optimize its performance. The interaction of the simulator with the operating system of the host involves issuing expensive system calls that might go unnoticed at first, but they produce a performance hit when scaling up to larger simulation instances.

An interesting case is the virtualization for the user processes, that is in the critical path of the simulator. As mentioned before, the user code runs in execution contexts that emulate a cooperative multitasking environment entirely in user space. Using the POSIX's `ucontexts`, the execution is transferred from one `ucontext` to another using a `swapcontext` function, which is passed a pointer to the stack to restore and a pointer to a storage where the current stack should be saved. At the first glance, this function runs entirely in user space without requiring intervention of the OS kernel, but it turns out that this is not true. Indeed, POSIX allows to specify a different signal mask for each `ucontext`, which induces an operation involving a system call during the swap. Since we do not need this feature, SimGrid now offers an alternative `ucontext` implementa-

tion that is free of system calls. Because the swap routine modifies specific registers, it is architecture dependent and it has to be programmed in assembly language. For the moment this option is only available for x86 and x86_64 hardware, and other architectures fall back to the standard `ucontexts`.

2.6 Performance Comparison With Previous Version

It is reasonable to think that the modifications introduced in this chapter can affect the simulator's performance. After all, state encapsulation usually increases the overhead to access it, and in the case of SimGrid, we introduced a new layer named requests to mediate any interaction with the simulated platform. In this section we try to compare the performance of a previous version of SimGrid based on the old design, with one that includes the modifications introduced in this chapter. Moreover, we measure the speedup obtained by the custom implementation of execution contexts presented in Section 2.5.

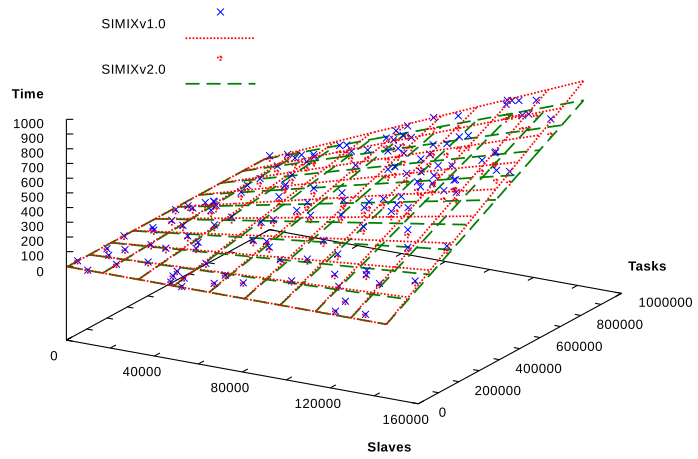


FIGURE 2.4: Master-Slaves experiment

The first experiment compares the running time of a master-slaves example written for the MSG communication API. The *master* process assingns computing tasks to the *slaves* that execute them. The number of slaves, and the number of tasks are parametrized values of the experiment that range from 0 to 160.000, and from 0 to 1.000.000 respectively. Figure 2.4 shows the results of the experiment for the whole range of values of the parameters, using SIMIX(in red) and SIMIX v2.0(in green). On average, SIMIX v2.0 is 14% faster than SIMIX, and there is no loses (always faster).

To measure the speedup obtained by the custom context implementation presented in Section 2.5, we present the results of an experiment that uses a Chord implementation

as the workload for the simulator. We compare the simulation time with the other context alternatives available in SimGrid(`ucontext` and pthreads). The simulation consists of $n = 100,000$ nodes executing during 1000 seconds, and it lasts 471 seconds using optimized contexts vs. 582 seconds using `ucontexts`. Operating system limitations make it impossible to run this experiment using pthreads. However, downscaled versions of the experiment show an order of magnitude slowdown for pthreads over our optimized contexts. The gain obtained with our custom implementation over `ucontexts` is relatively constant around 20%, showing the clear benefit of avoiding unnecessary system calls on the critical path of the simulation.

2.7 Summary

In this chapter, we presented several improvements to the SimGrid framework that are closely related to the handling of the shared state of the simulations. We have shown that in the old design it was almost impossible to instrument an effective state exploration algorithm, and that the efficient parallel execution of the user code was hard to achieve, due to the segregation of the shared state. We proposed a new architecture that solves these issues following the design guidelines of operating systems. It abstracts all the modifications of the shared state using an emulation of the system call mechanism called *requests*. Additionally, we showed the importance of understanding the interaction of the simulator with the host's operating system, that can lead to noticeable performance improvements in typical simulations. The experimental results show that the new design improves the performance of simulations while increasing the functionality provided by the layer. SIMIX v2.0 has better performace due its simpler design, and because many dynamic data structures were replaced by static counterparts. This initial preparation work, lays the foundation for the contributions that follow : the model checker, and the parallelization of the simulation.

Chapitre 3

SimGridMC : A Dynamic Verification Tool for Distributed Systems

The simulation of distributed systems is typically targetted towards the analysis of resource usage and performance optimization in particular scenarios of interest to the user. Model checking, on the other hand, attempts to detect unwanted behaviors of distributed systems by exhaustively exploring their state space.

In this chapter, we present SimGridMC a distributed systems model checker integrated into the SimGrid framework that complements the existing simulator functionality by allowing the user not only to simulate distributed systems, but also to verify their implementations. The objective of SimGridMC is to enable the application of model checking to the software that can be run in the simulator. Hence, SimGridMC verifies the implementations of the programs *without requiring any modifications to their code.*

We first present an overview of SimGridMC and show how it integrates into SimGrid. Then we explain the constitution of the model, followed by a description of the properties supported by SimGridMC. Moreover, we discuss the design decisions of the exploration technique, showing how it integrates into the existing infrastructure. Next, we present our solution to address the state explosion problem, that is based on dynamic partial order reduction. The effectiveness of this technique is related to the accuracy with which can be determined the dependency between transitions of the model, that in the case of SimGrid correspond to the networking APIs. To this end, we introduce a formal model of these transitions, written in the TLA$^+$ specification language. With this specification we derive an (in)dependency predicate that is used by the dynamic partial order reduction algorithm in SimGrid. Finally, we present a few verification experiments using

programs written for two different communication APIs supported by SimGrid, showing the ability of our approach to use a generic DPOR exploration algorithm for different communication APIs through an intermediate communication layer.

3.1 Overview

SimGridMC is a model checker that aims to find errors triggered by the nondeterministic behavior of the distributed systems written for the SimGrid simulation framework. It is designed with simplicity in mind, thus it is capable of verifying the same program code used for simulations without modifications. For this, it follows a dynamic verification approach that explores all possible executions of the program being verified. This avoids the need for an expensive and complex model building phase, allowing to use the model checking functionality at any point of the development process.

The dynamic approach is particulary well suited for dealing with the nondeterminism that arises from the asynchronous execution of the processes composing the program. However has its limitations compared with other abstract model checking approaches. For example, it can only achieve full verification with finite state spaces, but often distributed programs have an infinite number of states as these might not terminate. Nevertheless, we consider SimGridMC as a debugging tool that is most useful when it succeeds in providing counter-examples that lead to erroneous states, rather than when achieving full verification.

At a high level, SimGridMC takes a standard SimGrid program as input with local assertions inserted at relevant points in the code, which allow to express safety properties. Provided these, then it systematically explores the state space of the program to check the validity of the assertions.

Figure 3.1 shows how SimGridMC is integrated into the SimGrid architecture (see Figure 1.3). It replaces the SURF module, that provides the functionality for resource simulation not needed for model checking, with a state exploration algorithm (MC in the figure), that exhaustively generates the executions arising from all possible nondeterministic communication choices of the applications.

From a technical point of view, the refactoring work presented in Chapter 2 layed the ground for a seamless integration of the model checker into SimGrid. The simulator already includes the infrastructure for managing the scheduling of processes, and for intercepting the communication actions that affect the shared state (requests). Then, the

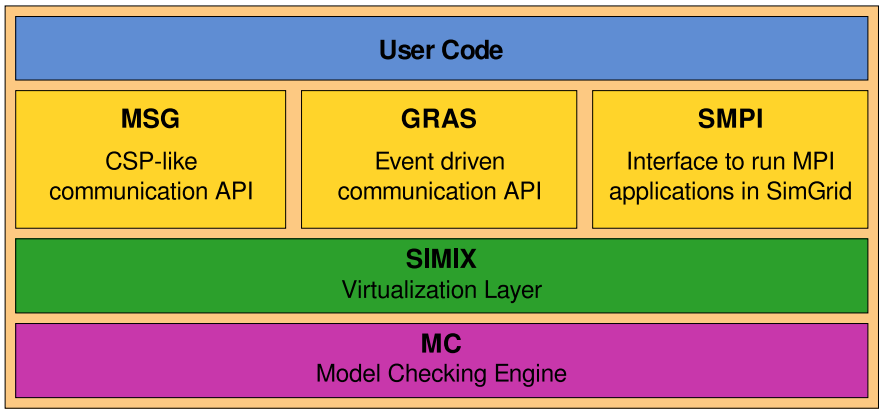


FIGURE 3.1: SimGridMC modules.

implementation work consisted of designing an efficient exploration algorithm, that instead of scheduling the processes according to the computations of the simulation core, it schedules them covering all *relevant* permutations of the communication actions.

3.2 The Model

As explained in Section 1.1.3, the model of a distributed system consists of asynchronous autonomous entities that interact through a shared state that represents the network (no global clock and no shared memory). The global state of the model is composed of the local state of every process plus the network state. The behavior of the model can then be described by its state graph. The nodes represents the different global states and the directed edges (transitions) represent the atomic steps between states that correspond to communication actions.

In SimGrid, the state of each running process is determined by its CPU registers, the stack, and the allocated heap memory. The network's state is the only shared state among processes, and it is given by the messages in transit, that comprise the communication requests queued in the mailboxes, and the source and destination buffers of the processes involved in the requests.

The only way a process can modify the shared state (the network) is by issuing calls to the communication APIs, thus the model checker considers these as the only relevant transitions. A process transition as seen by the model checker therefore comprises the modification of the shared state, followed by all the internal computations of the process until the instruction before the next call to the communication API. The state space is then generated by the different interleavings of these transitions ; it is generally infinite

even for a bounded number of processes due to the unconstrained effects on the memory and the operations that processes perform.

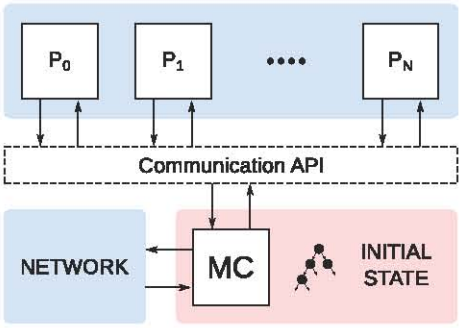


FIGURE 3.2: SimGrid MC Architecture.

Figure 3.2 illustrates the architecture of SimGrid MC. Each solid box labeled P_i represents a process in the distributed system being verified. The exploration algorithm labeled MC executes isolated from the processes. It intercepts the calls to the communication API (dashed box) and updates the state of the network. The areas colored blue represent the system being explored, the area colored red corresponds to the state of the model checker, which holds the exploration stack plus some other information that is detailed in the following sections.

3.3 The Properties

The properties describe the expected behavior of the model. SimGridMC only supports *safety* properties that take the form of local assertions inserted in the code of the program. These are standard expressions of the programming language used to write the program, and as such are subject to the scope constraints. An important implication of this, is that assertions can only predicate on the variables of a *single* process, and hence it is not possible to verify properties that relate states of different processes. This restriction is important at the moment of ensuring the soundness of the exploration algorithm presented in Section 3.7. Nevertheless, it is possible to work around some scoping limitations for *local variables* using global references, that are available at any point in the execution. Then, the value of an assertion depends on a single state, and thus it can be evaluated using a function call. Finally, the verification of deadlock freedom is done automatically by the exploration algorithm.

3.4 Explicit-state

Because the global state contains heaps whose structure cannot be easily analyzed, and the transition relation is determined by the execution of C program code, it is impractical to represent the state space or the transition relation symbolically. Instead, SimGrid MC is an explicit-state model checker that explores the state space by systematically interleaving process executions in depth-first order, storing a stack that represents the schedule history. Each entry of the stack contains a set of processes to interleave, a set of processes already explored, and the outgoing transition executed in that state. As the state space may be infinite, the exploration is cut off when a user-specified execution depth is reached. Of course, this means that error states beyond the search bound will be missed, however SimGridMC ensures complete exploration of the state space up to the search bound.

3.5 The Exploration

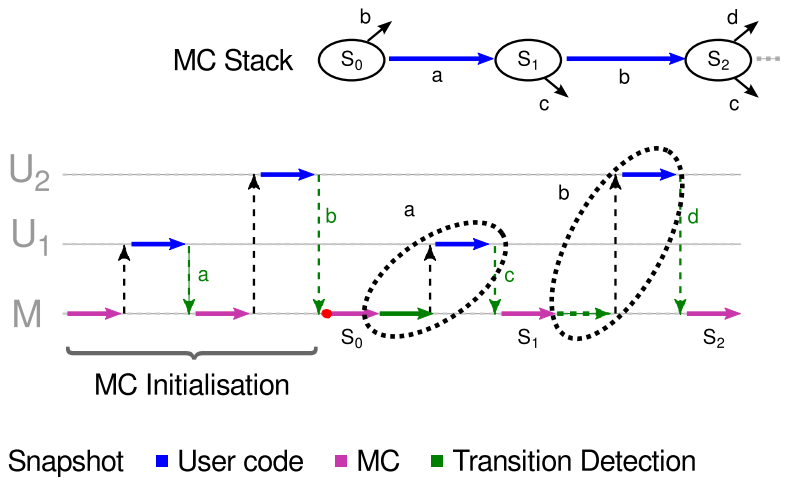


FIGURE 3.3: State Exploration by SimGrid MC.

Figure 3.3 illustrates the exploration technique used by SimGridMC on the example used in Section 1.4. The model checker first executes the initialization phase, that consists of scheduling all the processes up to (but excluding), their first call to the communication API (dashed green downwards arrows labeled a and b in this example). The resulting global state S_0 is considered the initial state of the exploration, hence it is pushed on the exploration stack, with the two possible detected transitions a and b. Additionally, a

snapshot corresponding to the initial state of the entire system is stored for backtracking purposes (indicated by a red dot in Figure 3.3).

After the initialization phase completes, the model checker chooses one action (say, a) for execution. It commits the changes generated by a to the network, and then schedules the associated process, that continues in the instruction right after a and performs all following local program steps up to, but excluding, the next API call (dashed green arrow c). This execution corresponds to one transition as considered by the model checker, which records the actions enabled at this point and selects one of them (say, b), continuing in this way until the exploration reaches the depth bound or no more actions are enabled ; depending on the process states, the latter situation corresponds either to a deadlock or to program termination.

3.6 Stateless Model Checking and Backtracking

When the depth first exploration of the state space reaches the end of a branch (be it a final state, or the specified depth bound), we need to backtrack to a suitable point in the search history and continue the exploration from that global state. A naïve implementation, would check-point the global system state at every step. The memory requirements and the performance hit incurred by copying all the heaps would reveal prohibitive. Instead, we adopt the idea of stateless model checking originally proposed by Verisoft [26], where the backtracking is implemented by resetting the system to its initial state and re-executing the schedule stored in the search stack until the desired backtracking point.

