Exploration efficace de l'espace d'états adaptée aux programmes distribués asynchrone adaptation de la réduction d'ordre partiel basée sur les dépliages pour les programmes MPI

The Anh Pham

UNIVERSITE BRETAGNE LOIRE MATHSTIC

école normale supérieure rennes

THÈSE DE DOCTORAT DE

ÉCOLE NORMALE SUPÉRIEURE DE RENNES
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

The Anh PHAM

Efficient state-space exploration for asynchronous distributed programs

Adapting unfolding-based dynamic partial order reduction to MPI programs

Thèse présentée et soutenue à l'ENS RENNES, le 6 décembre 2019
Unité de recherche : IRISA / INRIA Rennes - Bretagne Atlantique
Thèse N° :

Rapporteurs avant soutenance :

Laure Petrucci Professeur, IUT de Villetaneuse, Université Paris 13.
Radu Mateescu Directeur de Recherche, Inria Grenoble - Rhône-Alpes.

Composition du Jury :

Président :	*François Taïani*	
Examinateurs :	Radu Mateescu	Directeur de Recherche, Inria Grenoble – Rhône-Alpes.
	Laure Petrucci	Professeur, IUT de Villetaneuse, Université Paris 13.
	Stefan Leue	Professor, Universität Konstanz (Deutschland).
	Stephan Merz	Directeur de Recherche, Inria Nancy – Grand Est.
Dir. de thèse :	Martin Quinson	Professeur, ENS Rennes.
	Thierry Jéron	Directeur de Recherche, Inria Rennes – Bretagne Atlantique.

ACKNOWLEDGMENT

Firstly, I would like to express my sincere thanks to my respectful supervisors: Thierry Jéron and Martin Quinson. They gave me this topic, helped me a lot during the past three years. I learned from them a lot, not only scientific knowledge but also skills for research, such as presentation skills, writing skills, and so on. I would like to thank them for patiently listening to me in every meetings, always motivating but not putting pressure on me over the three years.

I am grateful to all the jury members for reading the manuscript and participating in the defense of this thesis, especially the reviewers for instructive comments to improve my dissertation.

I am grateful to the Sumo team, the Myriads team, and the INRIA Rennes for helping me and giving me an excellent research environment, great friends, and colleagues. I could not help but mention Loic Guegan and Arif Ahmed who encouraged me when I was under difficulties, as well as had exciting discussions with me.

I thank the Vietnamese friends who have helped me a lot in my daily life. They gave me happy and warm times like a family. I always regard them as family members, making me feel less missed in my hometown and country.

And finally, it is impossible not to mention my family, which is strong support for me during the past difficult period. Thank my lovely daughter, Pham Diep Chi, for giving me the power to overcome all difficulties and challenges. This thesis is also a gift to my parents, who I love the most.

TABLE OF CONTENTS

RÉSUMÉ

Contexte

Les applications distribuées à passage de messages font partie intégrante de la technologie de l'information puisqu'elles produisent des performances supérieures à celles que l'on pourrait obtenir d'un seul ordinateur, afin de répondre aux tâches de calcul de plus en plus coûteuses d'aujourd'hui. La conception d'applications distribuées pose des problèmes de correction notoires. En plus des difficultés intrinsèques à la programmation concurrente (*e.g.,* possibilité de *race conditions*), la programmation distribuée ajoute d'autres difficultés, comme l'absence de mémoire et d'horloge centralisées. Les applications modernes de calcul haute performance (HPC) doivent faire face à toutes ces difficultés lorsqu'elles tentent d'agréger et d'exploiter pleinement la puissance de calcul d'un ensemble de nœuds multicœurs. Dans ce contexte, l'approche classique pour garantir la correction consiste à s'appuyer sur des modèles de communication rigides, afin d'éviter des scénarios de synchronisation complexes. Malheureusement, ces schémas de communication rigides passent mal à l'échelle. La taille des plates-formes de calcul modernes impose donc des applications dont les schémas de communication sont irréguliers et dynamiques. Cependant, il devient alors pratiquement impossible d'assurer l'exactitudela correction de ces applications par des méthodes classiques de test. Il en découle donc un fort besoin de nouveaux outils de vérification de correction qui reposent sur des méthodes formelles.

Les méthodes formelles [53] sont des techniques mathématiques appliquées à la conception et l'analyse des systèmes matériels et du logiciel. Parmi les méthodes formelles, les techniques de vérification par modèles ou *model-checking* ont fortement attiré l'attention des chercheurs en raison de leur efficacité. Le *model checking* a été inventé au début des années 1980 et développé indépendamment par Clarke et Emerson [10] et par Queille et Sifakis [40]. Après près de 40 ans de développement, elle est devenue une méthode populaire et efficace pour vérifier les systèmes matériels et logiciels. Étant donné un modèle d'un système et une propriété, la vérification par modèle vérifie automatiquement si la propriété est satisfaite sur ce modèle

[6]. La technique est automatique en ce sens qu'un model-checker s'exécute automatiquement sans nécessiter aucune intervention de la part des utilisateurs. Le principe consiste généralement à explorer tous les comportements possibles du modèle du système. Cependant, l'espace d'états augmente exponentiellement avec le nombre de processus concurrents. La réduction par ordre partiel (POR) et la technique de dépliage sont deux techniques alternatives nées dans les années 90 candidates pour atténuer l'explosion de l'espace d'état et s'étendre à des applications de grande taille.

Les techniques de POR comprennent un ensemble de techniques d'exploration partageant l'idée que pour détecter les blocages, il suffit de couvrir chaque trace de Mazurkiewicz, *i.e.* chaque classe d'entrelacements équivalents par commutation d'actions consécutives indépendantes (*i.e.,* deux actions sont indépendantes si l'une n'active ni ne désactive l'autre, et si leur ordre relatif n'a aucun effet sur le résultat final). Cette réduction de l'espace d'états est obtenue en choisissant, à chaque état exploré et en fonction de l'indépendance des actions, seulement un sous-ensemble d'actions à explorer (par exemple, ensemble persistant, *ample set*, ou *stubborn set*) ou à éviter (par exemple, *sleep set*).

La technique de réduction dynamique d'ordre partiel (DPOR) a été introduite plus tard pour réduire l'explosion de l'espace d'états pour la vérification sans mémorisation d'état (*stateless*) des programmes. Dans ce contexte, alors que POR repose sur une relation d'indépendance définie statiquement et potentiellement imprécise, DPOR peut être beaucoup plus efficace en collectant dynamiquement cette relation au cours de son exécution. Néanmoins, des explorations redondantes peuvent encore exister qui conduiraient à des entrelacements déjà visités, et détectés par l'utilisation de sleep-sets. C'est ce qu'on appelle exploration *sleep set blocked* (SSB). Éviter les SSB pour obtenir l'optimalité est un défi et peut compter sur les dépliages pour l'obtenir.

Les dépliages sont un concept de théorie de la concurrence qui fournit une représentation compacte des comportements d'un modèle sous la forme d'une structure d'événements qui agrége à la fois les dépendances causales, la concurrence entre les événements, et les conflits qui indiquent des choix dans l'évolution du programme. Cette représentation peut être exponentiellement plus compacte qu'une sémantique d'entrelacement, tout en permettant de vérifier certaines propriétés, comme les propriétés de sûreté. Chaque configuration maximale du dépliage d'un programme correspond à une trace de Mazurkiewicz représentant une classe d'équivalence des exécutions du programme. UDPOR [27] est une technique DPOR optimale qui combine les

forces des POR et des dépliages et explore les comportements d'un programme sans exploration redondante de traces de Mazurkiewicz.

Notre objectif est de vérifier des applications distribuées asynchrones, à savoir des programmes MPI, en l'absence de modèle, mais avec la possibilité d'exécuter le code dans l'environnement de simulation SimGrid et de tirer profit de la technique UDPOR. La vérification des modèles se heurte alors déjà à de sérieux problèmes, comme la détermination des états globaux du système et la vérification de l'égalité des états. L'application d'UDPOR aux applications distribuées asynchrones pose de nouvelles difficultés. La détermination de la dépendance entre actions nécessite de spécifier la sémantique des programmes. Cependant, du fait du nombre important de types d'actions différents et de la complexité de leurs effets, préciser formellement la sémantique est une tâche très laborieuse et sujette à erreur. De plus, le calcul des extensions d'une configuration peut être coûteux (en gros, une configuration représente une classe d'équivalence d'exécutions, alors que ses extensions désignent des états qui sont directement accessibles à partir des états au cours ces exécutions). Par exemple, ce problème est NP-complet dans les réseaux de Petri. Par conséquent, UDPOR nécessite des algorithmes ingénieux pour calculer efficacement les dépliages. Or il n'est pas possible d'adapter directement une solution existante conçue pour des programmes concurrents ayant seulement des mutex [34]. Pour ces raisons, nos contributions ainsi que nos solutions sont liées à cet objectif de recherche.

Contributions

— Nous définissons d'abord un modèle abstrait compact de programmes distribués asynchrones comprenant peu de primitives. Ce modèle abstrait peut être utilisé pour modéliser des applications distribuées asynchrones, en particulier pour simuler une grande classe de programmes MPI. Pour y parvenir, nous révisons et étendons un modèle abstrait existant de programmes distribués en ajoutant de nouvelles primitives de synchronisation. Bien que la plupart des applications distribuées soient asynchrones, il reste encore des tâches qui doivent être synchronisées (par exemple les appels MPI_Win_lock, MPI_Win_unlock dans la programmation RMA des programmes MPI). En ajoutant les primitives de synchronisation au modèle abstrait, il est possible de coder une plus grande classe de programmes MPI, mais le modèle abstrait reste toutefois compact avec seule-

ment neuf primitives. Ainsi, au lieu de décrire la spécification formelle de la sémantique non triviale du standard MPI, nous ne décrivons formellement, dans le langage de spécification TLA+, que ce modèle abstrait comprenant seulement quelques primitives. En effet, notre modèle abstrait est facile à spécifier et sa spécification formelle est concise. Avec cette spécification précise du modèle abstrait, on peut alors raisonner sur les relations d'indépendance utiles pour les techniques de réduction d'ordre partiel. De plus, à partir de cette spécification, nous prouvons que dans notre modèle, une action tirable ne peut pas être désactivée en exécutant d'autres actions (propriété de persistance). Cette propriété est la clé de l'efficacité d'UDPOR pour notre modèle.

— Comme annoncé, nous avons comme objectif de tirer parti d'UDPOR pour la vérification de programmes MPI, dans le cadre du simulateur SimGrid. Puisque le modèle abstrait est capable d'exprimer une grande classe de programmes MPI, et que les primitives de notre modèle abstrait correspondent étroitement à celles fournies par le noyau de simulation SimGrid, nous adaptons la technique UDPOR sur le modèle abstrait plutôt que sur le modèle de programmation MPI général. Cependant, quel que soit le modèle de calcul, la question se pose de savoir comment calculer efficacement les extensions d'une configuration. Nous avons donc proposé les étapes permettant d'adapter UDPOR aux applications MPI qui peuvent être simulées par le simulateur SimGrid, en nous concentrant sur des algorithmes intelligents pour calculer les extensions. En exploitant la persistance du modèle abstrait et les observations précises sur les configurations et les relations d'indépendance entre les actions, nous avons proposé des méthodes efficaces pour calculer efficacement ces extensions (en temps polynomial).

— L'algorithme UDPOR étant récursif, le calcul des extensions d'une configuration obtenue en ajoutant un nouvel événement à une configuration obtenue à l'étape précédente conduit a priori à des recalculs de nombreux événements. Nous avons donc proposé un algorithme incrémental pour notre modèle, éliminant presque tous ces calculs et ne s'appuyant que sur les actions qui sont tirables dans l'état de la configuration.