To backtrack, SimGridMC retrieves the global state S_0 stored at the beginning of the execution, restores the process states (CPU registers, stack and heap) from this snapshot, and then replays the previously considered execution until it reaches the global state from which it wishes to continue the exploration. In this way, it achieves the illusion of rewinding the global application state until a previous point in history. To visualize this procedure, consider again the Figure 3.2. Here, the snapshot of the initial state is stored in the isolated memory area of the model checker highlighted in light red. When a backtracking point is reached, the blue area is overwritten with the copy in the initial snapshot. This allows to rewind the application, but preserving the exploration history intact.

An important drawback of the stateless approach, is that the visited global states are not stored, and hence SimGridMC has no way of detecting cycles in the search history

and may re-explore parts of the state space that it has already seen. Note that even if we decided to checkpoint the system state, dynamic memory allocation would require us to implement some form of heap canonicalization in order to reliably detect cycles (see Section 1.3.3). In the context of bounded search that we use, the possible overhead of re-exploring states because of undetected loops is a minor concern for the verification of safety properties. It would, however, become necessary for checking liveness properties as the algorithm has to detect loops through accepting states.

3.7 Coping With The State Explosion Problem

The main problem in the verification of distributed programs, even using stateless model checking, lies in the enormous number of interleavings that these programs generate. Usually, many of these interleavings are equivalent in the sense that they lead to indistinguishable global states. The state reduction techniques try to exploit the symmetry of the state space to reduce the number of interleavings that must be considered.

The idea of the Partial Order Reduction (POR) algorithms is to examine only a representative subset of all possible interleavings of the distributed processes. The reduction algorithms are correct provided every non-explored interleaving is semantically equivalent with respect to the considered properties to at least one interleaving that has been explored, so that no potential error is missed.

To understand how POR works, consider the model of a distributed system viewed as a set of *happened-before* relations ($\rightarrow$) (see Definition 2 on page 44) one for every possible run, each defining a partial order over the set of local states.

The model checker explores global states and thus generates global traces of the system that are possible serializations of these runs. Because a $\rightarrow$ relation is a partial order, there can be many serializations that differ only in the execution order of concurrent independent transitions. POR *tries* to explore only one serialization for each $\rightarrow$, based on information about which transitions are independent. Two transitions are said to be independent if and only if executing them from any given state in any order yield the same global state, and thus the same $\rightarrow$, as illustrated in Figure 3.4.

Formally, the independence is defined as follows (from [25]). Let Σ be the set of reachable states in the model, then a transition t is a partial function $\mathrm{t} : \Sigma \rightarrow \Sigma$. For any state $\sigma \in \Sigma$ the set of transitions enabled at σ is defined as $\mathrm{enabled}(\sigma) = \{\mathrm{t_i} \mid \mathrm{t_i}(\sigma) \in \Sigma\}$. In other words, $\mathrm{enabled}(\sigma)$ is the set of transitions that can happen in the state σ.

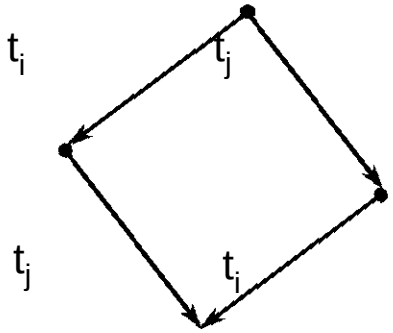


FIGURE 3.4: Two independent transitions $\mathrm{t_i}$, $\mathrm{t_j}$.

Definition 3. *Two transitions* $\mathrm{t_i}$ *and* $\mathrm{t_j}$ *are independent (noted* $\mathrm{I(t_i, t_j)}$*) if*

$$\begin{aligned}\forall \sigma \in \Sigma : \mathrm{t_i, t_j} \in \mathrm{enabled}(\sigma) \Rightarrow & \wedge\ \mathrm{t_i} \in \mathrm{enabled}(\mathrm{t_j}(\sigma)) \\ & \wedge\ \mathrm{t_j} \in \mathrm{enabled}(\mathrm{t_i}(\sigma)) \\ & \wedge\ \mathrm{t_i}(\mathrm{t_j}(\sigma)) = \mathrm{t_j}(\mathrm{t_i}(\sigma))\end{aligned}$$

What Definition 3 expresses is that if two transitions that are independent they always lead to the same state no matter their execution order. Therefore, what POR guarantees is that given two independent transitions, and a safety property to check that cannot distinguish between the two intermediate states, then we can safely explore only one interleaving without loosing the soundness of the verification. In other words, POR avoids exploring portions of the state space that are symmetric with respect to the property being checked. It is important to mention that in the context of SimGridMC, the the first two clauses of Definition 3 are always satisfied, as transitions (that correspond to communication actions) remain enabled until they are executed.

Precisely determining the (in)dependence of transitions can be costly, as it involves evaluating the precise effects of two transitions in either order *for any given reachable state*. In practice, dependence is therefore approximated, and for soundness this approximation has to be conservative in the sense that two transitions should be considered dependent except when one can prove the contrary. Classical POR techniques rely on static analysis to approximate the sets of independent transitions before performing the state space exploration. However, this approach does not work very well with software model checking, because many transition dependencies can only be determined at runtime, forcing the static analysis to produce very bad approximations, and thus yielding few reductions.

3.8 Dynamic Partial Order Reduction

Dynamic Partial Order Reduction (DPOR) aims to solve the problem of computing the independence of the transitions with static techniques by determining the dependency of the transitions dynamically at runtime. They key observation is that dependency can be detected at the end of each run (backtracking point) once the transitions were executed. At that point, all the runtime values of the transitions' arguments and effects on the global state are available, hence it is possible to determine the dependencies. Nevertheless, the overhead incurred by this computation may outweigh the benefits of using DPOR, particularly when working with explicit states as it involves the comparison of parts of the memory of the running processes. A plausible alternative is to rely on a formal framework describing the semantics of the transitions to simplify the mechanism.

The DPOR exploration used in SimGridMC is based on the algorithm introduced by Palmer et al in [49], however it is a bit simpler as we do not check for cycles in the state graph, we are only interested in verifying local assertions, and the absence of deadlocks.

The pseudo-code of the depth-first search algorithm implementing DPOR appears in Algorithm 4. With every scheduling history q on the exploration stack is associated a set *interleave*(q) of processes enabled at q and whose successors will be explored. Initially, an arbitrary enabled process p is selected for exploration. At every iteration, the model checker considers the history q at the top of the stack. If there remains at least one process selected for exploration at q, but which has not yet been explored, and the search bound has not yet been reached, one of these processes (t) is chosen and scheduled for execution, resulting in a new history q'. The model checker pushes q' on the exploration stack, identifying some enabled process that must be explored. Upon backtracking, the algorithm looks for the most recent history $\mathrm{s_j}$ on the stack for which the transition *tran*$(\mathrm{s_j})$ executed to generate $\mathrm{s_j}$ is dependent with the incoming transition *tran*(q) of the state about to be popped. If such a history exists, the process executing *tran*(q) is added to the set of transitions to be explored at the predecessor of $\mathrm{s_j}$, ensuring its successor will be explored during backtracking (if it has not yet been explored). The algorithm is somewhat simpler than the original presentation of DPOR [19] because it assumes that transitions remain enabled until they execute, which is the case for SimGrid.

To prove the correctness of the DPOR algorithm it is first necessary to present the notion of *persistent set* from [70]. A set T of enabled transitions in the state s is a persistent set if their occurrences cannot be affected by executing the transitions not in T, from s. In other words, whatever the system does from s while remaining outside of T, does not

Algorithm 4 Depth-first search with DPOR.

```
1: q := initial state
2: s := empty
3: for some p ∈ Proc that has an enabled transition in q do
4:    interleave(q) := {p}
5: end for
6: push(s,q)
7: while |s| > 0 do
8:    q := top(s)
9:    if |unexplored(interleave(q))| > 0 ∧ |s| < BOUND then
10:      t := nextinterleaved(q)
11:      q' := succ(t, q)
12:      for some p ∈ Proc that has an enabled transition in q' do
13:         interleave(q') := {p}
14:      end for
15:      push(s,q')
16:   else
17:      if ∃ i ∈ dom(s) : Depend(tran(s_i), tran(q)) then
18:         j := max({i ∈ dom(s) : Depend(tran(s_i), tran(q))})
19:         interleave(s_{j-1}) := interleave(s_{j-1}) ∪ {proc(tran(q))}
20:      end if
21:      pop(s)
22:   end if
23: end while
```

interact or affect T.

Definition 4. *A set* T *of transitions enabled in a state* s *is persistent in* s *if and only if, for all transitions* $\mathrm{t} \notin \mathrm{T}$ *such that there exists a sequence :*

$$\mathrm{s} = \mathrm{s}_0 \xrightarrow{\mathrm{t}_0} \mathrm{s}_1 \xrightarrow{\mathrm{t}_1} \mathrm{s}_2 \ldots \xrightarrow{\mathrm{t}_{n-1}} \mathrm{s}_n \xrightarrow{\mathrm{t}_n = \mathrm{t}} \mathrm{s}_{n+1}$$

whose last transition is t *and including only transitions* $\mathrm{t}_i \notin \mathrm{T}$, t *is is independent (see Definition 3) with respect to all transitions in* T.

The concept of persistent set is important to partial order reduction because it yields a subset of enabled transitions in a given state that should be interleaved in all possible permutations, while preserving the local assertions and deadlocks in the reduction [70]. Therefore, to prove the soundness of the DPOR algorithm it suffices to show that the interleave sets computed in the explored states are persistent sets.

Theorem 1. *Let* s *be a state visited by the algorithm,* $\mathrm{interleave}(\mathrm{s})$ *is a persistent set.*

Proof. If there is no enabled transition in s then $\mathrm{interleave}(\mathrm{s}) = \emptyset$ and hence it is trivially persistent. If there are enabled transitions in s, the algorithm will choose any of these for execution in s, but because once transitions become enabled they cannot become disabled anymore, all the rest of the enabled transitions in s will be eventually executed at some future point after s. On backtracking, the dependent transitions in s will be detected and incrementally added to the interleave set of s. Therefore, at the end of the exploration, all the transitions in $\mathrm{interleave}(\mathrm{s})$ form a persistent set. □

The effectiveness of the reductions achieved by the DPOR algorithm is crucially affected by the precision with which the underlying dependency relation can be computed. The two extremes are to consider all or no transitions as dependent. In the first case, the DPOR algorithm degenerates to full depth-first search. In the second case, it will explore only one successor per state and may miss interleavings that lead to errors. A sound definition of dependence must ensure that two transitions are considered independent only if they commute, and preserve the enabledness of the other transition, at any (global) state where they are both enabled. Because distributed processes do not share global memory, memory updates cannot contribute to dependence, and we need only consider the semantics of the communication actions. However, their semantics must be formally defined to determine (in)dependence.

3.9 Formalization of the Network Model in SimGrid

To implement an efficient DPOR exploration for SimGrid programs, it is important to determine with accuracy the dependency of their communication actions. In SimGrid, the programs can be written using one of its three communication APIs (MSG, SMPI, GRAS), however these lack any formal specification, as they were not designed for formal reasoning. One possibility is to manually specify the APIs guided by the source code and experimentation. However, this is a big, tedious, and daunting task, that in addition would have to be repeated for every API in SimGrid. To illustrate the complexity of this, Palmer et al. [49] have given a formal semantics of a substantial part of MPI for use with DPOR that consists of more than 100 pages of specification.

The key observation of the approach followed in this thesis is that in SimGrid all the user-level communication APIs are based on a minimal internal networking API with four primitives (presented in Section 2.4.1). As previously stated, these functions are the only way that the processes have to modify the shared state. Therefore, the implementation of DPOR in SimGrid operates at the level of these elementary primitives, requiring only to determine the dependency of a reduced set of transitions. This notably simplifies the formalization work as we only have to specify the semantics of four functions, yielding a simpler and elegant solution.

3.10 Overall Network Model

The formal model of the networking primitives is written in the TLA$^+$ [36] specification language. Figure 3.5 presents the first part of the formalization, that consists of the data model of the network. We model the network itself as well as an abstraction of a distributed program that uses it : this allows us to consider the invocation of network operations by the program and prove independence results between these invocations. The model is based on parameters RdV, Addr, Proc, and $\mathrm{Program}$, which represent the set of rendez-vous points, memory addresses, processes, and the program. A multi-process program is represented as an array (indexed by processes) of finite instruction sequences. We distinguish between instructions that invoke network operations (send, receive, wait, and test) and local instructions ; these sets are assumed to be pairwise disjoint. For future use, we also introduce “null” values NoP and NoA for processes and addresses. [2]

2. The axioms of set theory ensure that for any set there exists some value that is not an element of that set.

MODULE *SimixNetwork*

EXTENDS *Naturals, Sequences, FiniteSets*

CONSTANTS RdV, Addr, Proc, Program, ValTrue, ValFalse,
SendIns, RecvIns, WaitIns, TestIns, LocalIns

$\text{Partition(S)} \triangleq \forall x, y \in S : x \cap y = \{\} \vee x = y$

$\text{Instr} \triangleq \text{UNION } \{\text{SendIns, RecvIns, WaitIns, TestIns, LocalIns}\}$

$\text{NoP} \triangleq \text{CHOOSE p} : p \notin \text{Proc}$

$\text{NoA} \triangleq \text{CHOOSE a} : a \notin \text{Addr}$

ASSUME $\text{ValTrue} \in \text{Nat} \wedge \text{ValFalse} \in \text{Nat}$

ASSUME $\text{Partition}(\{\text{SendIns, RecvIns, WaitIns, TestIns, LocalIns}\})$

VARIABLES net, mem, pc

$\text{Comm} \triangleq [\text{id} : \text{Nat}, \text{rdv} : \text{RdV},$
$\text{status} : \{\text{“send”}, \text{“recv”}, \text{“ready”}, \text{“done”}\}$
$\text{src} : \text{Proc}, \text{dst} : \text{Proc},$
$\text{data\_src} : \text{Addr}, \text{data\_dst} : \text{Addr}]$

$\text{mailbox(rdv)} \triangleq \{\text{comm} \in \text{net} :$
$\text{comm.rdv} = \text{rdv} \wedge \text{comm.status} \in \{\text{“send”}, \text{“recv”}\}\}$

$\text{Buffers(pid)} \triangleq$
$\{\text{c.data\_src} : c \in \{d \in \text{net} : d.\text{status} \neq \text{“done”} \wedge (d.\text{src} = \text{pid} \vee d.\text{dst} = \text{pid})\}\}$
$\cup \{\text{c.data\_dst} : c \in \{d \in \text{net} : d.\text{status} \neq \text{“done”} \wedge (d.\text{src} = \text{pid} \vee d.\text{dst} = \text{pid})\}\}$

TypeInvariant $\triangleq$
$\wedge \text{net} \subseteq \text{Comm}$
$\wedge \text{mem} \in [\text{Proc} \to [\text{Addr} \to \text{Nat}]]$
$\wedge \text{pc} \in [\text{Proc} \to \text{Instr}]$

FIGURE 3.5: TLA⁺ model of the communication network : data model.

The system state is represented by three state variables net, mem, and pc. The variable net holds the history of (pending or completed) communication requests (in Comm), which are modeled as records containing a request identifier, the rendez-vous point, the status of the request, the source and destination processes, as well as the memory addresses from which the message content will be taken or where it will be delivered. Variable mem represents the current memory contents per process, and pc (program counter) points to the instruction that will be executed next, for each process.

Next, the module introduces two operators : mailbox(rdv) collects the pending requests for a given rendez-vous point, and Buffers(pid) is the set of memory addresses that appear in communication requests involving process pid, which have not yet completed.

Since TLA$^+$ is untyped, we document the intended types of the state variables by a type invariant, which will have to be shown to be preserved by each possible transition. The network is a set of communications, the system memory is modeled as an nested array associating processes and addresses to natural numbers (representing memory values). Finally, the program counters are modeled as an array yielding for each process some index that points to an instruction.

3.11 Formal Semantics of Communication Primitives

Figure 3.6 shows specifications of the primitive operations for our network model. Process pid can post a Send request for mailbox rdv when its program counter is at a "send" instruction. We distinguish two cases : if a receive request is waiting for communication at rdv, then the send request is matched with the oldest (lowest-numbered) such receive request. The status of the request changes to "ready", indicating that the communication can now actually be performed, and the src and data_src fields are updated according to the parameters of the send request. The communication ID is stored at the memory address indicated by process pid.

If no pending receive request exists for rdv, then a new communication record for rdv is created in the network from the parameters of the send request. The status of this request is set to "send", indicating that it is waiting for a matching receive request, and its ID is stored in the memory of process pid. In either case, the program of process pid advances to some new instruction. (Remember that this is an abstract program model and that any concrete program should be a refinement of what is allowed by this speci-

$$
\begin{array}{l}
\text{Send}(\text{pid}, \text{rdv}, \text{data\_r}, \text{comm\_r}) \triangleq \\
\quad \land\ \text{pc}[\text{pid}] \in \text{SendIns} \\
\quad \land\ \lor\ \exists \text{c} \in \text{mailbox}(\text{rdv}) : \\
\qquad \land\ \text{c.status} = \text{“recv”} \\
\qquad \land\ \forall \text{d} \in \text{mailbox}(\text{rdv}) : \text{d.status} = \text{“recv”} \Rightarrow \text{c.id} \leq \text{d.id} \\
\qquad \land\ \text{net}' = (\text{net} \setminus \{\text{c}\}) \cup \{[\text{c EXCEPT } !.\text{status} = \text{“ready”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad !.\text{src} = \text{pid}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad !.\text{data\_src} = \text{data\_r}]\} \\
\qquad \land\ \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{comm\_r}] = \text{c.id}] \\
\quad\ \lor\ \land\ \neg\exists \text{c} \in \text{mailbox}(\text{rdv}) : \text{c.status} = \text{“recv”} \\
\qquad \land\ \text{LET comm} \triangleq [\text{id} \mapsto \text{Cardinality}(\text{net}) + 1, \\
\qquad\qquad\qquad\qquad\quad \text{rdv} \mapsto \text{rdv},\ \text{status} \mapsto \text{“send”}, \\
\qquad\qquad\qquad\qquad\quad \text{src} \mapsto \text{pid},\ \text{dst} \mapsto \text{NoP}, \\
\qquad\qquad\qquad\qquad\quad \text{data\_src} \mapsto \text{data\_r},\ \text{data\_dst} \mapsto \text{NoA}] \\
\qquad\ \text{IN}\ \ \land\ \text{net}' = \text{net} \cup \{\text{comm}\} \\
\qquad\qquad\ \land\ \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{comm\_r}] = \text{comm.id}] \\
\quad \land\ \exists \text{ins} \in \text{Instr} : \text{pc}' = [\text{pc EXCEPT } ![\text{pid}] = \text{ins}]
\end{array}
$$

$$
\begin{array}{l}
\text{Recv}(\text{pid}, \text{rdv}, \text{data\_r}, \text{comm\_r}) \triangleq \\
\quad \land\ \text{pc}[\text{pid}] \in \text{RecvIns} \\
\quad \land\ \lor\ \exists \text{c} \in \text{mailbox}(\text{rdv}) : \\
\qquad \land\ \text{c.status} = \text{“send”} \\
\qquad \land\ \forall \text{d} \in \text{mailbox}(\text{rdv}) : \text{d.status} = \text{“send”} \Rightarrow \text{c.id} \leq \text{d.id} \\
\qquad \land\ \text{net}' = (\text{net} \setminus \{\text{c}\}) \cup \{[\text{c EXCEPT } !.\text{status} = \text{“ready”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad !.\text{dst} = \text{pid}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad !.\text{data\_dst} = \text{data\_r}]\} \\
\qquad \land\ \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{comm\_r}] = \text{c.id}] \\
\quad\ \lor\ \land\ \neg\exists \text{c} \in \text{mailbox}(\text{rdv}) : \text{c.status} = \text{“send”} \\
\qquad \land\ \text{LET comm} \triangleq [\text{id} \mapsto \text{Cardinality}(\text{net}) + 1, \\
\qquad\qquad\qquad\qquad\quad \text{rdv} \mapsto \text{rdv},\ \text{status} \mapsto \text{“recv”}, \\
\qquad\qquad\qquad\qquad\quad \text{src} \mapsto \text{NoP},\ \text{dst} \mapsto \text{pid}, \\
\qquad\qquad\qquad\qquad\quad \text{data\_src} \mapsto \text{NoA},\ \text{data\_dst} \mapsto \text{data\_r}] \\
\qquad\ \text{IN}\ \ \land\ \text{net}' = \text{net} \cup \{\text{comm}\} \\
\qquad\qquad\ \land\ \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{comm\_r}] = \text{comm.id}] \\
\quad \land\ \exists \text{ins} \in \text{Instr} : \text{pc}' = [\text{pc EXCEPT } ![\text{pid}] = \text{ins}]
\end{array}
$$

$$
\begin{array}{l}
\text{Wait}(\text{pid}, \text{comms}) \triangleq \\
\quad \land \text{pc}[\text{pid}] \in \text{WaitIns} \\
\quad \land \exists \text{comm} \in \text{comms}, \text{c} \in \text{net} : \text{c.id} = \text{mem}[\text{pid}][\text{comm}] \land \\
\qquad \lor \land \text{c.status} = \text{“ready”} \\
\qquad\quad \land \text{mem}' = [\text{mem EXCEPT } ![\text{c.dst}][\text{c.data\_dst}] = \text{mem}[\text{c.src}][\text{c.data\_src}]] \\
\qquad\quad \land \text{net}' = (\text{net} \setminus \{\text{c}\}) \cup \{[\text{c EXCEPT } !.\text{status} = \text{“done”}]\} \\
\qquad \lor \text{c.status} = \text{“done”} \\
\qquad \lor \text{UNCHANGED } \langle \text{mem}, \text{net} \rangle \\
\quad \land \exists \text{ins} \in \text{Instr} : \text{pc}' = [\text{pc EXCEPT } ![\text{pid}] = \text{ins}]
\end{array}
$$

$$
\begin{array}{l}
\text{Test}(\text{pid}, \text{comms}, \text{ret}) \triangleq \\
\quad \land \text{pc}[\text{pid}] \in \text{TestIns} \\
\quad \land \lor \exists \text{comm} \in \text{comms}, \text{c} \in \text{net} : \text{c.id} = \text{mem}[\text{pid}][\text{comm}] \land \\
\qquad \lor \land \text{c.status} = \text{“ready”} \\
\qquad\quad \land \text{mem}' = [\text{mem EXCEPT } ![\text{c.dst}][\text{c.data\_dst}] = \text{mem}[\text{c.src}][\text{c.data\_src}], \\
\qquad\qquad\qquad\qquad\qquad\qquad ![\text{pid}][\text{ret}] = \text{ValTrue}] \\
\qquad\quad \land \text{net}' = (\text{net} \setminus \{\text{c}\}) \cup \{[\text{c EXCEPT } !.\text{status} = \text{“done”}]\} \\
\qquad \lor \land \text{c.status} = \text{“done”} \\
\qquad\quad \land \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{ret}] = \text{ValTrue}] \\
\qquad\quad \land \text{net}' = \text{net} \\
\qquad \lor \land \neg \exists \text{comm} \in \text{comms}, \text{c} \in \text{network} : \land \text{c.id} = \text{memory}[\text{pid}][\text{comm}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land \text{c.status} \in \{\text{“ready”}, \text{“done”}\} \\
\qquad\quad \land \text{mem}' = [\text{mem EXCEPT } ![\text{pid}][\text{ret}] = \text{ValFalse}] \\
\qquad\quad \land \text{net}' = \text{net} \\
\quad \land \exists \text{ins} \in \text{Instr} : \text{pc}' = [\text{pc EXCEPT } ![\text{pid}] = \text{ins}]
\end{array}
$$

$$
\begin{array}{l}
\text{Local}(\text{pid}) \triangleq \\
\quad \land \text{pc}[\text{pid}] \in \text{LocalIns} \\
\quad \land \text{net}' = \text{net} \\
\quad \land \text{mem}' \in [\text{Proc} \to [\text{Addr} \to \text{Nat}]] \\
\quad \land \forall \text{p} \in \text{Proc}, \text{a} \in \text{Addr} : \text{mem}'[\text{p}][\text{a}] \neq \text{mem}[\text{p}][\text{a}] \Rightarrow \text{p} = \text{pid} \land \text{a} \notin \text{Buffers}(\text{pid}) \\
\quad \land \exists \text{ins} \in \text{Instr} : \text{pc}' = [\text{pc EXCEPT } ![\text{pid}] = \text{ins}]
\end{array}
$$

$$
\begin{array}{l}
\text{Next} \triangleq \exists \text{p} \in \text{Proc}, \text{data\_r} \in \text{Addr}, \text{comm\_r} \in \text{Addr}, \text{rdv} \in \text{RdV}, \text{ret\_r} \in \text{Addr} : \\
\quad \lor \text{Send}(\text{p}, \text{rdv}, \text{data\_r}, \text{comm\_r}) \\
\quad \lor \text{Recv}(\text{p}, \text{rdv}, \text{data\_r}, \text{comm\_r}) \\
\quad \lor \text{Wait}(\text{p}, \text{comm\_r}) \\
\quad \lor \text{Test}(\text{p}, \text{comm\_r}, \text{ret\_r}) \\
\quad \lor \text{Local}(\text{p})
\end{array}
$$

FIGURE 3.6: TLA$^+$ model of the communication network : operations (excerpt).

fication.)

The specification of the receive operation (Recv) is symmetrical, it first checks for pending send requests, otherwise it creates a new communication record using the parameters of the request.

A Wait operation allows a process to wait for the completion of any of a set comms of network operations (represented by the memory addresses in which their communication IDs are stored). It is enabled as soon as the status of one of these operations is either "ready" or "done". If the status is "ready", the communication is performed by transferring the contents of the memory buffer of the source process to the memory buffer of the destination process, and the status is updated to "done". If the status was already "done" before the Wait operation, the operation has no effect on the network or the memory.

The operation Test can be used by a process to check if given a set of communications comms, at least one has completed or not. If so, the primitive acts like a Wait, but also returns the result ValTrue in the memory address indicated by parameter ret. Otherwise, it returns ValFalse, but does not block the calling process as a Wait instruction would.

Finally, we also model local operations of processes by a loosely specified action Local, which does not modify the network, but may update the memory of the process that executes the operation, except for any locations that are the source or destination addresses of pending network operations in which the process participates.

The overall next-state action of the specification is simply defined as the disjunction of these elementary actions, for parameter values ranging over the appropriate sets.

3.12 Independence Theorems

Determining (in)dependence between transitions is fundamental for an efficient and correct DPOR-based exploration algorithm. We now reformulate the notion of independence in TLA⁺, and we prove the independence between certain fundamental primitives using the specification introduced in Section 3.11.

As presented in Section 3.8, two actions A and B are independent if at any state where both are enabled, neither disables the other one, and executing the actions in either order leads to the same state. This can be expressed in TLA⁺ as the following predicate over

actions :

$$\begin{aligned} \mathrm{I}(\mathrm{A},\mathrm{B}) \triangleq \textsc{Enabled}\ \mathrm{A} \wedge \textsc{Enabled}\ \mathrm{B} \Rightarrow &\wedge \mathrm{A} \Rightarrow (\textsc{Enabled}\ \mathrm{B})' \\ &\wedge \mathrm{B} \Rightarrow (\textsc{Enabled}\ \mathrm{A})' \\ &\wedge \mathrm{A} \cdot \mathrm{B} \equiv \mathrm{B} \cdot \mathrm{A} \end{aligned}$$

Observe that by definition, independence is symmetric : $\mathrm{I}(\mathrm{A},\mathrm{B}) \equiv \mathrm{I}(\mathrm{B},\mathrm{A})$. The actions A and B are dependent if $\neg\mathrm{I}(\mathrm{A},\mathrm{B})$. Typically, two actions are dependent if they may modify shared parts of the state space, such as shared objects or memory buffers.

Before introducing the theorems of independence, we state a useful lemma that is used throughout the proofs.

Lemma 1. *If two different type of actions* A,B *(*$\mathrm{Send},\mathrm{Recv},\mathrm{Wait},\mathrm{Test},\mathrm{orLocal}$*) are enabled in the same state, then they belong to different processes.*

Proof. All the actions in the specification are parametrized by the id of the process executing the action. If A and B are both enabled then their preconditions hold. Because the precondition of every action includes a predicate stating that the program counter should be in an instruction representing the corresponding type of the action, and the set of instructions corresponding to every action is disjoint with those of other actions, then the processes executing the actions A and B must be different. □

Based on the TLA⁺ specifications of the network primitives, we now state several theorems regarding the independence of transitions.

Theorem 2. *Any two* Send *and* Recv *transitions are independent.*

$$\begin{aligned} &\forall \mathrm{p}_1,\mathrm{p}_2 \in \mathrm{Proc}, \mathrm{rdv}_1,\mathrm{rdv}_2 \in \mathrm{RdV}, \mathrm{d}_1,\mathrm{d}_2 \in \mathrm{Addr}, \mathrm{c}_1,\mathrm{c}_2 \in \mathrm{Addr} : \\ &\quad \mathrm{I}(\mathrm{Send}(\mathrm{p}_1,\mathrm{rdv}_1,\mathrm{d}_1,\mathrm{c}_1), \mathrm{Recv}(\mathrm{p}_2,\mathrm{rdv}_2,\mathrm{d}_2,\mathrm{c}_2)) \end{aligned}$$

This result might be surprising at first glance.