— Nous avons implémenté la version quasi-optimale d'UDPOR dans un prototype adapté à notre modèle de programmation abstrait, c'est-à-dire avec sa relation d'indépendance. Nous avons effectué quelques expériences et comparé UDPOR à une recherche exhaustive sans mémorisation d'état sur plusieurs critères. Le

prototype est encore limité, pas encore connecté à l'environnement SimGrid, et ne peut donc être expérimenté que sur des exemples simples. Cependant, grâce aux premiers résultats prometteurs obtenus par ces expériences, nous pensons pouvoir affirmer que UDPOR peut être utilisé pour vérifier des applications MPI tout en atténuant l'explosion de l'espace d'état de ces applications.

En ce qui concerne les publications scientifiques de la thèse, les deux articles suivant ont été publiés au cours de la thèse :

— **The Anh Pham**, Thierry Jéron, Martin Quinson: Verifying MPI Applications with SimGridMC, In Proceedings of the First International Workshop on Software Correctness for HPC applications, CORRECTNESS@SC 2017, Denver, CO, USA, November 2017.

— **The Anh Pham**, Thierry Jéron, Martin Quinson, Unfolding-Based Dynamic Partial Order Reduction of Asynchronous Distributed Programs. In Proceedings of the FORTE conference, Kongens Lyngby, Denmark, June 2019.

ABSTRACT

Context

Distributed message passing applications are in the mainstream of information technology since they produce higher performance than one could get from a single computer to meet today's increasingly heavy computational tasks. Writing distributed applications poses notorious correctness challenges. In addition to intrinsic concurrent programming difficulties (*e.g.,* possible race conditions), distributed programming adds some of its own, such as the lack of centralized memory and the lack of a centralized clock. Modern High-Performance Computing applications must take on all these difficulties when attempting to aggregate and fully exploit the computational power of a set of multi-core nodes. In this context, the classical approach to ensure correctness is to rely on rigid communication patterns, so as to avoid complex synchronization scenarios. Unfortunately, these rigid communication patterns scale poorly. The size of modern computer platforms, thus mandates applications with communication patterns that are irregular and dynamic. However, it then becomes virtually impossible to ensure the correctness of such applications via classical testing approaches. There is thus a strong need for new correctness verification tools that rely on formal methods.

Formal methods [53] are applied mathematical techniques for the design and analysis of computer hardware and software. In formal methods, model checking technique attracts a lot of attention from researchers due to its effectiveness. Model checking was invented in the early 1980s, and independently developed by Clarke and Emerson [10] and by Queille and Sifakis [40]. After nearly 40 years of development, it has become a popular and efficient method for verifying hardware and software systems. Given a model of a system and a property, model checking automatically checks whether the property holds for that model [6]. It is automatic in the sense that a model checker automatically runs the verification process without requiring any manipulation from users. The principle is usually to explore all possible behaviors of the system model. However, state spaces increase exponentially with the number of concurrent processes. Partial order reduction (POR) and unfolding are two alternative candidate techniques born in

the 90's to mitigate this state space explosion and scale to large applications.

POR comprises a set of exploration techniques sharing the idea that to detect deadlocks it is sufficient to cover each Mazurkiewicz trace, *i.e.* a class of interleavings equivalent by commutation of consecutive independent actions (*i.e.,* two actions are independent if one neither enables nor disables the other one, and their relative ordering has no impact on the final outcome). This state space reduction is obtained by choosing at each explored state, based on the independence of actions, only a subset of actions to explore (*e.g.,* persistent set, ample set, stubborn set) or to avoid (*e.g.,* sleep set).

Dynamic partial order reduction (DPOR) was introduced later to alleviate state space explosion for stateless model checking of software. In this context, while POR would rely on a statically defined and imprecise independence relation, DPOR may be much more efficient by dynamically collecting it at run-time. Nevertheless, redundant explorations may still exist that would lead to an already visited interleaving, and detected by using sleep-sets. This is called sleep-set blocked (SSB) exploration. Avoiding SSBs to get optimality is a challenge. One can rely on unfoldings to get it.

Unfoldings is a concept of concurrency theory providing a compact representation of the behaviors of a model in the form of an event structure aggregating causal dependencies or concurrency between events, and conflicts that indicate choices in the evolution of the program. This representation may be exponentially more compact than an interleaving semantics, while still allowing to verify some properties, such as safety. Each maximal configuration of the unfolding of a program corresponds to a Mazurkiewicz trace representing an equivalence class of executions of the program. UDPOR [27] is an optimal DPOR technique that combines the strengths of PORs and unfoldings and explores the unfolding of a program without redundant explorations of Mazurkiewicz traces.

We aim at verifying asynchronous distributed applications, namely MPI programs, in the absence of a model, but with the possibility to run the code in the SimGrid simulation environment and leverage on the UDPOR technique. Model checking then already faces problems, like determining global states of the system and checking state equality. Applying UDPOR to asynchronous distributed applications brings new difficulties. Determining dependent actions requires to specify the semantics of the programs. However, with many different types of actions as well as their effects, which are very complex, formally specifying them is a very laborious and error-prone task. In addition,

computing the extensions for a configuration may be expensive (roughly speaking, a configuration represents an equivalence class of executions while its extensions denote states that are directly reachable from states in that executions), for example, it is NP-complete in Petri-Nets. So, UDPOR requires clever algorithms to compute unfoldings efficiently, and we cannot directly adapt an existing solution tuned for concurrent programs with only mutexes [34]. For those reasons, the contributions, as well as our solution, are related to the research target.

Contributions

— We first define a compact abstract model of asynchronous distributed programs consisting of few primitives. That abstract model can be used to model asynchronous distributed applications, in particular, it can simulate a large class of MPI programs. To achieve that, we revise and extend an existing abstract model of distributed programs by adding new synchronization primitives. Although most distributed applications are asynchronous, there are still tasks needing to be synchronized (*e.g.* MPI_Win_lock, MPI_Win_unlock calls in RMA programming of MPI programs). By adding the synchronization primitives to the abstract model, a larger set of MPI programs can be encoded, but the abstract model is still compact, with only nine primitives defined. Thus, instead of describing the formal specification of the non-trivial semantic of MPI standard, we only formally describe the abstract model with only a few primitives in the TLA+ specification language. Indeed, our abstract model is easy to specify, and the formal specification of the abstract model is brief. Having a precise specification of the abstract model, we can then reason about independence relations for partial order reduction techniques. In addition, from the specification, we prove that in our model, an enabling action can not be disabled by executing any other actions (persistence property). This property is the key to the efficiency of UDPOR in our model.

— As presented, we aim at leveraging UDPOR while verifying MPI programs in the setting of the SimGrid simulator. Because the abstract model is able to express a large class of MPI programs, and the primitives of our abstract model closely match the ones provided by SimGrid's simulation kernel, we adapt the UDPOR technique on the abstract model rather than the MPI programming model. However, regardless of the calculation model, a challenge arises as to how to effec-

tively compute extensions for a configuration. We proposed steps to adapt UDPOR to model check MPI applications that can be simulated by the SimGrid simulator, focusing on smart algorithms to compute extensions. By exploiting the persistence of the abstract model and insightful observations about configurations and the independence relations between actions, we proposed efficient methods computing such extensions efficiently (in polynomial time).

— Since the UDPOR algorithm is recursive, computing the extensions for a configuration obtained by adding a new event to a particular configuration in the previous step leads to re-computations of many events. We proposed an incremental algorithm for our model, eliminating almost all such re-computations and relying on only the actions that are enabled at the state of the configuration.

— We implemented the quasi-optimal version of UDPOR, a variant of UDPOR, in a prototype adapted to the abstract programming model, *i.e.* with its independence relation. We performed some experiments comparing UDPOR with an exhaustive stateless search on several benchmarks. The prototype is still limited, not connected to the SimGrid environment, and can only be experimented on simple examples. However, with first promising results obtained through the experiments we believe that it is enough to claim that UDPOR can completely be used to model check MPI applications as well as mitigating the state space explosion of such applications.

Regarding the scientific publications of the thesis, two papers have been published during the thesis as follows:

— **The Anh Pham**, Thierry Jéron, Martin Quinson: Verifying MPI Applications with SimGridMC, In Proceedings of the First International Workshop on Software Correctness for HPC applications, CORRECTNESS@SC 2017, Denver, CO, USA, November 2017.

— **The Anh Pham**, Thierry Jéron, Martin Quinson, Unfolding-Based Dynamic Partial Order Reduction of Asynchronous Distributed Programs. In Proceedings of the FORTE conference, Kongens Lyngby, Denmark, June 2019.

CHAPTER 1

INTRODUCTION

1.1 Introduction to distributed programs

From the early days when computers were invented, each computer acted as an independent entity; they could not connect to each other due to the lack of network infrastructure and connection protocols. Therefore computers worked independently of each other. In 1969, the first computer network, ARPANET (Advanced Research Projects Agency Network), was established. It is the first computer network employing packet switching allowing multiple computers to connect simultaneously.

Since then, due to the rapid development of technology, data exchange protocols have been completed, the network infrastructure has been developed and expanded. Computers that are separated from each other by geographical distance can be easily connected as well as exchange data at high speeds. In addition, computational tasks are also getting heavier. They require supercomputers that have a high computational speed or the connection of single computers to run simultaneously for solving a single task or job. For those reasons, computing systems composed of many networked computers are formed to satisfy specific computing tasks, and they are usually considered as *distributed systems (or distributed computer systems)*.

There are various definitions of distributed systems in the literature, and it is difficult to choose the most satisfactory one. However, from an end-user point of view, a distributed system is a collection of autonomous computers communicating through messages passing over communications networks, and it appears to users as a single coherent system [48]. Two properties can be seen through the definition. The first one is that each computer in a distributed system is autonomous, meaning it can operate independently from the other ones. There is neither shared memory nor common physical clock. Each computer can join or leave the system without breaking others. Besides, they are connected to solve a common task and communicate with others in the system by sending messages. The second property is that a distributed system

should appear as *a single coherent system*. Although it includes various computers, the way in which the computers communicate is mostly hidden from users. End users should not be aware that they are working with a system where processes, data, and control are distributed over a computer network [48].

A computer program that runs in a distributed system is called a distributed program. Since running in a distributed memory environment, processes in a distributed program work concurrently and also communicate by message passing not by sharing memory. There are no common variables between them; instead, each process has its own memory and local variables. One of the primary purposes of using distributed programs is to obtain much higher performance than one could get from a single computer. Thus they are also called high-performance computing applications (HPC).

Writing parallel and distributed programs poses notorious correctness challenges. In addition to intrinsic concurrent programming difficulties (*e.g.,* possible race conditions), distributed programming adds some of its own, such as the lack of centralized memory and the lack of a centralized clock. Modern High-Performance Computing applications must take on all these difficulties when attempting to aggregate and fully exploit the computational power of a set of multi-core nodes. In this context, the classical approach to ensure correctness is to rely on rigid communication patterns, so as to avoid complex synchronization scenarios. Unfortunately, these rigid communication patterns scale poorly. The size of modern computer platforms thus mandates applications with communication patterns that are irregular and dynamic. However, it then becomes virtually impossible to ensure the correctness of such applications via classical testing approaches. There is thus a strong need for new correctness verification tools that rely on formal methods. The next section introduces formal methods.