Proof. Assume that the Send and Recv action are both enabled. Then the processes p_1 and p_2 must be different by Lemma 1. Recall that data is only transmitted during Wait or Test steps. The only shared state that is modified by the postconditions of both actions is the network (net'), therefore no transition can disable the other from happening. If the Send and Recv transitions concern different rendez-vous points, then there is no shared state modified and thus they are trivially independent. On the contrary, if the requests are posted for the same rendez-vous point, two situations can happen : the mailbox is

empty and the two requests match each other independently in which order they are performed, or there are some pending requests (which must be of the same type, either all Send or all Recv requests), and the matching transition always pairs with the one with the lowest id, the head of the queue. The other transition, being of the same type as the requests pending on the rendez-vous point, is added at the tail of the queue. This also happens independently of the order in which the transitions are executed, and the resulting state is the same, which proves the theorem. □

Theorem 3. *Two* Send *or two* Recv *operations performed by different processes and posted to different rendez-vous points are independent.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{rdv}_1, \mathrm{rdv}_2 \in \mathrm{RdV}, \mathrm{d}_1, \mathrm{d}_2 \in \mathrm{Addr}, \mathrm{c}_1, \mathrm{c}_2 \in \mathrm{Addr} :& \\ \mathrm{p}_1 \neq \mathrm{p}_2 \wedge \mathrm{rdv}_1 \neq \mathrm{rdv}_2 \Rightarrow \wedge\, \mathrm{I}(\mathrm{Send}(\mathrm{p}_1, \mathrm{rdv}_1, \mathrm{d}_1, \mathrm{c}_1), \mathrm{Send}(\mathrm{p}_2, \mathrm{rdv}_2, \mathrm{d}_2, \mathrm{c}_2))& \\ \wedge\, \mathrm{I}(\mathrm{Recv}(\mathrm{p}_1, \mathrm{rdv}_1, \mathrm{d}_1, \mathrm{c}_1), \mathrm{Recv}(\mathrm{p}_2, \mathrm{rdv}_2, \mathrm{d}_2, \mathrm{c}_2))&\end{aligned}$$

Proof. From the definition of the Send action, if the processes and the rendez-vous points are different (hypothesis $\mathrm{p}_1 \neq \mathrm{p}_2 \wedge \mathrm{rdv}_1 \neq \mathrm{rdv}_2$), the two actions modify disjoint parts of the state space. In the case they find a matching receive request, these must be different because they belong to different rendez-vous points. On the contrary, if they create new requests they are queued in different rendez-vous points. Moreover, execution of one action will not disable the other one, and therefore the actions are independent.

The proof of independence for two Recv action is analogous. □

Theorem 4. Wait *or* Test *operations for the same communication request are independent.*

$$\begin{aligned}&\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c} \in \mathrm{Addr} : \mathrm{I}(\mathrm{Wait}(\mathrm{p}_1, \{\mathrm{c}\}), \mathrm{Wait}(\mathrm{p}_2, \{\mathrm{c}\})) \\ &\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c}, \mathrm{r}_1, \mathrm{r}_2 \in \mathrm{Addr} : \mathrm{I}(\mathrm{Test}(\mathrm{p}_1, \mathrm{c}, \mathrm{r}_1), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}_2)) \\ &\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{c}, \mathrm{r}_2 \in \mathrm{Addr} : \mathrm{I}(\mathrm{Wait}(\mathrm{p}_1, \{\mathrm{c}\}), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}_2))\end{aligned}$$

Proof. Assume that both Wait operations are enabled. Execution of the first one changes the status of the communications to "done" and copies the data from the sender to the receiver (if the status wasn't already "done", in which case even the first Wait operation is a no-op). The second Wait transition then finds the communication with "done" status and leaves the shared state unchanged. Because the memory addresses of the buffers of the sender and receiver are stored in the communication, the order of execution doesn't affect the final state.

The proof of independence of two Test actions, or for a Wait and a Test action, for the same communication request is similar. □

Note that in Theorem 4 we consider only singleton sets of communications because in the DPOR algorithm the dependency is evaluated during the backtracking once the action was executed, at which point only one communication from each set was already chosen.

Theorem 5. *Any two local actions of different processes are mutually independent.*

$$\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc} : \mathrm{p}_1 \neq \mathrm{p}_2 \Rightarrow \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Local}(\mathrm{p}_2))$$

Proof. Local actions of different processes modify disjoint parts of the system state space, hence they obviously commute.

□

Theorem 6. *Any two* Local *and* Send *or* Recv *transitions are independent.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{rdv} \in \mathrm{RdV}, \mathrm{d} \in \mathrm{Addr}, \mathrm{c} \in \mathrm{Addr} :\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Send}(\mathrm{p}_2, \mathrm{rdv}, \mathrm{d}, \mathrm{c}))\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Recv}(\mathrm{p}_2, \mathrm{rdv}, \mathrm{d}, \mathrm{c}))\end{aligned}$$

Proof. Consider a Local and a Send action. Again, p_1 and p_2 must be different processes due to Lemma 1. Therefore, they modify disjoint parts of array mem (in particular, the assumption on the modifications allowed by Local actions implies that only the memory of the process p_1 may be affected), and the only modifications to net are due to the Send action, and they are independent of any modifications that the Local action may perform.

The proof of independence between Local and Recv actions is analogous. □

Theorem 7. *Any two* Local *and* Wait *or* Test *transitions are independent.*

$$\begin{aligned}\forall \mathrm{p}_1, \mathrm{p}_2 \in \mathrm{Proc}, \mathrm{comm} \in \textsc{subset}\ \mathrm{Addr}, \mathrm{c}, \mathrm{r} \in \mathrm{Addr} :\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Wait}(\mathrm{p}_2, \mathrm{comm}))\\ \wedge\, \mathrm{I}(\mathrm{Local}(\mathrm{p}_1), \mathrm{Test}(\mathrm{p}_2, \mathrm{c}, \mathrm{r}))\end{aligned}$$

Proof. Consider a Local and a Wait action. For the same reasons as before, the processes performing these actions must be distinct and the only possible modification to the network stems from the Wait action and is independent of whatever modification the Local action performs. In case the Wait action modifies a memory location, it concerns the destination process of the communication request that is completed, and a location that is included in the CommBuffers of that process. Therefore, the Local action is not allowed

to modify the same location, and the effects on the memory of the two actions must commute. Moreover, the Local action cannot disable the Wait action ; in particular, it cannot modify the memory addresses in comm since the two processes are distinct.

The proof of independence between Local and Test actions is analogous.

□

Using the previous results we can now define an independence predicate

$$\begin{aligned}
\mathrm{Indep}(\mathrm{t_i}, \mathrm{t_j}) \triangleq\ & \exists \mathrm{p_1}, \mathrm{p_2} \in \mathrm{Proc}, \mathrm{rdv_1}, \mathrm{rdv_2} \in \mathrm{RdV}, \mathrm{d_1}, \mathrm{d_2} \in \mathrm{Addr}, \mathrm{c_1}, \mathrm{c_2} \in \mathrm{Addr} : \\
& \lor\ \mathrm{t_i} = \mathrm{Send}(\mathrm{p_1}, \mathrm{rdv_1}, \mathrm{d_1}, \mathrm{c_1}) \land \mathrm{t_j} = \mathrm{Recv}(\mathrm{p_2}, \mathrm{rdv_2}, \mathrm{d_2}, \mathrm{c_2}) \\
& \lor \land\ \mathrm{t_i} = \mathrm{Send}(\mathrm{p_1}, \mathrm{rdv_1}, \mathrm{d_1}, \mathrm{c_1}) \land \mathrm{t_j} = \mathrm{Send}(\mathrm{p_2}, \mathrm{rdv_2}, \mathrm{d_2}, \mathrm{c_2}) \\
& \quad \land\ \mathrm{p_1} \neq \mathrm{p_2} \land \mathrm{rdv_1} \neq \mathrm{rdv_2} \\
& \lor \land\ \mathrm{t_i} = \mathrm{Recv}(\mathrm{p_1}, \mathrm{rdv_1}, \mathrm{d_1}, \mathrm{c_1}) \land \mathrm{t_j} = \mathrm{Recv}(\mathrm{p_2}, \mathrm{rdv_2}, \mathrm{d_2}, \mathrm{c_2}) \\
& \quad \land\ \mathrm{p_1} \neq \mathrm{p_2} \land \mathrm{rdv_1} \neq \mathrm{rdv_2} \\
& \lor \land\ \mathrm{t_i} = \mathrm{Wait}(\mathrm{p_1}, \{\mathrm{c_1}\}) \land \mathrm{t_j} = \mathrm{Wait}(\mathrm{p_2}, \{\mathrm{c_2}\}) \\
& \quad \land\ \mathrm{c_1} = \mathrm{c_2} \\
& \lor \land\ \mathrm{t_i} = \mathrm{Test}(\mathrm{p_1}, \{\mathrm{c_1}\}, \mathrm{d_1}) \land \mathrm{t_j} = \mathrm{Test}(\mathrm{p_2}, \{\mathrm{c_2}\}, \mathrm{d_2}) \\
& \quad \land\ \mathrm{c_1} = \mathrm{c_2} \\
& \lor \land\ \mathrm{t_i} = \mathrm{Wait}(\mathrm{p_1}, \{\mathrm{c_1}\}) \land \mathrm{t_j} = \mathrm{Test}(\mathrm{p_2}, \{\mathrm{c_2}\}, \mathrm{d_2}) \\
& \quad \land\ \mathrm{c_1} = \mathrm{c_2} \\
& \lor\ \mathrm{t_i} = \mathrm{Local}(\mathrm{p_1}) \land \mathrm{t_j} = \mathrm{Local}(\mathrm{p_2}) \\
& \lor\ \mathrm{t_i} = \mathrm{Local}(\mathrm{p_1}) \land \mathrm{t_j} = \mathrm{Send}(\mathrm{p_2}, \mathrm{rdv_2}, \mathrm{d_2}, \mathrm{c_2}) \\
& \lor\ \mathrm{t_i} = \mathrm{Local}(\mathrm{p_1}) \land \mathrm{t_j} = \mathrm{Recv}(\mathrm{p_2}, \mathrm{rdv_2}, \mathrm{d_2}, \mathrm{c_2}) \\
& \lor\ \mathrm{t_i} = \mathrm{Local}(\mathrm{p_1}) \land \mathrm{t_j} = \mathrm{Wait}(\mathrm{p_2}, \{\mathrm{c_2}\}) \\
& \lor\ \mathrm{t_i} = \mathrm{Local}(\mathrm{p_1}) \land \mathrm{t_j} = \mathrm{Test}(\mathrm{p_2}, \{\mathrm{c_2}\}, \mathrm{d_2})
\end{aligned}$$

And its symmetric closure

$$\mathrm{Independent}(\mathrm{t_i}, \mathrm{t_j}) \triangleq \mathrm{Indep}(\mathrm{t_i}, \mathrm{t_j}) \lor \mathrm{Indep}(\mathrm{t_j}, \mathrm{t_i})$$

Finally, the dependency relation is defined as

$$\mathrm{Depend}(\mathrm{t_i}, \mathrm{t_j}) \triangleq \neg \mathrm{Independent}(\mathrm{t_i}, \mathrm{t_j})$$

3.13 Implementing Higher-Level Communications APIs

The implementation of the DPOR exploration algorithm at a lower level of abstraction simplifies the required formal framework, and allows to verify programs written for

all the communication APIs of SimGrid using the same exploration algorithm. However, it is important to note that the implementation of the higher level APIs must be carefully thought out, otherwise it can render useless the reductions obtained by DPOR (see Section 2.4.1 for more details).

A naïve implementation can introduce internal non-determinism that is not observable from the user application. Not addressing these issues may introduce additional non-determinism, generating spurious interleavings during model checking that can destroy the benefits of the DPOR exploration.

To illustrate this problem, consider the implementation of a *WaitAll* function that waits for all of the communications in a list to finish. Listing 3.1 shows a possible inefficient implementation, that expects a set of communications identifiers and repeatedly uses *WaitAny* for all unfinished communications, until no one is left. While correct, such an implementation would introduce a non-deterministic choice among the finished communications. Then the model checker would have to consider all their possible permutation orderings, which is irrelevant to the semantics of *WaitAll*.

An alternative to avoid this, is to issue all the *WaitAny* operations in sequence for all the communication operations as shown in listing 3.2. This implementation is semantically equivalent to the previous one but it avoids forcing the exploration algorithm to consider all the permutations, because at each iteration it waits for only one communication.

Listing 3.1: Inefficient WaitAll.

```
1 void WaitAll(comm_list[])
2 {
3   while(len(comm_list) > 0){
4     comm = WaitAny(comm_list);
5     list_remove(comm, comm_list);
6   }
7 }
```

Listing 3.2: Efficient WaitAll.

```
1 void WaitAll(comm_list[])
2 {
3   for(i=0;i<len(comm_list);i++){
4     WaitAny(comm_list[i]);
5   }
6 }
```

3.14 Comparison to ISP and DAMPI

In this section we further compare SimGridMC with two existing tools, namely ISP [69] and DAMPI [68], both designed to verify programs written using the MPI library. The first conceptual difference between these and SimGridMC is that the latter is designed

to study programs specifically written for SimGrid, which supports MPI but also other communication APIs, whereas ISP and DAMPI focus exclusively on MPI programs. This difference has an important impact on the architecture, SimGrid replaces the MPI runtime with its own implementation of the API, that includes the exploration algorithm. On the contrary, ISP and DAMPI intercept the API calls before they enter the MPI runtime using a precompilation step, then the exploration of the interleavings is performed by issuing the calls in all relevant orderings.

Performing the exploration at different points of the execution stack yields different approaches to apply the dynamic partial order reduction technique. As SimGrid internally represents all the communication APIs using a minimal set of networking primitives, similarly to how the MPI run-time is implemented on top of the basic OS communication API, it suffices then to reason about the dependency of these primitives, which have a much simpler semantics. On the other hand, ISP and DAMPI see the MPI runtime as a black box, and require a full formal specification of the API in order to perform a sound reduction. The generality and simplicity of SimGrid does not come for free. ISP and DAMPI, have exploration algorithms specifically tailored for MPI based on a reduced execution semantic (POE) that yield better reductions than those obtained in SimGrid. For example, consider a barrier operation on which all the processes participate. According to the MPI semantics in [49], the barrier is independent with all MPI communication functions, hence it is not considered by ISP and DAMPI for different interleavings. However, in SimGridMC the MPI barrier is implemented on top of lower level primitives, which might introduce internal non-determinism, forcing the exploration algorithm to consider more than one equivalent interleaving for the same barrier operation.

Moreover, DAMPI is a complete decentralized solution, that is designed to verify HPC applications using the same platforms that run them. It relies on a custom vector clock algorithm to detect and steer the executions of the program. It also uses a bound in the exploraion of the state space called *bounded mixing*. The intuition behind the bounding technique relies in the empirical observation that MPI programs go through *zones* of computations, and that the interaction among these zones is relatively low. Therefore, DAMPI explores all behaviors in K-sized windows (the zones), but it does not choose all trajectories that reach each K-sized window. The authors claim that it can provide coverage that is complementary to DFS, as it visits all depths (contrary to depth bounding) and does an interesting search at each depth.

Listing 3.3: Example 1.

```
1 if (rank == 0){
2   for (i=0; i < N-1; i++){
3     MPI_Recv(&val, MPI_ANY_SOURCE);
4   }
5   MC_assert(val == N);
6 } else {
7   MPI_Send(&rank, 0);
8 }
```

Listing 3.4: Example 2.

```
1 if (rank % 3 == 0) {
2   MPI_Recv(&val, MPI_ANY_SOURCE);
3   MPI_Recv(&val, MPI_ANY_SOURCE);
4 } else {
5   MPI_Send(&rank, (rank / 3) * 3);
6 }
```

3.15 Experimental Results

In this section we present two verification experiments using the SMPI and the MSG communication APIs supported by SimGrid. We thus illustrate the ability of our approach to use a generic DPOR exploration algorithm for different communication APIs through an intermediate communication layer. Each experiment aims to evaluate the effectiveness of the DPOR exploration at this lower level of abstraction compared to a simple DFS exploration. We use a depth bound fixed at 1000 transitions (which was never reached in these experiments), and run SimGrid SVN revision 9888 on a CPU Intel Core2 Duo T7200 2.0GHz with 1GB of RAM under Linux.

3.15.1 SMPI Experiments

The first case study is based upon two small C programs using MPI that are designed to measure the performance of our DPOR algorithm when dealing with synchronous communications. These programs are parametrized by the number of processes N.

The first example, presented in Listing 3.3, shows an MPI program with N+1 processes. The process with rank 0 waits for a message from each of the other processes, while the other processes send their rank value to process 0. The property to verify is coded as the assertion at line 5 that checks for the incorrect assumption of a fixed message receive order, where the last received message will be always from the process with rank N.

Table 3.1a shows the timing and the number of states visited before finding a violation of the assertion for 3, 4, and 5 processes. In this case, the number of processes does not have a significant impact on the number of visited states because the error state appears early in the visiting order of the DFS. Still, using DPOR helps to reduce the number of visited states by more than 50% when compared to standard DFS.

Table 3.1b shows the effectiveness of DPOR for a complete state space exploration of

the same program (without the assertion, and no bound reached). Here, the use of DPOR notably reduces the number of visited states by an order of magnitude.

TABLE 3.1: Timing, number of expanded states, and peak memory usage for the case studies in Listing 3.3 and Listing 3.4.

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	119	0.097 s	23952 kB	43	0.063 s	23952 kB
4	123	0.114 s	25008 kB	47	0.064 s	25024 kB
5	127	0.112 s	26096 kB	51	0.072 s	26080 kB

(a) Results for Listing 3.3 with assertion checking

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
2	13	0.054 s	21904 kB	5	0.046 s	18784 kB
3	520	0.216 s	23472 kB	72	0.069 s	23472 kB
4	60893	19.076 s	24000 kB	3382	0.913 s	24016 kB
5	-	-	-	297171	84.271 s	25584 kB

(b) Results for Listing 3.3 with full state space coverage

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	520	0.247 s	23472 kB	72	0.074 s	23472 kB
6	$>$10560579	$>$1 h	-	1563	0.595 s	26128 kB
9	-	-	-	32874	14.118 s	29824 kB

(c) Results for Listing 3.4 with full state space coverage

The second example, presented in Listing 3.4, shows the relevance of performing the partial order reduction dynamically. This time the number of processes in the system should be a multiple of 3. Every process with a rank that is a multiple of three will wait for a message from the next two processes, thus process 0 will receive from processes 1 and 2, process 3 from processes 4 and 5, etc. It is quite obvious that each group of three processes is independent from the others, but standard static reduction techniques would not be able to determine this because the value of rank is defined at runtime. Again, no property is verified, as we try to compare the reductions obtained by DPOR

when covering the complete state space.

Table 3.1c shows the experimental results for a complete exploration of the state space for 3, 6, and 9 processes. In this case the DFS with 6 processes was interrupted after one hour, and up to that point it had visited 320 times more states than the complete state space exploration of the same program but for 9 processes with DPOR enabled.

3.15.2 MSG Experiment : CHORD

As our second case study we consider a more realistic SimGrid program. It is an implementation of the Chord [59] protocol using the MSG communication API of SimGrid. Chord is a well known peer-to-peer lookup service, designed to be scalable, and to function even with nodes leaving and joining the system. This implementation was originally developed to study the performance of the SimGrid simulator.

The algorithm works by arranging the nodes in a logical ring. It works in phases, that stabilize the lookup information on every node that describe the structure of the ring. During each phase, nodes exchange messages to update their knowledge about who left and joined the ring, and eventually converge to a consistent global vision.

Listing 3.5 shows a simplified version of Chord's main loop. In MSG, processes exchange *tasks* containing the messages defined by the user. Each node starts an asynchronous task receive communication (line 3), waiting for requests from the other nodes to be served. If there is one (the condition at line 4 is true), a handler is called to reply with the appropriate answer using the same received task (line 5). Otherwise, if the delay for the next lookup table update has passed, it performs the update in four steps : request the information (lines 7-9), wait for the answer (lines 12-14), update the lookup tables (line 19), and notify changes to other nodes (line 22).

Running Chord in the simulator, we occasionally spotted an incorrect task reception in line 14 that led to an invalid memory read, producing a segmentation fault. Due to the deterministic scheduling produced by the simulator, the problem only appeared when running simulations with more than 90 nodes. Although we thus knew that the code contained a problem, we were unable to identify the cause of the error because of the size of the instances where it appeared and the amount of debugging information that these generated.

We decided to use SimGridMC to further investigate the issue, exploring a scenario with just two nodes and checking the property $task == update\_task$ at line 16 of listing 3.5. In a matter of seconds we were able to trigger the bug and could understand the

Listing 3.5: Main loop of CHORD (simplified).

```
1  while(1) {
2    if (!rcv_comm)
3      rcv_comm = MSG_task_irecv(&task);
4    if (MSG_comm_test(rcv_comm)) {
5      handle(task);
6    } else if(time > next_update_time) {
7      /* Send update request task */
8      snd_comm = MSG_task_isend(&update_task);
9      MSG_task_wait(snd_comm);
10
11     /* Receive the answer */
12     if(rcv_comm == NULL) /* <- BUG! */
13       rcv_comm = MSG_task_irecv(&task);
14     MSG_task_wait(rcv_comm);
15
16     MC_assert(task == update_task); /* <-- Assertion verified by the MC */
17
18     /* Update tables with received task */
19     update_tables(task);
20
21     /* Notify some nodes of changes */
22     notify();
23   } else {
24     sleep(5);
25   }
26 }
```

source of the problem by examining the counter-example trace, which appears in listing 3.6. It should be read top-down and the events of each node are tabulated for clarity. The Notify task sent by node 1 in line 22 of listing 3.5 is incorrectly taken by node 2 at line 14 as the answer to the update request sent by it in line 8. This is due to an implementation error in the line 12 : the code reuses the variable *rcv_comm*, incorrectly assuming this to be safe because of the guard of that branch, but in fact the condition may change after the guard is evaluated.

The Chord implementation that was verified has 563 lines of code, and the model checker found the bug after visiting just 478 states (in 0.280 s) using DPOR ; without DPOR it had to compute 15600 states (requiring 24 s) before finding the error trace. Both runs had an approximate peak memory usage of 72 MB, measured with the `/usr/bin/time` program provided by the operating system.

Listing 3.6: Counter-example.

```
#line          Node 1                  #line          Node 2

 3: rcv_comm = MSG_task_irecv(&task)
 4: MSG_comm_test(rcv_comm) == FALSE
 8: snd_comm =
      MSG_MSG_task_isend(&update_task)
                                        3: rcv_comm = MSG_task_irecv(&task)
 9: MSG_task_wait(snd_comm)
                                        4: MSG_comm_test(rcv_comm) == TRUE
                                        5: handle(task)
                                        3: rcv_comm = MSG_task_irecv(&task)
                                        4: MSG_comm_test(rcv_comm) == FALSE
14: MSG_task_wait(recv_comm)
22: Notify()
 3: rcv_comm = MSG_task_irecv(&task)
                                        8: snd_comm =
                                             MSG_MSG_task_isend(&update_task)
                                        9: MSG_task_wait(snd_comm)
                                       14: MSG_task_wait(recv_comm)
```

3.16 Summary

In this chapter we have presented SimGridMC, a model checker for distributed C programs that may use one of three different communication APIs. Like similar tools, SimGridMC is based on the idea of stateless model checking, which avoids computing and storing the process state at interruptions, and relies on dynamic partial order reduction in order to make verification scale to realistic programs. One originality of SimGridMC is that it is firmly integrated with the pre-existing simulation framework provided by SimGrid, allowing programmers to use the same code and the same platform for verification and for performance evaluation. Another specificity is the support for multiple communication APIs. We have implemented sensibly different APIs in terms of a small set of elementary primitives, for which we could provide a formal specification together with independence theorems with reasonable effort, rather than formalize three complete communication APIs. We have been pleasantly surprised by the fact that this approach has not compromised the degree of reductions that we obtain, and allowed us to handle complex implementations like the Chord P2P protocol among others, that would be otherwise intractable.

Thanks to the work presented in Chapter 2, the integration of the model checker in

the existing SimGrid platform has been conceptually simple. The simulation and model checking share core functionality such as the virtualization of the execution environment and the ability to execute and interrupt user processes. However, model checking tries to explore all possible schedules, whereas simulation first generates a schedule that it then enforces for all processes. SimGrid benefited from the development of SimGridMC in that it led to a better modularization and reorganization of the existing code. Moreover, the deep understanding of the execution semantics gained during this work helped us envision the parallel simulation kernel presented in Chapter 4.

Chapitre 4

Parallelizing the Simulation Loop

A challenging area of research in simulation is the problem of scalability. The size and complexity of distributed systems increases every day, together with the need for tools to help the developers understand and optimize these systems. Simulators are no exception to this problem, they must be capable of scaling to handle instances of the size required for these increasing demands. A simulation can be either memory bounded, CPU bounded, or both, meaning that the experiments may be limited by the amount of memory available or the time required to compute the simulation evolution. The amount of memory required to run a simulation is related with variables like the number of simulated processes (and their memory usage), together with the size of the simulated platform, that is tied to the number of hosts, links, the topology and the routing information. CPU limitations on the other hand are related with the amount of computation performed by the simulated processes, and the amount of interactions with the platform being simulated. Examples of these are typical HPC or grid applications which have processes that perform large computations, or Peer-to-Peer programs composed of millions of processes that do not compute much locally, but generate a huge amount of interactions with the platform that results in a high workload for the simulator.

The problem of memory bounded simulations can be simply overcomed by increasing the amount of available memory on the computer running the simulation. However, for CPU bounded simulations there is no straightforward solution. Simulations are inherently sequential, and due to the current hardware limitations, the CPUs improve mostly in the parallel computing power. The classical approaches to parallelize simulations (see Section 1.2.2) are not very well suited to the general problem of simulating distributed

programs. Conservative space decomposition techniques are heavily dependent on the characteristics of the system being simulated, such as the *lookahead*. Using distributed systems terminology, the lookahead represents the latency between the processes. A platform with low latency would hardly benefit from a conservative approach, where the processes are allowed to advance in the simulated time at most to the minimum of the latencies that interconnect them. According to [42, 51], DES is classified as an ordered data-driven algorithm, where the parallelization structure depends on the events of the system. The authors conclude that for these systems optimistic parallelization is the only general-purpose approach. In optimistic simulations processes can freely advance in the simulation time, but they require rewinding the processes to consistent global states when an out of orderevent is detected. This is an expensive operation (in particular when simulating *programs*), and in systems with a high amount of exchanged messages it occurs rather frequently. Finally, time-parallel decomposition, is impractical for simulations whose state consists of the running memory of the program. The complexity of accurately predicting the running state of a program at future points in time makes time decomposition techniques impractical.

In this chapter we explore a novel approach that is conceptually simpler than the classical parallelization schemes. The key observation is that user processes only modify local data, hence they are intrinsically parallel. The idea consists of keeping the simulation sequential, but parallelizing the execution of the user processes at every simulation round. To better understand this approach, we present an analysis of the complexity of the parallelization, and we give a criterion to estimate in what scenarios a speedup can be expected. The experimental results show that it leads to noticeable speedups in certain scenarios, but also leads to performance hits in others. An interesting contribution of this work is the observation of the relation between the precisions of the models and the potential parallelization of the simulations.

4.1 Parallel Execution of User Code

The parallelization scheme presented in this section tries to avoid the complexity of maintaining the coherence of multiple simulation timelines. Instead, it keeps a unique timeline, but exploits the potential parallelism that exists inside it.

Algorithm 5 shows again the pseudocode of SimGrid's main loop as it was in version 3.4, and Figure 4.1 illustrates the execution of one iteration with processes $\mathrm{P}_{\mathrm{t_n}} =$

Algorithm 5 Main Loop.

```
1: time ← 0
2: P_time ← P
3: while P_time ≠ ∅ do
4:    schedule(P_time)
5:    handle_requests()
6:    time ← surf_solve(&done_actions)
7:    P_time ← process_unblock(done_actions)
8: end while
```

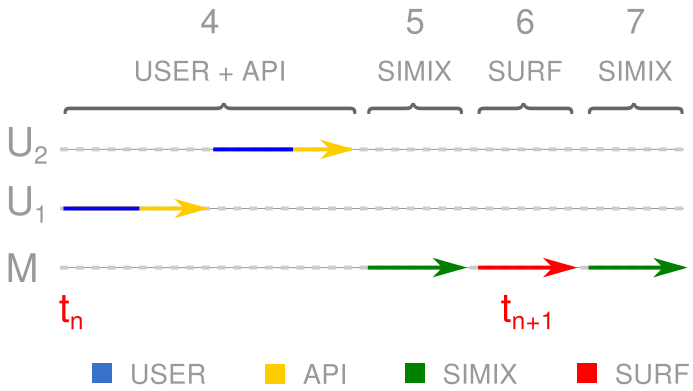


FIGURE 4.1: Steps of the main loop iteration.

$\{U_1, U_2\}$. Each step of the loop corresponds to one step of the simulation iteration, here numerated using the line number, from 4 to 5. The key observation here is that the simulated time advances only between calls to the simulation core in line 6. This means that all the actions performed by U_1 and U_2, when scheduled at line 2 happen at the *same* simulated time, hence there is no observable difference between the two possible execution orderings. Formally, the states of the processes in P_{time} are concurrent according to the $\rightarrow$ relation being generated,

$$\forall p_i, p_j \in P_{time},\ s.p_i \parallel s.p_j$$

Therefore, if all the processes in P_{time} in are executed in parallel, there is no risk of out-of-order executions.