1.2 Formal methods

Formal methods [53] are applied mathematical techniques for the design and analysis of computer hardware and software. Software is ubiquitous in most areas of modern life, and software bugs can be costly. For example, the Y2K bug might be the most expensive bug since the development of informatics. It is estimated that around 500 million dollars had to be spent to fix this bug. Software testing is the most common method for debugging and verifying software systems. It is considered to be easy for users, but it can not prove that absolutely no unwanted behaviors exist in the system

since users do not know how to create test-cases that trigger such bad behaviors, and exhaustive testing is not feasible for a complex software system. Formal methods can be used independently or as a complement to the testing technique in order to ensure the correct behavior of software systems.

In the software development life cycle, formal methods can be applied at several steps. It can also be used to model and analyze existing software systems. By using formal methods in the design phase, one can early identify faults, before implementing the software [21]. Two basic levels of formal methods are usually mentioned: specification and verification.

During the design process of the software life cycle, formal specification is the use of a formal language that has precise syntax, vocabulary, and formal semantics to describe the requirements that the system must achieve. The functionality and specific behaviors (*e.g.,* safety properties) that are required from the system are explicitly defined in specifications [53]. Obviously, a formal specification language (*e.g.,* TLA+, Z notation, Lotos, CSP) cannot be a natural language because the semantics of a natural language is ambiguous and its syntax is not well-defined. So, a formal specification language must be based on mathematical concepts to avoid ambiguity. A system is considered correctly implemented if it meets all the requirements in the system specification. Depending on what kind of system and what level of specifications we want to document, a proper formal specification language should be used. For example, the *Z notation*, a formal specification language, is usually used in Model-based specification approach, can be used to specify non-functional properties of systems, such as usability, performance, size, and reliability, but it does not support concurrency [52] that is supported by TLA+.

Formal specification brings several benefits. For instance, it not only helps software developers have an insight understanding of user requirements but also avoids misinterpretation of the requirements. Besides, by using the mathematical concepts, it can be studied and analyzed by using mathematical methods to detect errors in system requirements at an early stage. Obviously, modifying an already implemented system is usually more costly than correcting errors in the system requirements before deployment.

Besides the benefits of using formal specifications, applying it in practice has some obstacles. Because of using mathematical concepts, formal specifications can be difficult to read and understand for software developers. So, it mandates training in formal

specification techniques that most software engineers lack. In addition, specification languages do not always adequately describe requirements of complex systems (*e.g., Z notation and B* do not support specifying concurrency). However, these obstacles do not preclude the benefits that specification provides, so it is increasingly applied in software engineering.

Formal verification is another level of formal methods. Formal verification [47] is related to the process of verifying the correctness of a system with respect to a particular formal specification by using mathematical techniques. Two main techniques usually mentioned in formal verification are theorem proving, and model checking. Theorem proving is a deductive method, consisting in checking if a software system satisfies its requirements (specification) by mathematical reasoning. Theorem proving is considered a laborious process for users since it requires expert knowledge in mathematics and deep understanding of the system to express the system and requirements as formulas in formal logic that can be understood by the target theorem prover. Therefore, in practice, it is only used effectively by experts in this area and hard to be exploited widely by software developers. Whereas model checking becomes an efficient and easy to use method for users if state spaces are finite or finite abstractions can be exhibited. By exploring all behaviors of a system, it can answer whether the system satisfied a particular property (requirement). Besides, it is entirely automatic technique, and the verification process is performed automatically if the model and specification are provided. However, writing the model and the specification as well as interpreting results requires expertise from users. The following section gives more details of the model checking method.

1.3 Model checking

Model checking technique was invented in the early 1980s, and independently developed by Clarke and Emerson [10] and by Queille and Sifakis [40]. After nearly 40 years of development, it has become a popular and effective method for verifying hardware and software systems. Given a model of a system and a property, model checking automatically checks whether the property holds for that model [6]. It is automatic in the sense that a model checker automatically runs the verification process without requiring any manipulation from users. The main principle of model checking is to traverse all the reachable states and transitions of the system to check if the given property is sat-

isfied. Since, using an exhaustive search, model checking is complete on the model, produces reliable and sound outputs, all the states that violate the property can be found. If the property is found not to hold in a particular execution, the model checker can give a counterexample denoting an execution of the system model leading to the bugs.

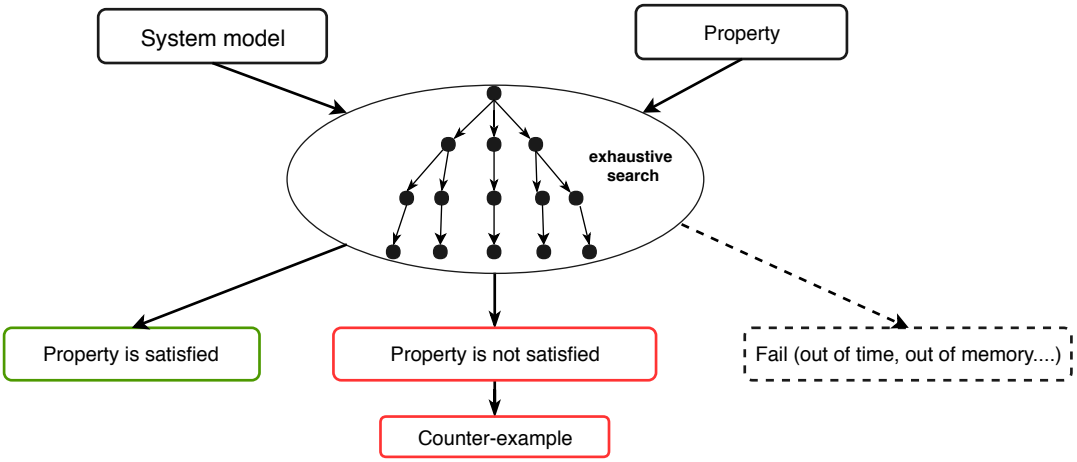


Figure 1.1 – Model checking scheme.

1.3.1 Process of model checking

Basically, there are three different states in the process of verifying a model with model checking:

— Modeling: Modeling is the process in which a software system (here we are only concerned with software) is described in a precise and compact way. Usually, some high-level and abstract modeling language (*e.g.,* Promela, SysML, Lotos) is used. The output of this process is the model of the software system that can be understood by the target model checker (*i.e.,* the model checker that will be used to check the model). The model can be made manually or automatically extracted from the code of the software. Wherever the model comes from, it should be compact as much as possible. Besides, critical properties that are relevant to the verification process should be preserved. For example, deadlocks must be retained in the model if one wants to verify deadlocks in the system. For a large, complex system, the modeling process is hard, requiring experience and a deep understanding of the system. So, ensuring that the model accurately reflects the

operations of the system is challenging for users of model checking. They must answer the question: is the model correct. There is no formal way to efficiently prove if the model correctly reflects the behavior of the system. So, one can rely on simulation to improve the quality of the model before running the verification process.

— Specifying the properties: Property specification languages that can be understood by the model checker are used in this phase to describe the properties. In general, a property is a requirement in the specification of the system. Temporal logic (*e.g.,* Linear Temporal Logic, Computation Tree Logic) is often used to describe properties. The soundness of the specification of the properties should be ensured, meaning that the specification must describe precisely the properties that one wants to verify.

— Verifying: The model checker is run to check if the specified property is satisfied or not. Basically, there are three possible outputs. Firstly, the property is satisfied. Secondly, the property is not satisfied. In this case, the model checker produces a counterexample showing an execution of the system model that leads to an error state in which the property is violated. However, if the property is violated, it does not mean that there is a corresponding error in the system. The model may be incorrect, or it does not reflect accurately what the system performs. So, the model should be improved, removing flaws, and precisely describing what the system does. Another reason is that there is an error in formalizing the property; the property is not exactly formalized. One must restart the verifying stage with a correct model and an accurate specification property. The last output is a failure (*e.g.,* out of time, out of memory) since the number of states is too large to be explored or to be stored in the computer memory. In this case, the model should be reduced in some way.

As presented above, in traditional model checking, the input of a model checker is a model and a property. After the model is verified, then the target system is built. However, in software model checking the situation is different because one wants to verify properties of an already existing code. Software model checking has two broad variants: the first approach follows the manner of the traditional model checking, verifying a model that is usually extracted from the source code of the program by using static analysis; the second approach systematically explores the state space of the program. So, in the latter approach, the model checker can take the real code of a software

program [14], and Dynamic partial order reduction is the most prominent method for exploring as well as reducing the state space of concurrent programs. It is the primary technique discussed in the thesis.

1.3.2 Linear Temporal Logic

A natural language can be used to specify requirements, properties of systems. Unfortunately, natural languages are often ambiguous, and one phrase may have multiple meanings. So they are not acceptable for system specification which requires high accuracy and consistency. Therefore, there is a need for using a formal language which is unambiguous and concise. Propositional logic can be chosen to specify systems because of its precision and conciseness. However, it is not able to express how a system changes over time. An extension of traditional propositional logic, Linear Temporal Logic, can specify properties exhibiting the behavior of systems over time [6]. By using LTL, a wide range of interesting properties can be specified.

In general, there are three main kinds of properties that are most interesting, namely safety (nothing bad happens), liveness (something good eventually happens), fairness (under certain conditions, something will occur infinitely often) [6]. For example, in mutual exclusion algorithms, there is no situation such that two processes are in their critical sections (safety property), or each process will eventually occupy its critical section (liveness). All processes will occupy the critical section infinitely often (fairness).

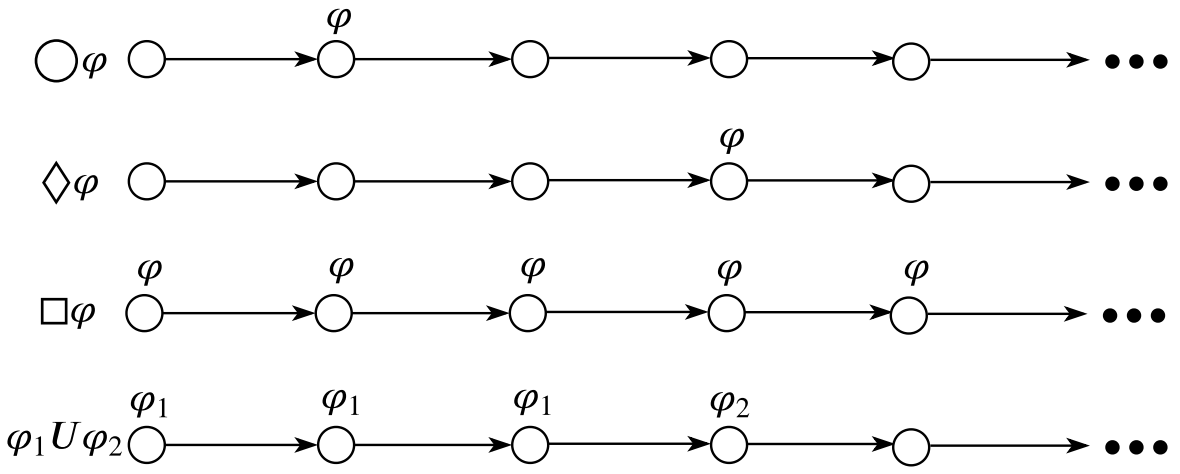


Figure 1.2 – Intuitive semantics of temporal operators.

LTL formulas are built up from a finite set of propositional variables AP, the logical operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and temporal operators $\mathcal{U}$ (until), **X** (next, also denoted by $\bigcirc$), **F** (eventually, also denoted by $\Diamond$), **G** (globally, also denoted by $\Box$). LTL formulas are defined inductively.