Algorithm 6 shows the pseudo-code of the main loop, but now with the parallel execution of the user processes. The sequential scheduling function is now replaced in line 4 by a parallel one. The rest of the simulation loop remains unchanged. Note that this is only possible after the rearchitecture work presented in Chapter 2, that factored out all the modifications to the shared state of the simulation from the user context, into

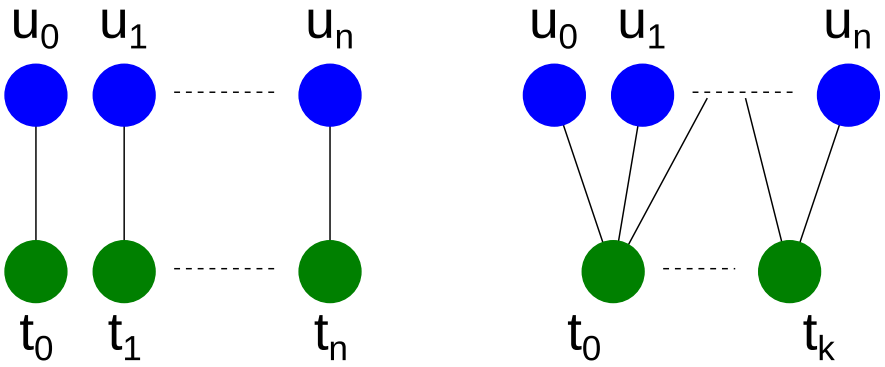
SIMIX v2.0 that executes in the maestro context at line 5.

Algorithm 6 Parallel Main Loop.

```
1: time ← 0
2: P_time ← P
3: while P_time ≠ ∅ do
4:    parallel_schedule(P_time)
5:    handle_requests()
6:    time ← surf_solve(&done_actions)
7:    P_time ← process_unblock(done_actions)
8: end while
```

4.2 Architecture of the Parallel Execution

In the previous section we have presented the parallelization scheme proposed in this work. In this section we discuss two possible approaches to implement the scheduling of the processes.



(a) One thread per process. (b) Pool of worker threads.

FIGURE 4.2: Two possible parallelization schemes.

As explained in Section 1.4.3, SimGrid's virtualization environment folds the processes of the program under simulation on execution contexts or co-routines that execute sequentially. This mechanism relies on `ucontexts`, which are part of the POSIX [62] standard. A `ucontext` consists of some stack space, a function to execute in that stack, and an interface with three main primitives to get, set, and swap `ucontexts`. They were originally designed as an evolution of the setjmp/longjmp functions, similar to *continuations* [56] in other languages.

A natural extension to execute the user code in parallel is to replace the `ucontexts` with full featured threads, one for each user process as illustrated by Figure 4.2a. However this approach is not adapted to our case for several reasons. First, every operating system has a hard limit on the number of threads that can be owned by a user (in the order of thousands), where we envision simulations with millions of processes. In addition, even if the simulation is small enough to not encounter system limits, this design is inefficient since the number of available CPUs to run the threads is usually much lower than the amount of threads to execute, resulting in a waste of performance due to the almost permanent contention for the CPUs and the required context switches between concurrent threads.

A more appealing alternative that does not suffer from the contention problem, nor from operating system limitations, is to distribute the scheduling jobs among a pool of worker threads as shown in Figure 4.2b. This approach is not as simple to implement as the previous one, because it needs to split the running state of every process from its execution context. However, this can still be achieved by using the `ucontexts`, that allow to manipulate the execution context as a first order value, similarly to a continuation. Hence, the new design still uses a `ucontext` for each simulated process, but these are scheduled by a pool of worker threads in parallel, a combination that only recent operating systems allow, letting us to take advantage of the best features of each alternative : light co-routines handled entirely in user space for each simulated process, and their parallel execution using a reduced number of threads according to the amount of available CPUs or cores.

Figure 4.3 depicts a parallel scheduling round $\mathrm{t_n}$ based on a pool with two working threads $\mathrm{T_1}$ and $\mathrm{T_2}$. $\mathrm{P_{t_n}}$ has four user processes that execute in the contexts $\mathrm{U_1}$, …, $\mathrm{U_4}$. Here, the workload is balanced between the threads that run in parallel. However, inside each thread the scheduling mechanism remains sequential, the only difference is that now the context swapping is done between the thread's context and the user contexts. An important difference is that at the end of each call to the $\mathrm{parallel\_schedule()}$ function there is a barrier that synchronize the worker threads, point at which the maestro thread is resumed, after all the contexts have been scheduled.

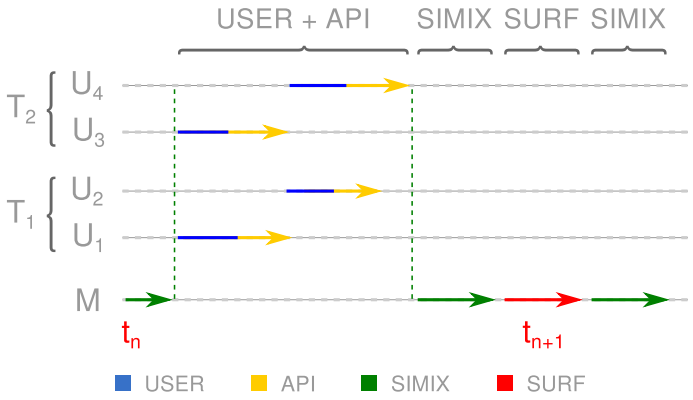


FIGURE 4.3: Parallel Scheduling Round.

$$
\begin{aligned}
C_{seq} &= \sum_{t_i \in S(P,R,M)} \Big(C_{surf}(R, M) + C_{smx}(|P_{t_i}|) + C_{usr}(P_{t_i}) \Big) \\
C_{par} &= \sum_{t_i \in S(P,R,M)} \Big(C_{surf}(R, M) + C_{smx}(|P_{t_i}|) + C_{thr}(|T|) + \max_{w \in T}(C_{usr}(P^{w}_{t_i})) \Big)
\end{aligned}
$$

FIGURE 4.4: Computational cost of sequential and parallel simulations.

4.3 Analysis of the Cost of Simulations

It is a common belief that parallel execution always leads to performance improvement. The simulation is however an inherently sequential problem (see Section 1.2.2 at page 46), and it is important to know the different costs at play to understand the implementation trade-offs and predict the scenarios for which a benefit can be expected compared to the standard sequential execution.

Figure 4.4 shows an approximation of the computational cost of the sequential simulation loop (Algorithm 5 on page 115) and that of the parallelized one (Algorithm 6 on page 116). In both cases it is estimated adding up the costs of each individual scheduling round (loop iterations). C_{surf} represents the cost of computing the time of the next ending actions (or timestamps), incurred in the call to *surf_solve()*. It depends on both the size of the platform R and the precision of the models M. C_{smx} is the cost of SIMIX to handle the requests issued by the processes. This is a function of the number of processes since each process issues one request per scheduling round. C_{usr} is the cost of the user code plus its scheduling (swap of execution contextes). It depends on the complexity of the computations executed by each user process, whose execution time differs from one simulation usage to another. Particular to the parallel execution, T is the set of worker

threads, $\mathrm{C_{thr}}$ is the cost of their synchronization that depends on their number, and $\mathrm{P}^{\mathrm{w}}_{\mathrm{t_i}}$ is the subset of processes in $\mathrm{P_{t_i}}$ scheduled by the thread w.

The following equation details the criterion to decide whether the parallel execution can outperform the sequential one in a given setting.

$$\mathrm{C_{par}} < \mathrm{C_{seq}} \Leftrightarrow \sum_{\mathrm{t_i} \in \mathrm{S(P,R,M)}} \left(\mathrm{C_{thr}}(|\mathrm{T}|) + \max_{\mathrm{w} \in \mathrm{T}} (\mathrm{C_{usr}}(\mathrm{P}^{\mathrm{w}}_{\mathrm{t_i}})) \right) < \sum_{\mathrm{t_i} \in \mathrm{S(P,R,M)}} \mathrm{C_{usr}}(\mathrm{P_{t_i}})$$

Intuitively, if the proportion of blue arrows (user code) is dominant in Figure 4.3, then a significant improvement can be expected from the parallel execution. In other words, the size of the user code greatly impacts the potential gain of the parallel execution.

A simulation of a few processes with very small local computations would not benefit from the parallel execution of the user processes since the performance cost of the thread synchronization would not be amortized by the parallel execution. When $\mathrm{C_{usr}} \to 0$, the performance penalty of parallelism for a simulation of K scheduling rounds is given by

$$\mathrm{C_{par}} - \mathrm{C_{seq}} \approx \mathrm{K} \cdot \mathrm{C_{thr}}(\mathrm{T}) \tag{4.1}$$

Another observation is that the precision of the models (ε) also has a big impact on the potential gains of the parallelism. It is important to recall that precision is not the same that the accuracy, as this last depends on the kind of study envisioned in the experiences. Many simulations can be performed using lower precisions without affecting their realism, as the properties observed might be insensitive to the precision in certain intervals. As previously mentioned, in DES the time is discretized into many points where the state of the system changes. The precision ε determines the minimal possible amount of time between two of these points. The smaller ε is, the larger the amount of possible timestamps $\mathrm{t_i}$ in $\mathrm{S(P, R, M)}$ gets, as a same time interval can be divided into more timestamps. Then, if the total amount of work done by the application remains constant (i.e. the amount of exchanged messages), with a precise model it can possibly be distributed across more timestamps, resulting in more numerous but smaller scheduling sets $\mathrm{P_{t_i}}$, and thus less user code executed in parallel and a higher cost of thread synchronization (K gets bigger).

This characterization allows us to analyze where the performance losses come from. First, multi-threading does not come for free, locking and unlocking the threads has a cost that depends on their number, so potential parallelism between the code segments to execute in parallel should be large enough to amortize this cost. Second, the barrier that synchronizes the threads at the end of each scheduling round is an idling point, and

the simulation can continue only when the last thread reaches the barrier. Load balancing between threads is thus an important aspect. In the next section we present a solution to these two problems.

4.4 An Efficient Pool of Worker Threads

According to the equation 4.1, the performance of the thread pool used to implement the parallel execution of the user code is critical for the efficiency of the simulation. In typical scenarios where the processes do not perform big computations (as presented in Section 4.7), the number of unavoidable system calls involved to control the worker threads in each scheduling round have an elevated computational cost. Hence, an efficient thread pool implementation must minimize these as much as possible. More precisely, there are two aspects of a thread pool that are critical to its performance : the control of the worker threads, that is tightly related to the underlying synchronization scheme, and the even distribution of the work among processes, to minimize the idling threads at the end of each simulation round.

In this section we detail the main design decisions that were taken to implement an optimized thread pool that incurs the lowest amount of system calls possible per iteration of the simulation loop.

4.4.1 The Control of the Worker Threads

When a list of tasks is ready to be processed by the pool, an *Apply* function stores a reference to the list in a variable whose location is known to all the members of the pool, then it unblocks the threads, and finally it blocks itself waiting for them to finish. This synchronization scheme can be easily implemented using primitives provided by the POSIX standard, that allow to write portable applications even if each operating system supports different system calls as the building blocks for thread synchronization. For example, Linux has Futexes, while BSD has Spin-locks, and their direct usage remains tedious and should be reserved to advanced users.

A simple way of implementing the thread control scheme of this *Apply* function is to use two condition variables. One to signal the arrival of the tasks to process, and one to signal the end of their processing. More precisely, for this latter functionality, one needs to mimic the behavior of a barrier, used to wait for all the threads to finish processing their tasks. Unfortunately these are not directly part of POSIX.

Algorithm 7 Barrier(barrier b, int num_threads)

```
1: pthread_mutex_lock(b.lock)
2: b.thread_waiting ← b.thread_waiting + 1
3: if b.thread_waiting == num_threads then
4:    b.thread_waiting ← 0
5:    pthread_cond_broadcast(b.proceed)
6: else
7:    pthread_cond_wait(b.proceed, b.lock)
8:    pthread_mutex_unlock(b.lock)
9: end if
```

Although it is possible to construct one from the pthreads primitives as shown in Algorithm 7, this poses serious difficulties from the performance point of view. The typical way of instrumenting a barrier is using a condition variable and a counter. Conceptually, when the threads enter the barrier they increment the counter and block in the condition variable. When the counter reaches the amount of threads indicated by the user, a broadcast is sent to release all the threads. The issue with this implementation is that the POSIX specification requires a mutex to protect every condition variable, even if there is no real need of mutual-exclusive access, as is the case here. One might argue that it is mandatory to avoid race conditions when incrementing the counter, however this can be also achieved using an atomic operation, common in many modern processors. The use of a mutex just to protect the condition variable introduces an unnecessary overhead, due to the system calls involved to resolve the contention when acquiring the mutex, and when releasing it.

Because the overhead of the parallelization should be minimized as much as possible our approach provides a specialized synchronization abstraction named *events*. Conceptually, an event is a combination of a condition variable and a barrier that has two primitives : *signal* and *wait*. Every event is associated to a group of threads waiting for a signal, that is issued by a given controller thread. Signaling an event is a blocking operation, that releases the threads waiting on it. They perform their computations and when done they simply block again in the event by waiting for the next signal. Once every thread is blocked again, the controller thread is released.

The event abstraction is implemented using a combination of Linux's futex_wait / futex_wake system calls, and several atomic operations. Both system calls accept a *ref-*

a Signal(event e)

```
1: myflag ← e.done
2: e.thread_num ← 0
3: e.work ← e.work + 1
4: futex_wake(&e.work, e.num_workers)
5: futex_wait(&e.done, myflag)
```

b Wait(event e).

```
1: myflag ← e.work
2: atomic(mycount ← e.thread_num + 1)
3: if mycount = e.num_workers then
4:     e.done ← e.done + 1
5:     futex_wake(&e.done, 1)
6: end if
7: futex_wait(&e.work, myflag)
```

FIGURE 4.5: The event interface.

erence to an integer value as the first argument, and an integer value as the second one. Function futex_wait(&a, b) atomically verifies that $*a == b$ and if true it sleeps at &a awaiting to be awaked, and futex_wake(&a, b) wakes at most b threads sleeping at &a.

Figure 4.5 presents the pseudo-code of the Signal and Wait functions of the event interface, and Figure 4.6 shows an execution cycle of these, with a caller C and $T_1, \ldots, T_n$ worker threads.

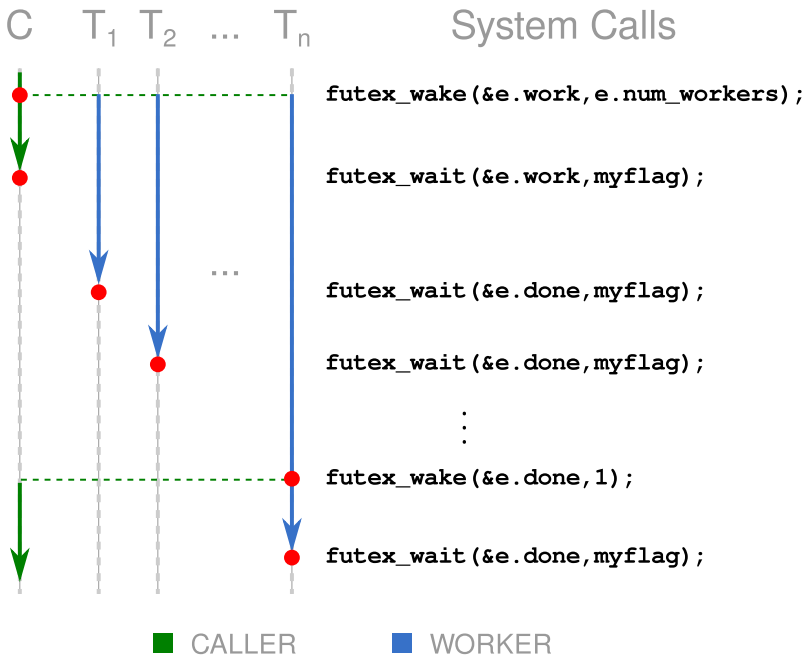


FIGURE 4.6: An Execution of the Event Abstraction.