Definition 1 *Every atomic proposition p is an LTL formula. If φ_1 and φ_2 are LTL formulas then: $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\bigcirc\varphi_1$, $\Diamond\varphi_1$, $\Box\varphi_1$, $\varphi_1 \mathcal{U} \varphi_2$ are also LTL formulas*

Intuitively, $\bigcirc\varphi$ denotes that φ holds in the next step after the current one while $\Box\varphi$ means that φ holds on all steps. $\Diamond\varphi$ states that eventually φ holds. Finally, $\varphi_1 \mathcal{U} \varphi_2$ expresses that φ_1 must be true and remains true until when φ_2 becomes true. Figures 1.2 sketches the intuitive meaning of temporal operators in LTL.

1.3.3 State space explosion

Model checking is a complete method since it exploits an exhaustive search discovering all states of a system. However, with an extensive system, the number of states can be enormous. For a concurrent system, the number of states can grow exponentially with respect to the number of components in the system (*e.g.,* processes). This dramatic growth is known as the state space explosion problem. The state space explosion problem is the main obstacle to avoiding model checking from being used in practice. Unfortunately, the state space explosion problem is a regular phenomenon in model checking of large scale systems. Several techniques have been proposed for more than 30 years in order to combat the state space explosion problem. The following section reviews some common methods.

— Partial Order Reduction (POR): POR is considered one of the most efficient techniques for alleviating the state space explosion of concurrent programs. POR tries to reduce the state space of a system by exploiting the commutativity between independent actions. The effect of concurrent actions may not depend on the order in which they are executed. So, instead of exploring all orders, discovering only one arbitrary order may be sound to verify safety properties. POR leverages on that observation, exploring only a subset of enabled actions at every visited state based on independence. There are several techniques to compute such subsets. Valmari proposed stubborn sets [49] while ample sets [37] were developed by

Peled. The last method, persistent sets was invented by Godefroid [17], and it is probably most mentioned in the literature.

— Bounded model checking [7]: The principle of bounded model checking is quite straightforward. It only traverses the model (a finite-state transition system) for some fixed value k of steps and checks if a given property is satisfied within this bound. All paths of length k of the model and the negation of the property are encoded by a propositional logic formula and then this formula is passed to a SAT solver (SAT-based Bounded Model Checking). If the formula is unsatisfiable, the property holds. Otherwise, the property is not satisfied. One can start with a small value of k, and then increase k until finding a counterexample or up to the path diameter (*i.e.,* the longest loop-free path between any two states). When k is the path diameter, then the state space of the model is completely discovered. An evolution of SAT-based Bounded Model Checking is SMT-based Bounded Model Checking [4]. The SAT solver is replaced by an SMT solver [1], and the formula is often more compact than those obtained with SAT-based Bounded Model Checking.

— Symbolic Model Checking [29]: This method aims at compressing the state space by using compact symbolic representations. Usually, states and transitions are encoded by Boolean formulas represented by Binary Decision Diagrams. So, symbolic model checking works with Boolean functions rather than explicit states and transitions. In practice, the memory requirements for presenting Boolean functions are usually smaller than for storing explicit states and transitions.

— Abstraction [12]: The idea is to generate a simple finite model (abstract model) sufficient to verify the property from the system model. This can be done by eliminating details irrelevant to the property, a set of states in the system model is represented by an abstract state in the abstract model, and then the model checking procedures are applied on the abstract model rather than an actual model. For example, counterexample-guided abstraction refinement called CEGAR-based model checking [11] is a typical technique in this group. In this technique, one first builds the initial abstract model from an original model (or concrete program). After that, the abstract model is model-checked. If no bug is found, then the original model is safe. If the abstract model admits a counterexample, there are two

1. SMT stands for satisfiability modulo theories.

cases: (i) if the counterexample is present in the original model, then there is a bug; (ii) otherwise, one must refine the abstraction to eliminate the behavior that caused the erroneous counterexample, and then model-check again the new abstract model.

1.3.4 Stateless model checking and stateful model checking

As presented model checking employs an exhaustive search traversing all states of the model of the system that are reachable from the initial state. However, there may be some states that are revisited by the search. To avoid re-exploring the same state, one can record all the already visited states and check if a certain state is already visited, in this case, we call it stateful model checking.

Algorithm 1 illustrates a naive search algorithm for stateful model checking [16]. The principle is trivial. The search will explore all successors of any states that it encounters. The set $setStates$ includes all states whose successors must be explored. The set $visitedStates$ consists of all already visited states. Starting from s_0, the search recursively explores all successors of every state s in $setStates$ by executing all enabled actions at s. A state can only be explored if it is not in $visitedStates$.

Stateful model checking can avoid revisiting states; however, maintaining visited states is costly in terms of memory usage. Besides, in software model checking, checking whether two states of a program are equal is not trivial. Even if we assume that each state can be represented by an identifier, computing unique identifiers for states is hard [16], since a state relates to many factors (*e.g.,* data, variables). The problem is more difficult with distributed programs that contain many processes and communication channels. This problem has attracted attention from researchers. For example, in [20], the authors proposed methods for checking state equality at run-time by using memory introspection rather than relying on the source code analysis.

Due to the challenges to be solved with stateful model checking, stateless model checking [16] was born with a new approach. In the stateless model checking, the search only stores states of the current explored path, and other states are disregarded. Because only part of the exploration is traced, the search is not able to check whether a state has been discovered before, this means that a state can be traversed several times. Partial order reduction techniques which will be discussed in next chapters are used in this context to alleviate that redundancy, and some of those techniques

Algorithm 1: Stateful model checking exploration

```
1  setStates := {s_0}
2  visitedStates is empty.
3  while setStates ≠ ∅
4  do
5  |  s := a state in setStates
6  |  setStates := setStates \ {s}
7  |  if s ∉ visitedStates then
8  |  |  visitedStates := visitedStates ∪ {s}
9  |  |  foreach action t enabled at state s do
10 |  |  |  add the sucessor of s after t to setStates
```

are optimal, exploring only one (complete) execution per each equivalence class of executions if the state space of the model does not contain any cycles. In the case of existing cycles, a pruning (cut-off) technique that requires state equality checking must be used, but the problem becomes more complicated, and it is out of the scope of the thesis.

1.4 Introduction to SimGrid

SimGrid [9] is a simulator of distributed applications. Several user interfaces are proposed, ranging from the classical and realistic MPI formalism, to less realistic simgrid-specific APIs that ease the expression of theoretical distributed algorithms. These user interfaces are built upon a common interface, that is implemented either on top of a performance simulator, or on top of a model checker exploring exhaustively all possible outcomes from a given initial situation.

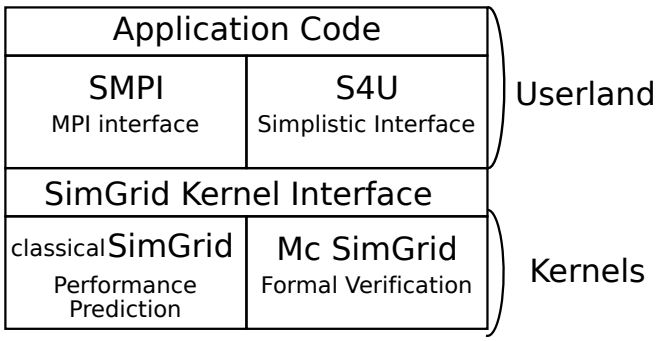


Figure 1.3 – SimGrid architecture.

SimGrid can emulate existing MPI applications using two virtualization mechanisms. First, the distributed application is *folded* into a single image. All MPI processes are compiled as threads within a single OS process. Second, all communication is mediated through the simulator by a specific reimplementation of the MPI standard. This virtualization, detailed in [13], was initially proposed as a way to leverage SimGrid's simulation model so as to predict the performance of unmodified MPI applications. McSimGrid, a stateful model checker in SimGrid, exploits it to formally assess the correctness of the application.

1.4.1 Model checking with SimGrid

Principles When running in model checking mode, SimGrid explores all the possible execution paths starting from an initial application configuration, for a fixed set of inputs. This last restriction makes the tool better adapted to applications that are not data-dependent.

Let's consider a fully distributed model, in which each process executes sequentially and only interacts with others through message exchanges. In this model, execution paths are completely determined by the communications. The only indecision points at which the execution can branch are situations where a given process is waiting for a message that can come from more than one sender. For example, if two processes (*i.e.*, MPI ranks) $rank_1$ and $rank_2$ send a message to process $rank_0$, which accepts any incoming message, then McSimGrid will (1) completely explore the scenario where the message of $rank_1$ arrives first, then (2) rewind the application and (3) completely explore the scenario where the message of $rank_2$ arrives before that of $rank_1$. Similarly, the communication order can induce indecision points when functions such as `MPI_Waitany()` or `MPI_Testsome()` are used.

As presented, the model checking approach suffers from the well known *state space explosion problem*, meaning that the number of execution paths to explore can easily become intractable in practice. However, many of the execution paths are redundant in practice. SimGrid provides two reduction techniques to detect and avoid those redundant paths: Dynamic Partial Ordering Reduction and State Equality.

Dynamic Partial Ordering Reduction was first proposed by Flanagan and Godefoid in [14]. The key idea is that some events are *independent* of each other, meaning that their relative ordering has no impact on the final outcome. For example, local events

occurring on separate hosts are independent. If two events t_1 and t_2 are independent, then two histories only differing by the order of t_1 and t_2 are semantically equivalent. It is then sufficient to explore only one execution path in each such equivalence class. McSimGrid implements a classical DPOR algorithm, at the level of point-to-point communications.

The State Equality reduction technique is based on the simple idea that there is no need to explore twice the outcome of a given state. In abstract models, detecting that the verified system has returned to an already visited state is as simple as computing a hash of all known variables values. In the case of MPI programs, the problem is much more challenging since all information of the program must be introspected. The SimGrid approach, detailed in [20], is to partially reconstruct the semantics of the process memory (global variables, heap, and stack) using a set of tools and libraries that were initially intended for use by debuggers. It is then possible to design a heuristic that only considers relevant (*i.e.*, actually used) bits during state comparison. The heuristic that is implemented in SimGrid is efficient in practice even if it remains fallible because memory semantics cannot always be perfectly reconstructed (*e.g.,* the heap area).

1.4.2 MPI verification in SimGrid

This section highlights several typical use cases enabled by McSimGrid, namely, the discovery of safety and liveness bugs. These use cases are for unmodified MPI applications at small- to mid-range scale.

Safety Properties Safety properties are the simplest type of properties that can be checked with McSimGrid, as they consist of simple assertions. To find a counter-example, a model checker simply searches for a state in which the assertion does not hold. McSimGrid 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.

In [31], the authors showed how McSimGrid can efficiently find bugs in several MPI programs that were specifically written as experimental test cases. The *DPOR* technique was shown to greatly reduce the size of the explored state space during exhaustive verifications. McSimGrid also found a bug in an implementation of the Chord P2P protocol that proved challenging to isolate through testing.

Liveness Properties These properties are often expressed in *Linear Time Logic* (LTL). As such, they combine first-order propositions with quantifiers over time. For example, LTL allows one to express that a given property P holds at some point in the future (noted $\diamond P$). The fact that once you press on the brake pedal the car will slow down after a finite number of events is a classical liveness property. Counter-examples to such properties are infinite execution paths taken by the application without reaching the expected state. Since actual computer systems are finite, such infinite paths must contain cycles. The Model Checker must thus search for cycles in the execution occurring after an eventual triggering event ("the brake pedal is pressed" in our previous example) and before the expected event ("the car slows down").

The classical approach is to build a Büchi automaton that represents the opposite of the considered property (such automata encode and recognize infinite sequences) and to explore in a double-DFS the cross-product of the automaton with the application. Once the exploration reaches an accepting state (*i.e.*, a state satisfying the triggering condition) and until the ending event, the Model Checker actively searches for loops: if it explores again a state that was already explored since the accepting state, then an infinite loop that violates the property has been found.

When considering real applications, a key difficulty is to evaluate whether the application has reached a state that was already explored. To address this difficulty, McSimGrid leverages the features that form the basis for the State Equality reduction technique, presented earlier. McSimGrid was used to verify several applications from the MPICH3 testsuite (consisting of up to 1,300 lines in C or Fortran), exhaustively searching for non-progressive loops (*i.e.*, livelocks) in various scenarios involving 2 to 6 processes. McSimGrid was able to verify these scenarios in less than a day with State Equality reduction enabled, proving the absence of livelocks in these applications. McSimGrid was also able to find a bug in an erroneous implementation of mutual exclusion in which the request of a given host was deliberately never answered. These results are detailed in a research report [20].

1.5 Contributions of the thesis

We are inspired by the fact that HPC applications that are often implemented by using MPI libraries are becoming increasingly popular, but ensuring their accuracy is intractable. Besides, we also have an efficient simulator called SimGrid that can study

the behavior of large-scale distributed computer systems. For example, SimGrid can evaluate heuristics, prototype applications, or assess legacy MPI applications. In addition, in recent years, in research on software model checking, there has been a significant step in mitigating the state space explosion problem of concurrent programs. These reasons motivated us to study how to adapt recent studies to verify distributed applications based on the setting of SimGrid simulator, especially MPI applications. Therefore the contributions of the thesis revolve around these research questions, and are follows:

— The thesis revises and extends an existing abstract model of distributed programs by adding new synchronization primitives. Although most distributed applications are asynchronous, there are still tasks needing to be synchronized. For example, in Remote memory access mode in MPI, to synchronize distributed processes that access the same target window (a shared memory declared by a particular process), one can use MPI synchronization functions (*e.g.,* MPI_Win_lock, MPI_Win_unlock). Thus, there is a need for simulating such kind of tasks. By adding the synchronization primitives to the abstract model, a larger set of MPI programs can be encoded. However, the extended model also entails a re-analysis in order to reason about independence relations for partial order reduction techniques. Therefore, in this thesis, besides extending the abstract model, we specify the abstract model in TLA+, and then study the model by simulations as well as define and prove the independence relation.

— The main target of the thesis is efficiently adapting UDPOR (an optimal partial order reduction technique) to verify MPI programs. In the UDPOR exploration algorithm, computing extensions for a configuration can be costly (roughly speaking, a configuration represents an equivalence class of executions while its extensions denote states that are directly reachable from states in that executions). A brute-force approach for computing such values would require to iterate over all subsets of events in the configuration, resulting in an algorithm running in exponential time. By exploiting the properties of the abstract model and insightful observations about configurations and the independence relations of the model, we propose an algorithm that can avoid iterating over all the subsets and only considers interesting ones that are identified by very few events. Finally, the algorithm that computes the extensions runs in polynomial time.

— Since the UDPOR algorithm is recursive, computing the extensions for a configu-

ration obtained by adding a new event to a particular configuration in the previous step leads to re-computations of many events. We proposed an incremental algorithm for our model, eliminating almost all such re-computations and relying on only the actions that are enabled at the state of the configuration.

— We implemented a prototype encoding UDPOR and a variant of UDPOR (UDPOR with k-partial alternatives) exploiting our proposed methods and model. A prototype has been experimented on some benchmarks, gaining promising first results.

1.6 Organization of the manuscript

The rest of this manuscript is organized as follows.

Chapter 2 is mainly dedicated to present some notable studies on (Dynamic) POR. Besides those works about POR techniques, we also review some model checkers used to verify MPI programs.

Chapter 3 recalls notions of interleaving and concurrency semantics, and how a transition system can be unfolded into an event structure with respect to an independence relation. The UDPOR algorithm is also detailed in this chapter.

We use chapter 4 to describe our abstract model of distributed programs. Later, the formal specification of the abstract model as well as independence theorems are presented. The chapter is closed by demonstrations of how to encode some basic functions of the MPI standard as well as simple MPI programs.

The adaptation of UDPOR to our programming model and how to make UDPOR efficient are demonstrated in chapter 5. The adaptation is evaluated on several experiments.

Finally, the last chapter concludes the thesis by summarizing our contributions and discussing perspectives.

CHAPTER 2

STATE OF THE ART

This chapter will discuss some main critical features related to the evolution of partial order reduction. Besides briefly presenting some recent studies on POR, the chapter also presents some works applying DPOR to verify MPI programs. The chapter is closed by reviews of some model checkers used for checking MPI programs.

2.1 Partial order reduction

POR techniques have been developed since the early 90s of the last century, and it comprises a set of exploration techniques (see *e.g.* [17]), sharing the idea that, to detect deadlocks (and, by extension, for checking safety properties) it is sufficient to cover each *Mazurkiewicz trace*, *i.e.* a class of interleavings equivalent by commutation of consecutive *independent* actions. This state space reduction is obtained by choosing at each state, based on the independence of actions, only a subset of actions to explore (ample, stubborn or persistent sets, depending on the method), or to avoid (sleep set). Although PORs consist of several different variants, perhaps persistent set and sleep set techniques are most often mentioned in studies of PORs.

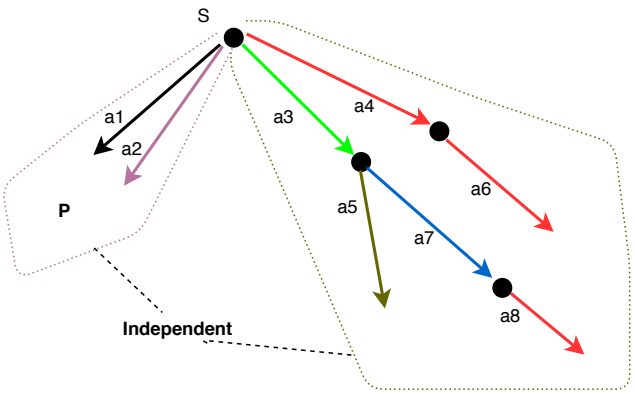


Figure 2.1 – Persistent set.

Figure 2.1 visualizes the persistent set of state s. A subset P of enabled actions at state s is a persistent set of s if all actions in P (*i.e.,* a_1 and $a2$) are independent with all actions reachable from s (and outside P) without executing any action in P.

Early POR relies on a statically defined and imprecise independence relation. For example, given two write actions $write(arr[i])$ and $write(arr[j])$ (they write a value to an element of the array arr), a static analyzer always considers that they are dependent because it is not able to detect whether $i = j$ or not. However, if $i \neq j$ the actions are independent.

Dynamic partial order reduction (DPOR) [14] was later introduced to combat state space explosion for stateless model checking of software. In this context, while POR relies on a statically defined and imprecise independence relation, DPOR may be much more efficient by dynamically collecting it at run-time. While executing a program, DPOR can check exactly which threads/processes access which memory locations. Figure 2.2 illustrates the main principle of the DPOR technique. While executing the write action of thread *red (*(0x2AAA) :=7)*, DPOR detects that it accesses the same memory location as the write action (*(0x2AAA) :=5) of thread *black* accessed before. Thus, the actions are dependent, and thread *red* is added to the persistent set of state s. When backtracking, a transition of the thread *red* that is enabled at state s should be explored from s. Obviously, in this example, DPOR precisely computes dependence relation, and in practice, persistent sets computed by DPOR are often smaller than those computed by static analyzers.

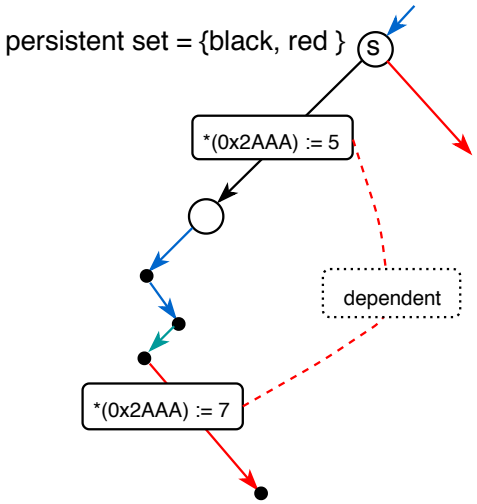
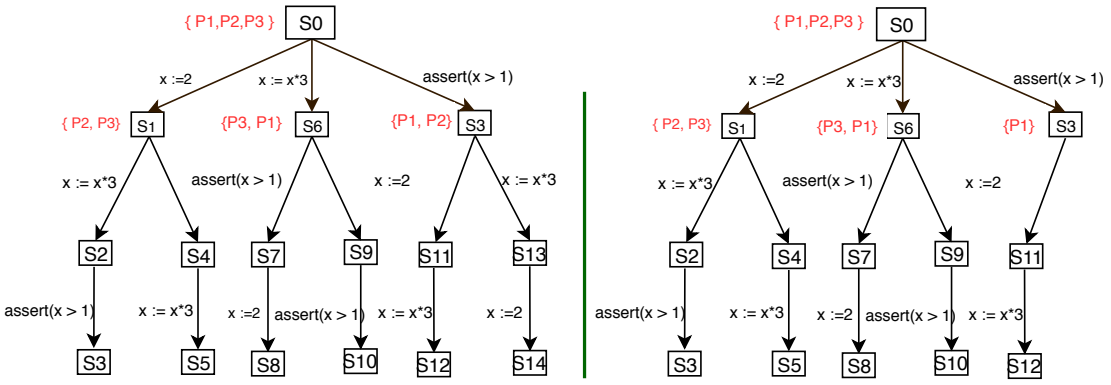


Figure 2.2 – Dynamic partial order reduction.

2.2 Recent studies on DPOR

Although often producing better reductions than those created by POR, redundant explorations, named *sleep-set blocked* (SSB, *i.e.* the search must backtrack since it detects that continuing exploring the current trace will lead to an already visited state), may still exist in DPOR. Thus, DPOR is unable to always explore exactly one interleaving per Mazurkiewicz trace. In the last few years, two research directions were investigated to improve DPOR. The first one tries to refine the independence relation: the more precise, the less Mazurkiewicz traces exist, thus the more efficient could be DPOR. The second one focuses on minimizing the number of explored executions per each Mazurkiewicz trace. Let's take a look at some representatives in these directions.

DPOR with *observers* In [5] independence is built lazily, conditionally to future actions called *observers*. The principle is straightforward. In the original DPOR, if two write actions concern the same variable, then they are always considered dependent. However, in practice, changing the execution order of two dependent actions, that concern the same variable, does not change the resulting executions if such a variable is not read later. So, DPOR with *observers* only considers that two write actions concerning the same memory location are dependent if at least one of them has a followed read action concerning the same memory location.



6 traces are explored by DPOR

5 traces are explored by DPOR with Observers

Figure 2.3 – A comparison between DPOR and DPOR with observers.

For example, a program spawns three concurrent processes $P1, P2, P3$ executing

$x := 2$, $x := x * 3$ and $assert(x > 1)$, respectively. Figure 2.3 depicts the explorations of DPOR (left) and DPOR with *observers* (right). While the first method explores six traces, the later one traverses only five traces. Let's zoom in on state $S3$, because $x := 2$ and $x := x * 3$ concern the same variable, $P2$ is added to the persistent set of $S3$, and then action $x := x * 3$ is fired from $S3$. However, DPOR with *observers* does not consider that $x := 2$ and $x := x * 3$ are dependent since there is no read action executed after them, then $P2$ is not performed from $S3$.