Both functions accept an event e as the only argument. An event has four integer

fields associated : work, done, thread_num, and num_workers. The first two are used by the futex system calls to block/unblock the caller thread and the worker threads respectevely. The field thread_num keeps count of the number of workers that called Wait on the event, and num_worker is a constant field that indicates the total number of threads associated to the event. After the initialization of the event, all the worker threads participating on it are blocked at line 7 of Wait in the futex_wait(&e.work, myflag). When the caller thread calls Signal, it unblocks the worker threads (at line 4), and then it blocks himself in the next line futex_wait(&e.done, myflag) (the two first system calls in Figure 4.6). Next, the workers perform their computations, and when done they call Wait, that atomically increments the counter of worker threads waiting for a new signal in the event. To block, these issue the system call futex_wait(&e.work, myflag) again. Moreover, the last worker thread that issues Wait wakes the caller (at line 5) before blocking itself.

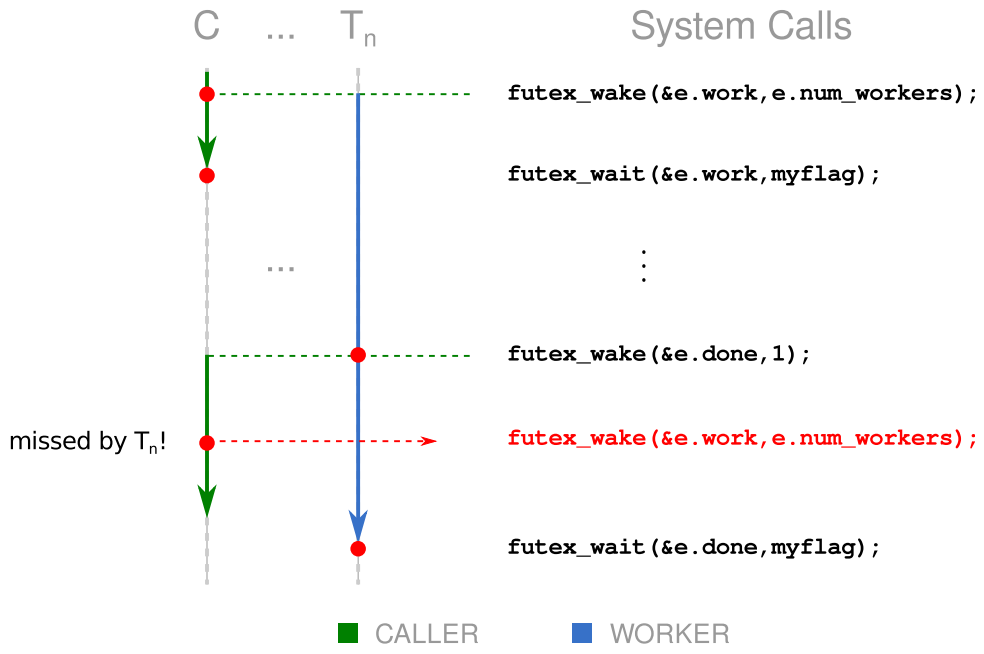


FIGURE 4.7: Missed Futex Wake System Call.

The variable myflag is used to avoid missing signals in both directions. A possible scenario where this can happen is shown in Figure 4.7, where the last thread T_n, which unblocks the caller at line 5 of the Wait function, does not block at line 7 before the caller issues a new signal, therefore it misses the futex_wake system call and leads to a deadlock situation. In that particular case, the new call to Signal increases the value of work (at line 3), hence when the delayed worker thread calls futex_wait(&e.work, myflag) at

line 7, the comparison $\mathrm{e.work} == \mathrm{myflag}$ fails (recall that myflag has the old value of work saved at line 1 of Wait), letting the thread continue as it would have catched the signal.

Thanks to this design the optimal cost in the number of system calls $\mathrm{N}+3$ is achieved (being N the number of worker threads). For each call to the apply function there is one system call to unblock all the worker threads, one to block the signaler, then one for each worker when finishing the computations and waiting again, and finally one to resume the signaler. For the moment this functionality is Linux dependent, but fallback implementations using classical POSIX synchronization primitives are provided for other systems.

4.4.2 Task Assignment

An advantage of using a pool of worker threads to run the processes in parallel is that they are naturally isolated. After the modifications introduced in Section 2.4 they can not modify the shared state of the simulator. The only unavoidable piece shared data among all the worker threads that remains is the list of tasks to process, that should be evenly assigned to avoid unbalanced workloads for optimum performance while avoiding race conditions.

A simplistic approach is to protect the list of tasks with a mutex, but it reveals very inefficient. Under this scheme, to fetch a task every worker has to compete to acquire the lock, remove the element from the list, and release the lock. This would transform the list in a bottleneck, due to the high degree of contention.

Our approach strives to avoid the use of any kind of synchronization mechanism. Instead, we use an array of tasks that allows direct access to the elements, and an index pointing to the next task that should be processed that is atomically incremented by each worker thread during the fetch. In this scheme there are no blocking operations involved and thus no expensive system calls. Finally, workers fetch tasks on demand, therefore the imbalance is minimized. Idling threads can only exists when no more tasks remain to be processed, and hence they have to wait in the barrier at the end of the apply function, until all the threads have done executing their last tasks.

4.5 Experimental Settings

The forthcoming experiments rely on two use cases. The first one consists of a classic parallel matrix multiplication algorithm (PMM) for a grid of processors using a *double-diffusion* communication pattern at each iteration. It was chosen because of its symmetric nature and because of the large amount of computation run by each process at each step.

The second case uses the well-known Chord [59] peer-to-peer DHT (Distributed Hash Table) protocol. It is designed to be scalable and capable of functioning even with nodes leaving and joining the system. The nodes form a logical ring and maintain local routing information called a *finger table*. Each node keeps information about $\mathrm{O}(\log \mathrm{n})$ other nodes, where n is the total number of nodes in the system. Periodically, the nodes exchange messages to update their knowledge about the logical ring and their neighbors, and the system eventually converge to a consistent global vision. It has been shown that any lookup request issued by a node is resolved with only $\mathrm{O}(\log \mathrm{n})$ messages generated in the network. Chord was chosen because it is representative of a large body of algorithms studied in the P2P community.

We performed an experiment similar to the one of [3]. We consider n nodes that all join the Chord ring at time $\mathrm{t} = 0$. Every node performs a *stabilize* operation every 20 seconds, a *fix_fingers* operation every 120 seconds, and an arbitrary *lookup* request every 10 seconds. When the simulated time has reached 1000 seconds, each node stops its process and the simulation ends. To ensure that experiments are comparable between different settings we tuned the parameters to ensure that the amount of applicative messages exchanged during the simulation, and thus the workload onto the simulation kernel, remains comparable (with $100,000$ nodes, about 25 million messages are exchanged in this scenario).

Each experiment was run on a machine of Grid'5000 [8] with two AMD 12-core CPUs at 1.7 GHz and 48 GB of RAM. SimGrid v3.6 beta (git version 8d32c7) was used for these experiments.

4.6 Cost of Thread Synchronization

The overhead introduced by the synchronization primitives that control the thread-pool is a determining factor of the potential achievable speedup of the parallel execution. Because of this, we first measure the overhead ($\mathrm{K} \cdot \mathrm{C}_{\mathrm{thr}}(\mathrm{T})$) by comparing the standard sequential simulation time to a parallel execution that uses a thread pool with just one

thread. The pool with a single worker thread adds the synchronization costs without providing any speed up to the simulation, allowing the measurement of the threading overhead.

For PMM, the cost of synchronization is negligible because the number of scheduling rounds is relatively low and so is the number of system calls involved. However, in the case of Chord it exhibits an overhead of 16% (*i.e.* an increment of 76 s in a simulation of 471 s). This cost, which remains relatively high despite the high level of system optimization we did, clearly demonstrates that parallel execution is only beneficial in some specific conditions that we now try to better characterize.

4.7 Evaluation of the Parallelization Speed-Up

We now present a set of experiments that summarizes the scenarios where the parallelism can outperform the sequential execution. In cases like PMM where C_{usr} is large, the parallel execution trivially outperforms the sequential one. For PMM using 9 nodes with matrices of size 1500, the sequential simulation takes 31 s and the parallel one with 4 threads takes 11 s, representing a speedup of more than 50%.

The case of Chord is more challenging due to C_{usr} being very small, as is typical for most peer-to-peer applications : processes exchange a lot of messages and perform few calculations. In this case, we study the impact the model's precision over the potential parallelism of the simulation. This is done by measuring the average amount of user processes ready to run at each scheduling round ($|P_{t_i}|$) for several values of the precision ε, that defines the smallest possible increment of the simulation time in a simulation round. Results presented in Table 4.1 clearly show that small values of ε (corresponding to important precisions) lead to reduced scheduling sets. This can be explained by the dispersion of actions, whose amount remains constant, across more timestamps. Better speedups are thus expected for big values of ε, *i.e.* for moderate simulation precision.

ε	10^{-5}	10^{-3}	10^{-1}	Constant network
$\lvert P_{t_i} \rvert$	10	44	251	7424

Table 4.1: Average scheduling set size in Chord as a function of ε, the simulation precision.

Figure 4.8 reports the obtained running times as a function of the number of nodes (from 1000 to 2 millions nodes). The top graph represents the results for a constant net-

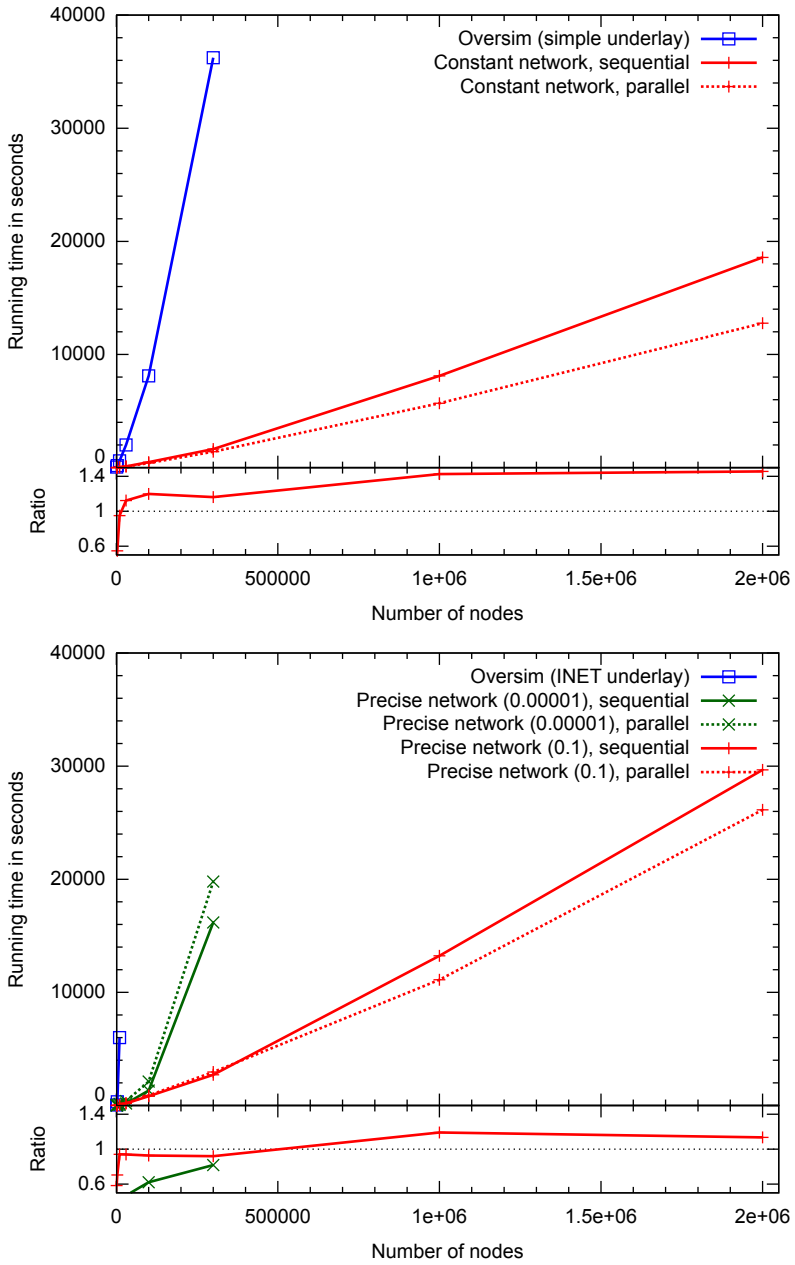


FIGURE 4.8: **Top** : Running times of the Chord simulation with a constant network model on SimGrid, compared to Oversim with a simple underlay. **Bottom** : Running times of the Chord simulation with a precise network model on SimGrid, compared to Oversim with the INET underlay. The bottom part of each graph shows the ratio between the parallel and the sequential modes (when the curve is above 1, the parallel mode is faster than the sequential mode).

work model, which consists in applying a constant delay of 0.1 second for each communication. We were able to simulate 2 million nodes in 3h18 with the sequential mode, and in 2h24 in parallel mode. The ratio depicted below the graph shows that the parallel mode is preferable to the sequential mode even for small scenario sizes and provides a speedup of up to 40% (with 24 threads).

The bottom graph shows the same experiment with the precise network model described in [66]. As explained before, the numerical precision used to represent the time has an impact on the performance of parallel execution. Using the default value ($\varepsilon = 10^{-5}$), we could only simulate up to 300k nodes in less than one night of computation. When the precision is reduced to $\varepsilon = 10^{-1}$, the amount of timestamps at wich events can occurr is reduced, which ensures that the simulation consists of less but bigger scheduling sets. In this conditions, the synchronization costs are betterly amortized by the amount of work done in parallel, making the whole approach profitable. In these conditions, we were able to simulate 2 million nodes in 8h15 in sequential mode, and in 7h15 in parallel mode. The relative gain of the parallel mode is smaller than previously : it becomes beneficial only for 500,000 processes and more and the speedup remains under 20%.

A more disturbing result is the fact that the sequential simulation performance is also highly impacted by ε. This is unexpected since the amount of work remains the same in each case, and understanding the cause of this phenomenon remains to be done in future work, although preliminar analysis suggest this is due to cache pollution effects.

The memory usage for simulating 2 million nodes was about 36 GB, that represent 18 kB per node, including 16 kB for the user stack.

4.8 Comparison to OverSim

Figure 4.8 also reports the simulation timing obtained with OverSim [3] on the same machine. On the top, OverSim was used with the simplest network underlay, comparable to our constant time model. We could not simulate more than $300,000$ nodes in less than a night of computation with this settings.