The idea can be lifted to message passing programs in the case where there are some send actions concerning the same mailbox (see [5]) to get more reductions.

Constrained DPOR In [3], Elvira et al. propose an extension of DPOR called constrained DPOR. It considers conditional independence relations [18] where commutations are specified by constraints. Using conditional independence can avoid more unnecessary exploration. For example, two processes $P1, P2$ of a concurrent program fire two atomic actions $if(a > 1)x := 1$ and $x := x * 2$. DPOR with unconditional dependence considers that the two actions are dependent even if $a \leq 1$ (an independence constraint) without considering the constraint $if(a > 1)$. Constrained DPOR takes into account constraints when computing dependence of actions, and uses an SMT solver to synthesize them. Experiments in [3] show that in almost all the benchmarks, constrained DPOR outperforms the standard DPOR on all the following aspects: number of explored traces, run time and number of visited states.

Context-Sensitive DPOR Context-Sensitive DPOR [2] is another extension of DPOR and exploits context-sensitive independence to extend the performance of sleep sets. Standard DPORs are based on context-insensitive independence and require two actions to be independent in all contexts, while Context-Sensitive DPOR considers two actions might be independent in the particular explored context. From the implementation point of view, Context-Sensitive DPOR is equipped with a state equality detection, that checks if the execution order of two dependent actions does not change the overall result, to cut more unnecessary explorations.

Let's look at the example shown in Figure 2.4. A concurrent program contains three processes $p1, p2$ and $p3$ executing $x := 3$, $x := 3$ and $y := x$, respectively. Similar to DPOR, Context-Sensitive DPOR firstly explores an execution, suppose it is $p1p2p3$ (here we consider actions named by process names). Since $p1$ and $p2$ are depen-

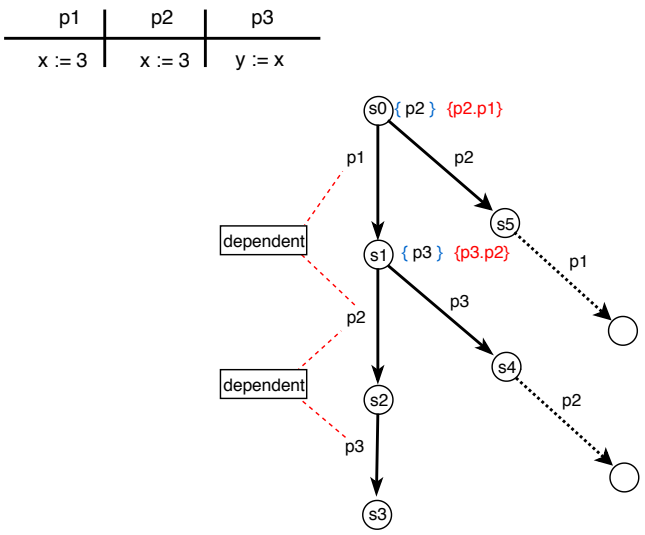


Figure 2.4 – Context-sensitive DPOR

dent, $p2$ is added to the backtracking set of state $s0$. However, $p1.p2$ and $p2.p1$ [1] lead to the same state, Context-Sensitive DPOR annotates at $s0$ that $p1$ should not be executed after $p2$, meaning that $p1$ should not be fired from $s5$. Similarly, it annotates that, after executing $p3$ from $s1$, $p2$ should not be fired. Obviously, more reductions are obtained by using Context-Sensitive DPOR. However, it also requires implementing a state equality detection that may be a challenge for some applications, such as distributed applications.

Minimizing the number of explored executions This direction proposes alternatives to persistent sets in order to minimize the number of explored interleavings. Remember that optimality is obtained when exactly one interleaving per Mazurkiewicz trace is built. In [1] authors propose *source sets* that are often smaller than persistent sets and operates over interleaving semantics. Although DPOR with *source sets* outperforms the traditional DPOR, it may still lead to sleep-set blocked explorations. To explore exactly one interleaving per Mazurkiewicz trace, and never encounter sleep-set blocking, the authors combine *source sets* with a novel mechanism, called *wakeup trees*, leading to an optimal DPOR algorithm.

In [27] the authors propose *unfolding-based DPOR* (UDPOR), an optimal DPOR method combining the strengths of PORs and unfoldings with the notion of *alterna-*

1. Both $p1.p2$ and $p2.p1$ must be executed.

tives and operating over concurrent semantics. The approach is generalized in [34] with a notion of k*-partial alternative* allowing to balance between optimal DPOR and sometimes more efficient sub-optimal DPOR. UDPOR will be detailed in section 3.2.

Distributed DPOR Besides the above mentioned methods that focus on reducing the number of explored traces, some other techniques improve the performance of DPOR by proposing new schemes of implementations. For example, in [55], the authors distributed DPOR on a computer cluster. Each node in the cluster is responsible for exploring some backtracking points of some backtracking set (persistent set). However, performing DPOR in a distributed environment can lead to a situation where two nodes explore the same backtracking point in the persistent set of the same visited state. Two avoid this situation, the persistent set of a state is only distributed when it has been completely constructed.

DPOR in the verification of MPI programs Some approaches already try to use DPOR techniques for the verification of asynchronous distributed applications, such as MPI programs (Message Passing Interface). In the absence of model, determining global states of the system and checking equality [39] are already challenging. In [36], an approach is taken that is tight to MPI. A significant subset of MPI primitives is considered, formally specified in order to define the dependency relation, and then to use the DPOR technique of [14].

In [43], the efficiency DPOR is improved by focusing on particular deadlocks (*i.e. orphaning deadlocks* that happen when there is no matching send for a particular receive in some MPI program execution), but at the price of incompleteness. The work develops an effective heuristic for reducing persistent sets to alleviate exponential schedule explosion caused by non-deterministic receives. The heuristic hinges on some observations; for example, the fact that the number of sends is not equal to the number of receives in a particular program execution can cause a deadlock. Experiments in [43] demonstrated a significant saving in exploring the state space of a program in order to detect deadlocks, however, it fails to detect deadlocks in some particular MPI programs.

Recent research in [8] focuses on detecting deadlocks in MPI programs but in the context where the dependence, causality relation between actions are not clearly defined. Because the semantics of MPI is ambiguously defined, depending on "zero buffering" or "infinite buffering" mode, send operations and collective operations may

or may not block. However, the modes are not decided by users but depends on some factors, such as the availability of the internal storage, the MPI implementation. For example, in the zero buffering mode, an MPI_Send must wait for a matching posted MPI_Receive to return, but it can return without waiting for any matching MPI_Receive operation in the infinite buffering mode. To apply DPOR, the dependence and causality relations of MPI programs must be rigorously defined. The authors proposed a method in which an MPI program is firstly explored by (Dynamic) Partial order reduction in the infinite buffering mode to get the reduced state space. After that, a proposed "post-processing" algorithm is applied to the reduced state space to detect deadlocks.

2.3 Model checkers for MPI programs

The idea of applying model checking to actual programs originated in the late 90s [16, 32]. It was later applied to many systems and interfaces such as Java [30], multi-threaded programs [33] or distributed programs [54]. In the context of MPI-based parallel programs also, several tools have been proposed [19].

Runtime instrumentation tools such as Marmot [26] or MUST [22] intercept and verify every MPI call. This catches API misusages, such as type mismatch between a send and the corresponding receive. In addition, a dependency graph of the calls (either centralized in Marmot or distributed in MUST) can catch some deadlocks. Unfortunately, such testing tools only explore *some* of the possible execution paths.

Several static tools based on code analysis were proposed. TASS [46] and CIVL [45] rely on symbolic execution and state enumeration techniques to propagate the interval of values taken by the application variables. It requires source code annotations specifying bounds on input variables to reduce the number of false positives. Similarly, PARCOACH [42] detects through static analysis potentially problematic sections of code and adds assertions which are then checked at runtime. This does not require any code annotations but will miss failures that occur on unexplored execution paths. Formal approaches, such as that implemented in this work, are needed for exhaustive coverage.

Hermes [23] implements a hybrid technique combining explicit-state dynamic verification with symbolic analysis to discover deadlocks in MPI programs. The dynamic verification component is responsible for executing interesting traces of programs while the symbolic one encodes a set of interleavings of the observed trace into a formula

and then verifies it to detect communication deadlocks. If there are no property violations, an analysis is performed to obtain another trace that has not already been verified. *Hermes* outperforms some other verification tools (*e.g.,* CIVIL tool) in some benchmarks, however, it also has some limitations, for example, it assumes that the data that is received by a non-deterministic receive and then used in a conditional statement in programs is an integer variable or a *tag*.

One of the first formal tools specifically designed for MPI programs was MPI-Spin [44], an extension to the classical Spin model checker. But it requires the user to manually build an abstracted model of the application. Gauss [35] is an automated model extractor, but it remains limited to small applications because much information that is required to build an efficient and accurate model of the application is only known at runtime.

Many tools use the PMPI instrumentation layer of MPI to observe and steer the application. Nasty-MPI [25] delays the calls to experience pessimistic schedules that often trigger bugs in the application. ISP [51] and DAMPI [50] are formal tools that dynamically explore all the possible execution paths while applying adequate reduction techniques to not re-explore Mazurkiewicz traces when possible.

2.4 Conclusion

As stated, (Dynamic)POR has attracted a lot of attention from the formal methods community. In recent years there have been many notable studies to improve the POR techniques, typically optimizing the number of explored executions and refining independence relations. However, studies seem to be largely focused on shared memory programs, but less focused on message passing ones, typically MPI programs. In order to adapt these studies into message passing applications, it often takes a lot of effort.

Some tools use methods outside of DPOR such as static analysis, symbolic analysis to verify MPI programs. However, they also have certain limitations as stated, and such techniques are out of the scope of this thesis. This thesis focuses mainly on verification of MPI programs by exploiting a state of the art method, namely UDPOR. To the best of our knowledge, this is the first attempt to adapt UDPOR to verify MPI applications.

CHAPTER 3

PRELIMINARIES

This chapter introduces labelled transition systems (LTS) that is a typical mathematical concept in computer science usually used to model behaviors of hardware as well as software systems. Besides introducing labelled transition systems, concurrency semantics, specifically unfolding semantics, will be mentioned in detail. Different aspects of unfolding semantics are reviewed, ranging from the basic notions, dependency relation, and definitions as well as how to build the unfolding of a program from its LTS semantics. The chapter is closed by a detailed presentation of Unfolding-based DPOR.

3.1 Interleaving and concurrent semantics

3.1.1 Labelled transition systems

A labelled transition system is a graph where nodes represent states, and edges express transitions between the states. A state describes the information of the system at a particular time. For instance, a state of a multi-threaded program may indicate the current values of all global variables as well as local variables, and the current value of the program counter while a global state of a distributed program is the union of the states of the individual processes and the states of communication channels. Each edge can be labelled by an action that triggers the transformation of the system from one state to another.

Definition 2 (Labelled transition system) *A* labelled transition system *(LTS) is a tuple* $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ *where* S *is the set of states,* $I \subseteq S$ *is the set of initial states,* Σ *is the alphabet of actions, and* $\rightarrow \subseteq S \times \Sigma \times S$ *is the transition relation.*

Example 1 *Figure 3.1 shows an example of an LTS modelling a simple beverage vending machine that delivers either tea or coffee. The state space is* $S = \{s_0, s_1, s_2, s_3\}$*, and the set of initial states is* $I = \{s_0\}$*. The set of actions of the LTS is* $\Sigma = \{get\_tea,$

$coin, get\_coffee, \tau\}$. *In the figure symbols* τ *denote some internal activity of the machine that may not be of interest. In total, five transitions are defined in the LTS:* $(s_0, coin, s_1)$, (s_1, τ, s_2), (s_1, τ, s_3), (s_2, get_tea, s_0), *and* (s_3, get_coffee, s_0).

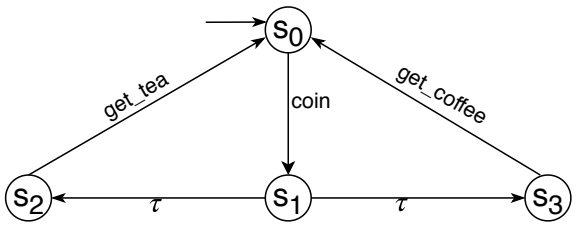


Figure 3.1 – A transition system of a simple beverage vending machine.

For convenience, we often write $s \xrightarrow{a} s'$ instead of (s, a, s'), and if $(s \xrightarrow{a_1} s')$, $(s' \xrightarrow{a_2} s'')$, we can write $(s \xrightarrow{a_1.a_2} s'')$. Intuitively, the behavior of the transition system can be described as follows. The transition system starts from the initial state (s_0). From that state, according to the transition relation, the transition system evolves. If s is the current state and $s \xrightarrow{a} s'$ is a transition, then action a is executed and the transition system evolves from state s into the state s'. There may exist more than one possible transition from some state. In that case, the LTS can arbitrarily select a transition to evolve. The evolution of the LTS stops when encountering a state where there is no outgoing transition.

Consider the example in Figure 3.1, from the initial state s_0, there is only one possible transition $s_0 \xrightarrow{coin} s_1$. So, the transition $s_0 \xrightarrow{coin} s_1$ is selected and taken, then the LTS evolves from s_0 to s_1 after performing a $coin$ action. In s_1, there are two internal steps. Thus, an internal step is selected arbitrarily, either selecting $s_1 \xrightarrow{\tau} s_2$ or $s_1 \xrightarrow{\tau} s_3$. If $s_1 \xrightarrow{\tau} s_2$ is selected and taken the LTS moves from s_1 to s_2, and after that an action get_tea is performed. Otherwise, $s_1 \xrightarrow{\tau} s_3$ is selected, the LTS moves from s_1 to s_3 and then a get_coffe is performed.

We have notions related to labelled transition systems as follows:

Post states: Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a labelled transition system. Given a state s, the set of direct successors of s after performing action a is defined as follows:

$$PostState(s, a) = \{s' \in S \mid s \xrightarrow{a} s'\}$$

$PostState(s)$ includes all states reachable from state s by performing any action.

$$PostState(s) = \bigcup_{a \in \Sigma} PostState(s, a)$$

Similarly, we define Pre states:

Pre states: Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a labelled transition system. Given a state s and an action a, the set of direct predecessors of s is defined as follows:

$$PreState(s, a) = \{s' \in S \mid s' \xrightarrow{a} s\}$$

$PreState(s)$ includes all states that can reach s by performing any action a.

$$PreState(s) = \bigcup_{a \in \Sigma} PreState(s, a)$$

Enabling actions: Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a labelled transition system. Given a state $s \in S$, the set of transitions enabled at state s denoted $enabled(s)$ is defined as follows: $enabled(s) = \{a \in \Sigma \mid \exists s' \in S, s \xrightarrow{a} s'\}$.

Terminal state: A state s in a labelled transition system $\mathcal{T}$ is called terminal (or deadlock) if and only if $enabled(s) = \emptyset$.

Roughly, terminal states of a labelled transition system $\mathcal{T}$ are states without any outgoing transitions. When modeling a program, a terminal state represents the termination or a deadlock of the program. Hence, when $\mathcal{T}$ encounters a terminal state, the real program modeled by $\mathcal{T}$ terminates or encounters a deadlock.

Deterministic Transition System: Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a labelled transition system. $\mathcal{T}$ is said deterministic if:

1. $|I| = 1$, that is, there is only one initial state, and
2. $\forall a \in \Sigma$ and $\forall s \in S$, $|PostState(s, a)| \leq 1$, that is, for all actions and states, performing an action from a given state leads to at most one successor state.

Otherwise, $\mathcal{T}$ is said non-deterministic.

Reachable states: Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a transition system. A state $s \in S$ is called reachable in $\mathcal{T}$ if there exists a sequence of states and actions $s_0 a_0 s_1 a_1 s_2 a_2 ... s_n$ such that s_0 is an initial state, $s_i \xrightarrow{a_i} s_{i+1}$, $\forall i \geq 0$, and $s_n = s$.

Executions: An execution (or run) describes a possible behavior of the transition system, it can be formally defined as follows:

Definition 3 (Executions) *Let $\mathcal{T} = \langle S, I, \Sigma, \rightarrow \rangle$ be a transition system. An execution of $\mathcal{T}$ can be finite or infinite:*

- *A finite execution of $\mathcal{T}$ is a finite sequence of states and actions: $s_0 a_0 s_1 a_1 ... s_n$ s.t. s_0 is an initial state, $s_i \xrightarrow{a_i} s_{i+1}$, $\forall i \geq 0$, and if s_n is a terminal state, we say that $\mathcal{T}$ is a complete execution.*
- *An infinite execution of $\mathcal{T}$ is an infinite sequence of states and actions: $s_0 a_0 s_1 a_1 s_2 a_2 ...$ such that s_0 is an initial state, $s_i \xrightarrow{a_i} s_{i+1}$, $\forall i \geq 0$.*

In the example shown in Figure 3.1, "$s_0\ coin\ s_1\ \tau\ s_2\ get\_tea\ s_0 ...$" is an execution of the LTS while "$s_1\ \tau\ s_2\ get\_tea\ s_0\ coin\ s_1$" is not an execution because s_1 is not the initial state.

3.1.2 Independent actions

Independence is a key notion in both POR techniques and unfoldings, linked to the possibility to commute actions:

Definition 4 (Commutation and independence) *Two actions a_1, a_2 of an LTS $\mathcal{T} = \langle S, \{s_0\}, \Sigma, \rightarrow \rangle$* commute *in a state s if they satisfy the two conditions:*

- *executing one action does not enable nor disable the other one:*

$$a_1 \in enabled(s) \wedge s \xrightarrow{a_1} s' \implies (a_2 \in enabled(s) \Leftrightarrow a_2 \in enabled(s')) \tag{3.1}$$

- *their execution order does not change the overall result:*

$$a_1, a_2 \in enabled(s) \implies (s \xrightarrow{a_1 \cdot a_2} s' \wedge s \xrightarrow{a_2 \cdot a_1} s'' \implies s' = s'') \tag{3.2}$$

A relation $I \subseteq \Sigma \times \Sigma$ *is a* valid independence relation *if it under-approximates commutation, i.e.* $\forall a_1, a_2, I(a_1, a_2)$ *implies that* a_1 *and* a_2 *commute in all states. Conversely* a_1 *and* a_2 *are* dependent *and we note* $D(a_1, a_2)$ *when* $\neg(I(a_1, a_2))$.

A *Mazurkiewicz trace* is an equivalence class of complete executions (or interleavings) of an LTS $\mathcal{T}$ obtained by commuting adjacent independent actions. By the second item of Definition 4, all these interleavings reach a unique state. The principle of all POR approaches is precisely to reduce the state space exploration while covering at least one execution per Mazurkiewicz trace. If a deadlock exists, a Mazurkiewicz trace leads to it and will be discovered. More generally, safety-property violations in any acyclic state space can be detected by POR algorithms.

3.1.3 Event structures

Besides labelled transition systems, an event structure is another mathematical concept that is usually used to model computer programs. An event structure consists of a set of events that are bound by one of three relations: concurrency, conflict, and causality. While two conflicting events cannot be performed together, two events that are causally related have a dependency, one of the two events can only happen after the other. A classical model of true concurrency is prime event structures.

Definition 5 (Prime event structure) *Given an alphabet of* actions Σ*, a* Σ-prime event structure *(Σ-PES) is a tuple* $\mathcal{E} = \langle E, <, \#, \lambda \rangle$ *where* E *is a set of* events*,* $<$ *is a partial order relation on* E*, called the* causality relation*,* $\lambda : E \to \Sigma$ *is a function labelling each event* e *with an action* $\lambda(e)$*,* $\#$ *is an irreflexive and symmetric relation called the* conflict relation *such that the set of causal predecessors or* history *of any event* e*,* $\lceil e \rceil = \{e' \in E : e' < e\}$ *is finite, and conflicts are inherited by causality:* $\forall e, e', e'' \in E, e\#e' \wedge e' < e'' \implies e\#e''$.

Intuitively, $e < e'$ means that e must happen before e', and $e\#e'$ that those two events cannot belong to the same execution. Two distinct events that are neither causally ordered nor in conflict are said *concurrent*. An event e can be characterized by a pair $\langle \lambda(e), H \rangle$ where $\lambda(e)$ is its action, and $H = \lceil e \rceil$ its history.

Example 2 *Figure 3.2 illustrates a PES modeling a concurrent program. In the PES each box represents an event with a number that is the event identifier while arrows*

illustrate the causality relation, and dotted lines denote conflict relation[1]*. The set of events is* $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$*, and regarding the labeling function* λ*, for instance* $\lambda(e_1) = \langle 0, x := a \rangle$*,* $\lambda(e_8) = \langle 1, x := a' \rangle$*. Causality and conflict relations are such that* $e_1 < e_6$*,* $e_3 < e_4$*,* $e_3 \# e_8$ *for example.*

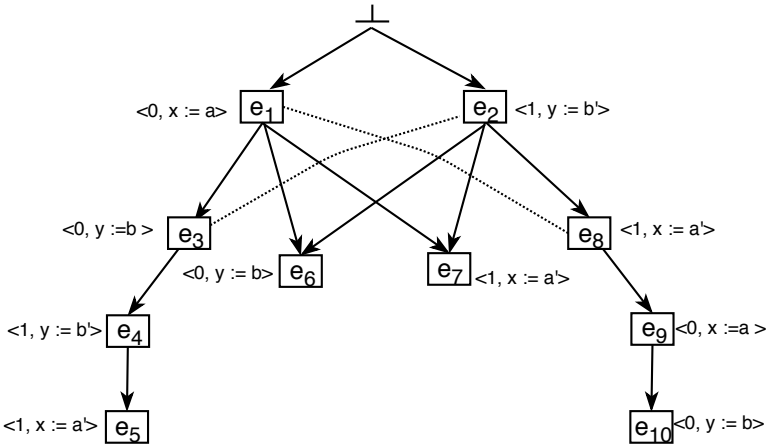


Figure 3.2 – A prime event structure

Configuration and maximal events:

A *configuration* is a set of events $C \subseteq E$ that is both *causally closed* ($e \in C \implies \lceil e \rceil \subseteq C$) and *conflict free* ($e, e' \in C \implies \neg(e \# e')$). For example $\{e_1, e_2, e_6\}$ is a configuration; $\{e_1, e_4, e_5\}$ is not a configuration since it is not causality closed; $\{e_1, e_2, e_3\}$ is not a configuration because $e_2 \# e_3$.

Maximal events of a configuration C denoted by $maxEvents(C)$ is the set containing all events $e \in C$ such that e does not belong to the history of any other event e' in C. Formally $maxEvents(C) = \{e \in C : \nexists e' \in C, e < e'\}$. For instance, $maxEvents(\{e_1, e_2, e_6, e_7\}) = \{e_6, e_7\}$.