On the bottom, we also plot the running times we obtained with the INET network model of OverSim. Relying on a packet-level simulation, it offers the best accuracy of the framework and is thus comparable to the precise model of SimGrid. In this setting, the simulation failed to reach $30,000$ nodes : the initialization phase was not finished after a night of computation.

Those results show that the new design of SimGrid reaches high scalability and performance, including with precise network models. In particular, SimGrid's worst case (precise model, $\varepsilon = 10^{-5}$) outperforms OverSim's best case (constant model) while the offered simulation accuracy is much better.

4.9 Summary

In this chapter we addressed the problem of parallelizing the simulations. We presented a parallel architecture that mixes `ucontexts` and threads, and thus avoids wasting resources in unwanted contention resolution, yet allowing the simulation of millions of processes. We implemented and analyzed an efficient thread pool data structure that uses a custom synchronization construction which relies directly on the operating system primitives to reduce the cost of thread synchronization. We analyzed the cost of sequential and parallel execution and gave an analytical criterion to characterize situations in which a speedup can be expected from parallel execution. We validated this criterion experimentally alongside with the other design choices made in the newest version of the SimGrid framework. Finally, we showed that SimGrid is capable of simulating peer-to-peer systems to an unprecedented scale, one order of magnitude faster and with better accuracy than OverSim.

Overall, we think that this work demonstrates the difficulty to get a parallel version of a P2P simulator faster than its sequential counterpart, provided that the sequential version is optimized enough. During the work leading to this results, we encountered several situations where the parallel implementation offered nearly linear speedups, but these always resulted from blatant performance mistakes in the sequential version. We hope that this return of experience and our work on the design of the framework will be applicable to other simulation toolkits as well.

Chapitre 5

Conclusions and Future Work

5.1 Conclusions

Distributed systems have an increasingly central role in the future of computing. The enormous amount of distributed applications underlying the infrastructure of many business operations, and scientific research make this technology a central concern for the future of world's economy and sustainability. Even if these systems are widely deployed, they are still hardly understood. Ensuring that a distributed systems is robust and efficient in all relevant situations remains highly challenging. Therefore, the contributions in this thesis build towards easing the study of the performance and correctness of distributed systems.

In Chapter 2 we first tried to close the existing gap between simulation and model checking of distributed systems. These two methodologies allow to study the performance and the correctness of the systems respectively. To our knowledge this is the first work specifically directed towards their integration in a single tool. Before this work, the developers willing to assess the performance and the correctness of a distributed system had to develop custom models or prototypes for each methodology, that carried an elevated cost and in general prohibitive. The key challenges for the success of such integration resides in designing an architecture that adapts to the requirements of both methodologies while not affecting their performance. We have shown that due to the requirements on the shared state of the program under study imposed by the model checker, operating systems designs serve as a guideline for a common architecture. This also resulted benefical for the general performance of the simulations thanks to the more compact code base, as reflected by the experiences.

Particular to model checking, the main contribution of this thesis is an approach to cope with the state space explosion problem based in dynamic partial order reduction. The difficulty of using DPOR in the context of SimGrid resides in the fact that the programs are written using the APIs offered by the simulator that lack any formal specification. The solution presented in Chapter 3 consists of introducing an internal set of networking primitives with full formal semantics, used to express all the user-level communication APIs on top of it. The exploration with DPOR is performed at the level of the primitives, and thus can deal with any of the APIs in SimGrid. The verification experiments show that the reductions obtained with this approach are significant, and that the model checker is capable of finding bugs in non trivial programs such as implementation of the Chord protocol. Moreover, the Chord verification experience shows that the idea of unifying the simulation with the model checking in a single framework is promising. The bugged Chord implementation was originally developed to evaluate the performance of the simulator, but thanks to the integration we were able to model check it just by introducing a few assertions and changing a flag in the command line.

Concerning the assessment of performance, we worked towards the scalability of CPU bound simulations. Traditionally, the simulation of distributed programs is sequential due to the limitations of classical parallelization techniques, that are mostly tailored towards parallel discrete event simulation of system models. However, current multicore architectures impose a parallel paradigm to improve the performance of applications. In Chapter 4 we proposed a different approach to parallelize the simulations of distributed programs, that keeps the classical simulation loop sequential but executes the simulated processes in parallel. After analyzing the time complexity of the approach, and comparing it to the standard sequential execution, we conclude that there are two scenarios that can benefit of the parallelization : HPC applications that perform big local computations, and typical P2P applications with a large amount of processes. Moreover, transversal to these two categories, we discovered that the precision of the models that simulate the resources behavior is a determinant of the potential parallelism of our approach. Therefore, according with the kind of envisioned study it is possible to adjust this precision to obtain faster, but still useful simulations. Finally, after the experimental results from the case studies, we can say that SimGrid is at the moment one of the most scalable, accurate, and fast simulation suites available to study distributed system.

5.2 Future Work

As future work, we envision different lines of work on each area of the contributions.

Software verification. SimGridMC is currently restricted to the verification of safety properties such as assertion violations or the detection of deadlock states. The verification of liveness properties would require us to detect cycles, which is currently impossible due to the stateless approach. For similar reasons, state exploration is limited by a (user-definable) search bound. We intend to investigate hybrid approaches between stateful and stateless model checking that would let us overcome these limitations, however for this it is necessary to modify the DPOR algorithm.

The current integration of SimGridMC into SimGrid is only architectural. The functionality is still completely separated, the user can either simulate the application or verify it. An interesting line of future research is to allow to use both techniques at the same time. The rationale behind the idea is that recent trends in distributed systems suggest that the notion of correctness is not only related with the behaviors of the application but also to its performance aspects. For many applications, not attaining a certain minimum level of performance is as bad as a crash. For example, in Cloud computing resource costs are paid on demand thus it is critical to the success of a business to have guarantees regarding the computational costs of the application. Energy consumption and dissipation is another variable that becomes essential when designing data centers and HPC clusters. An attractive option is to extend SimGrid, to enable the verification of performance properties by exhaustively exploring the state space of the programs but taking into account the constraints and timings enforced by the simulated platform. To visualize the approach, consider the Figure 1.2c in page 60. The idea is to explore all possible executions with its timings, instead of a single trace as in the figure. The impact of this work would simplify the optimization of distributed systems. For example, it adds the ability to automatically find best/worst execution traces in terms of consumed bandwidth, the number of messages exchanged, latency, or consumed energy. Moreover, leaving the performance variables aside, it would allow classical state space exploration, but only considering the states that are reachable in the platform desired by the user.

Simulation To further improve the scalability, and speed of the simulator, we plan to follow a similar approach to parallelize other steps of the simulation loop, while still keeping the whole simulation sequential. For example, the handling of process requests

(line 5 of Algorithm 6 in page 116) could be done in parallel using again a pool of worker threads. Because processing the requests of the processes involve modifying shared data structures, it would be necessary a carefull study and modification of SIMIX v2.0's internals to minimize the shared state affected by each request.

Another line of work that we consider promising is related with the expressiveness of the networking primitives. Despite, the current four operations are enough to write SimGrid's communication APIs on top of them, we think it is important to have a fine-grained control over all the possible events of the communications. This would allow to express richer communication schemes such as GOAL [27], a domain specific language designed to write group communication operations. To illustrate this, consider the `Wait` and `Test` primitives introduced in Section 2.4.1 in page 75. These only allows to block or detect if a given communication is finished or not. However, it would be also interesting to detect other steps in the communication, like blocking until the matching end is found, or until the communication has started, but not finished, etc. This would also imply extending the formal specification of the networking primitives to consider them during model checking.

The work of this thesis aims to develop the theory and tools required to provide a unified framework for the study and development of distributed computer systems, capable of assessing performance aspects as well as correctness properties directly on executable programs. We think that an appealing approach is to close the gap between simulation and model checking unifying both methodologies in the same framework. Having such tool can greatly simplify the development of correct distributed systems, as it eliminates the cost of writing multiple models for the same application. With the complexity of distributed programs on the rise, it becomes necessary to develop new methodologies and tools to help the developer better understand the behavior of these systems. Distributed systems are in the mainstream of information technology.

Bibliographie

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[2] F. Baccelli, D. R. McDonald, and J. Reynier. A mean-field model for multiple TCP connections through a buffer implementing RED. *Perform. Eval.*, 49 :77–97, September 2002.

[3] I. Baumgart, B. Heep, and S. Krause. OverSim : A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[4] W. H. Bell, D. G. Cameron, L. Capozza, A. P. Millar, K. Stockinger, and F. Zini. OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications*, 17(4), 2003.

[5] G. Bianchi. Performance analysis of the IEEE 802.11 distributed coordination function. *Selected Areas in Communications, IEEE Journal on*, 18(3) :535 –547, mar 2000.

[6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[7] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs (extended abstract). In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, pages 172–186, London, UK, 1997. Springer-Verlag.

[8] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000 : A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, Winter 2006.

[9] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata : Application to model-checking. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97 : Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer Berlin / Heidelberg, 1997.

[10] J.-Y. L. Boudec, D. McDonald, and J. Mundinger. A generic mean field convergence result for systems of interacting objects. In *Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, pages 3–18, Washington, DC, USA, 2007. IEEE Computer Society.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : $10\hat{2}0$ states and beyond. *Inf. Comput.*, 98 :142–170, June 1992.

[12] R. Buyya and M. Murshed. GridSim : A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *J. of Concurrency and Computation : Practice and Experience (CCPE)*, 14(13-15), Decembre 2002.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. CloudSim : A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. *Software : Practice and Experience*, 41(1) :23–50, Jan. 2011.

[14] H. Casanova, A. Legrand, and M. Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[15] K. M. Chandy and R. Sherman. Space-time and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57, 1989.

[16] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50 :752–794, September 2003.

[17] J. Esparza and S. Schwoon. A BDD-Based Model Checker for Recursive Programs. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer Berlin / Heidelberg, 2001.

[18] A. Ferscha. *Parallel and Distributed Simulation of Discrete Event Systems*. McGraw-Hill, 1995.

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

[20] S. Floyd and V. Jacobson. Traffic Phase Effects in Packet-Switched Gateways. *SIGCOMM Comput. Commun. Rev.*, 21 :26–42, April 1991.

[21] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM - Special issue on simulation*, 33(10) :30–53, October 1990.

[22] V. K. Garg, Ph.D. *Elements of Distributed Computing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[23] T. M. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. P2PSim. http://pdos.csail.mit.edu/p2psim/, 2005.

[24] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 176–185, London, UK, 1991. Springer-Verlag.

[25] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*. Springer, New York, USA, 1996.

[26] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of programming languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM.

[27] T. Hoefler, C. Siebert, and A. Lumsdaine. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 574–581, Washington, DC, USA, 2009. IEEE Computer Society.

[28] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23 :279–295, May 1997.

[29] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. 16th IEEE Intl. Conf. Automated software engineering (ASE 2001)*, pages 254–261, Washington, DC, USA, 2001. IEEE Computer Society.

[30] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. PeerSim. http://peersim.sourceforge.net/.

[31] T. Kiesling and S. Pohl. Time-parallel simulation with approximative state matching. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, pages 195–202, 2004.

[32] C. Killian, J. W. Anderson, R. Jhala, and A. 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*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association.

[33] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace : language support for building distributed systems. In *Proc. ACM SIGPLAN Conf. Programming language design and implementation (PLDI 2007)*, pages 179–188, San Diego, CA, USA, 2007. ACM.

[34] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace : language support for building distributed systems. *SIGPLAN Not.*, 42 :179–188, June 2007.

[35] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, July 1978.

[36] L. Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.

[37] A. M. Law and D. W. Kelton. *Simulation Modelling and Analysis.* McGraw-Hill Education - Europe, 2000.

[38] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[39] B. Liu, D. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency : fluid simulation vs. packet-level simulation. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1244 –1253 vol.3, 2001.

[40] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27 :67–82, July 1997.

[41] S. McCanne, S. Floyd, and K. Fall. The Network Simulator (ns2). http://nsnam.isi.edu/nsnam/.

[42] M. Mendez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimization for amorphous data-parallel programs. In *PPoPP '10 : Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2010. ACM.

[43] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12 :1094–1104, October 2001.

[44] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC : A pragmatic approach to model checking real code. In *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[45] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 362–371, New York, NY, USA, 2008. ACM.

[46] M. Musuvathi, S. Qadeer, and T. Ball. Chess : A systematic testing tool for concurrent software, 2007.

[47] T. Ott, J. Kemperman, and M. Mathis. Window Size Behavior in TCP/IP with Constant Loss Probabillity. In *4th IEEE Workshop on High-Performance Communication Systems*, June 1997.

[48] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput : a simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 303–314, New York, NY, USA, 1998. ACM.

[49] R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *Proc. ACM Wsh. Parallel and distributed systems : testing and debugging (PADTAD 2007)*, pages 43–53, London, UK, 2007. ACM.

[50] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 377–390, London, UK, 1994. Springer-Verlag.

[51] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46 :12–25, June 2011.

[52] J. Pujol-Ahulló, P. García-López, M. Sànchez-Artigas, and M. Arrufat-Arias. An extensible simulation tool for overlay networks and services. In *SAC '09 : Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2072–2076, New York, NY, USA, 2009. ACM.

[53] S. Qadeer and J. Rehof. Context-Bounded Model Checking of Concurrent Software. In *TACAS'05*, pages 93–107. Springer, 2005.

[54] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22 :416–430, March 2000.

[55] K. Ranganathan and I. Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing – HPDC'02*, page 352, Washington, DC, USA, 2002. IEEE Computer Society.

[56] J. C. Reynolds, A. V. Wijngaarden, A. W. Mazurkiewicz, F. L. Morris, C. P. Wadsworth, J. H. Morris, M. J. Fischer, and S. K. Abdali. The discoveries of continuations, 1993.

[57] G. F. Riley. Large-scale Network Simulations With GTNETS. In *Winter Simulation Conference*, 2003.

[58] R. Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 101–, Washington, DC, USA, 2001. IEEE Computer Society.

[59] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan. Chord : a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1) :17–32, Feb. 2003.

[60] E. Strohmaier. Japan Reclaims Top Ranking on Latest TOP500 List of World's Supercomputers. http://www.top500.org/lists/2011/06/press-release, June 2011.

[61] D. Terry. TechPack - cloud computing. http://techpack.acm.org/cloud/, 2010.

[62] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. IEEE, New York, NY, USA, 2004.

[63] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Reduced Execution Semantics of MPI : From Theory to Practice. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 724–740, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 156–165, London, UK, 1991. Springer-Verlag.

[65] A. Varga. Omnet++ community site. http://www.omnetpp.org/.

[66] P. Velho and A. Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *Proc. of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10, 2009.

[67] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–12, Washington, DC, USA, 2000. IEEE Computer Society.

[68] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[69] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. *SIGPLAN Not.*, 44(4) :261–270, 2009.

[70] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of the 4th International Conference on Concurrency Theory*, CONCUR '93, pages 233–246, London, UK, 1993. Springer-Verlag.

[71] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall : Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.

[72] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST : transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[73] V. yee Vee and W. jing Hsu. Parallel discrete event simulation : A survey. Technical report, Centre for Advanced Information Systems, SAS, Nanyang Technological University, 1999.