Local configuration: *The local configuration* of event e denoted by $[e]$ is the configuration containing e as well as all events in the history of e. Formally $[e] := \lceil e \rceil \cup \{e\}$. For example $[e_4] = \{e_4, e_3, e_1\}$.

A configuration C is characterized by its causally maximal events since it is exactly the union of local configurations of these events: $C = \bigcup_{e \in maxEvents(C)} [e]$; conversely a

1. Here only initial conflicts are drawn, and note that conflicts are inherited by causality.

conflict free set K of incomparable events for $<$ defines a configuration $config(K)$ and $C = config(maxEvents(C))$.

A configuration C, together with the causal and independence relations define a *Mazurkiewicz trace*: all interleavings are obtained by causally ordering all events in the configuration C but commuting concurrent ones. We use $conf(\mathcal{E})$ for the *set of configurations* of $\mathcal{E}$.

Maximal configuration: A configuration C is said maximal if C cannot be extended to a new configuration by adding some event $e \in E$. Formally, C is maximal if $\nexists e' \in E : e' \notin C, C \cup \{e'\}$ is a configuration. For example $\{e_1, e_2, e_6, e_7\}$ is a maximal configuration, but $\{e_1, e_2, e_7\}$ is not maximal.

Immediate conflict: Two events e and e' are in immediate conflict, denoted by $e \#^i e'$, iff $e \# e'$ and both $\lceil e \rceil \cup [e']$ and $\lceil e' \rceil \cup [e]$ are configurations. For instance, $e_1 \#^i e_8$, $e_2 \#^i e_3$.

We also have new definitions based on the immediate conflict. While $\#^i(e)$ contains all events in E in immediate conflict with e, $\#^i_U(e)$ consists of the same set of events but restricted to U, formally $\#^i(e) := \{e' \in E : e \#^i e'\}$, $\#^i_U(e) := \#^i(e) \cap U$.

Extensions: The set of *extensions* of C is $ex(C) = \{e \in E \setminus C : \lceil e \rceil \subseteq C\}$, *i.e.* the set of events not in C but whose causal predecessors are all in C. For example, $ex(\{e_1, e_2\}) = \{e_3, e_6, e_7, e_8\}$, $ex(\{e_2\}) = \{e_8\}$.

When appending an extension to C, only resulting conflict-free sets of events are indeed configurations. These extensions constitute the set of *enabled events* $en(C) = \{e \in ex(C) : \nexists e' \in C, e \# e'\}$ while the other ones are *conflicting extensions* $cex(C) := ex(C) \setminus en(C)$. For example $en\{e_1, e_2\} = \{e_6, e_7\}$ while $cex(\{e_1, e_2\}) = ex(e_1, e_2) \setminus en(\{e_1, e_2\}) = \{e_3, e_8\}$.

Parametric Unfolding Semantics. Given an LTS $\mathcal{T}$ and an independence relation I, one can build a prime event structure $\mathcal{E}$ such that each linearization of a maximal configuration (*i.e.,* a total ordering of all the events in the configuration that respects the causality relation) represents an execution in $\mathcal{T}$, and conversely, to each Mazurkiewicz trace in $\mathcal{T}$ corresponds a configuration in $\mathcal{E}$ [34].

Definition 6 (Unfolding) *The* unfolding *of an LTS $\mathcal{T}$ under an independence relation I is the Σ-PES $\mathcal{E} = \langle E, <, \#, \lambda, \rangle$ incrementally constructed from the initial Σ-PES $\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ by the following rules until no new event can be created:*

- *for any configuration $C \in conf(E)$, any action $a \in enabled(state(C))$, if for any $e' \in maxEvents(C)$, $\neg I(a, \lambda(e'))$, add a new event $e = \langle a, C \rangle$ to E;*
- *for any such new event $e = \langle a, C \rangle$, update $<$, $\#$ and λ as follows: $\lambda(e) := a$ and for every $e' \in E \setminus \{e\}$, consider three cases:*
  (i) if $e' \in C$ then $e' < e$,
  (ii) if $e' \notin C$ and $\neg I(a, \lambda(e'))$, then $e \# e'$,
  (iii) otherwise, i.e. if $e' \notin C$ and $I(a, \lambda(e'))$, then e and e' are concurrent.

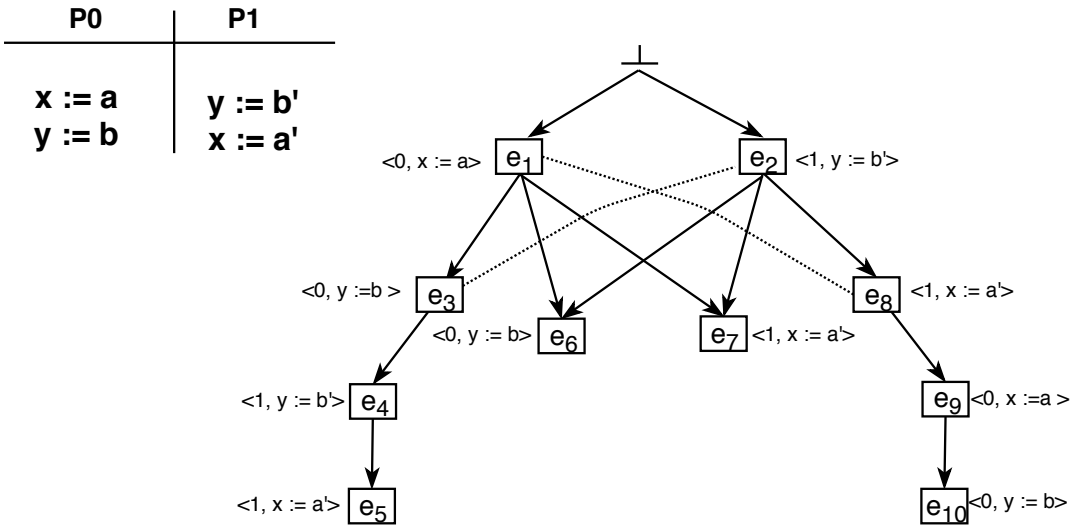


Figure 3.3 – A toy program, and its unfolding semantics.

Given a PES that is built from a LTS by Definition 6, we define the *state* of a configuration C in the PES, and denote by $state(C)$, the state s of the LTS obtained by executing all the actions associated with the events in C in any order compatible with the causality order. We write $enab(C) = enabled(state(C)) \subseteq \Sigma$ for the set of actions enabled at $state(C)$, while $actions(C)$ denotes the set of actions labelling events in C, i.e. $actions(C) = \{\lambda(e) : e \in C\}$.

Example 3 *Figure 3.3 displays a concurrent program P composed of two concurrent processes $P0$ and $P1$. There are two global variables x and y. Both processes $P0$ and*

$P1$ write values to the two global variables. The independent relation is $I(x := a, y := b')$ and $I(x := a', y := b)$. We now build the unfolding semantics for the program.

We first create an empty unfolding (demonstrated by a $\perp$ symbol in the figure) with only one empty configuration. Step 2 is creating all new events by finding a configuration C that enables an action a which is dependent on all $maxEvents(C)$. Having only one empty configuration with no maximal events, and two enabled actions at initial state: $x := a$ and $y := b$, we create events e_1 and e_2.

The unfolding now has three configurations to consider: $\{e_1\}$, $\{e_2\}$, and $\{e_1, e_2\}$. Action $y := b$ is enabled at $state(\{e_1\})$ and it is dependent on $\lambda(e_1)$, so we create e_3. Similarly action $x := a' \in enabled(state(\{e_2\}))$, and it is dependent on $\lambda(e_2)$, we then create e_8. Both actions $y := b$ and $x := a'$ are enabled at $state(\{e_1, e_2\})$ and they are dependent on $\lambda(e_1)$ and $\lambda(e_2)$ where both e_1 and e_2 are maximal events of the configuration $\{e_1, e_2\}$. Thus, e_6 and e_7 are created. Similarly, we can create the remaining events: e_4, e_5, e_9, and e_{10}.

There are in total three maximal configurations, which are $\{e_1, e_3, e_4, e_5\}$, $\{e_1, e_2, e_6, e_7\}$, and $\{e_2, e_8, e_9, e_{10}\}$.

Definition 6 presents how to build a Σ-PES from an LTS. Conversely, from a Σ-PES $\mathcal{E} = \langle E, <, \#, \lambda, \rangle$ one can define a corresponding LTS $\mathcal{T}$ = $\langle S, I, \Sigma, \rightarrow \rangle$ where $S = conf(E)$, $I = \{\perp\}$, and the transition relation $\rightarrow = \{(s, a, s') \mid s, s' \in config(E), s' = s \cup \{e\}, e \in E, \lambda(e) = a\}$.

3.2 Unfolding-based dynamic partial order reduction

The unfolding semantics is a compact representation for modeling behaviors of concurrent programs. Each maximal configuration of the unfolding of a program corresponds to a Mazurkiewicz trace representing an equivalence class of executions of the program. UDPOR [27] is an optimal DPOR technique that explores a reduced LTS of the program, based on the unfolding semantics, without redundant explorations of Mazurkiewicz traces. Algorithm 2 presents the UDPOR exploration algorithm of [27]. As other DPOR algorithms, it explores only a part of the LTS of a given terminating distributed program P according to an independence relation I, while ensuring that the explored part is sufficient to detect all deadlocks. The particularity of UDPOR is to use the concurrency semantics explicitly, namely unfoldings, which makes it both complete

and optimal, in the sense that it explores exactly one interleaving per Mazurkiewicz trace, never reaching any sleep-set blocked execution.

Algorithm 2: Unfolding-based POR exploration

```
1  Procedure Explore(C, D, A)
2      Compute extensions of C (ex(C))
3      Add all events in ex(C) to U
4      if en(C) ⊆ D then
5          Return
6      if (A = ∅) then
7          choose e from en(C) \ D
8      else
9          choose e from A ∩ en(C)
10     Explore(C ∪ {e}, D, A \ {e})
11     if ∃J ∈ Alt(C, D ∪ {e}) then
12         Explore(C, D ∪ {e}, J \ C)
13     U := U ∩ Q_{C,D}
```

The algorithm works as follows. Executions are represented by configurations, thus equivalent to their Mazurkiewicz traces. The set U, initially empty, contains all events met so far in the exploration. The procedure $Explore$ has three parameters: a configuration C encoding the current execution; a set D (for *disabled*) of events to avoid, playing a role similar to a sleep set in [17], thus preventing revisits of configurations; a set A (for *add*) of events conflicting with D and used to guide the search to events in conflicting configurations in $cex(C)$ to explore alternative executions.

First, all extensions of C are computed and added to U (line 4). The search backtracks (line 6) in two cases: when C is maximal ($en(C) = \emptyset$, *i.e.* a deadlock (or the program end) is reached), or when all events enabled in C should be avoided ($en(C) \subseteq D$), which corresponds to a redundant call, thus a sleep-set blocked execution. Otherwise, an enabled event e is chosen (lines 7-10) in A if this guiding information is non empty (line 10), and a "left" recursive exploration $Explore(C \cup \{e\}, D, A \setminus \{e\})$ is called (line 11) from this extended configuration $C \cup \{e\}$, it continues trying to avoid D, but e is removed from A in the guiding information. When this call is completed, all configurations containing C and e have been explored, thus it remains to explore those that contain C but not e.

Alternatives (see Definition 7) are computed (line 12) with the function call $Alt(C, D \cup \{e\})$. Alternatives play a role similar to “backtracking sets" in the original DPOR algo-

rithm, *i.e.* sets of actions that must be explored from the current state. If an alternative J exists, a right "recursive" exploration is called $Explore(C, D \cup \{e\}, J \setminus C)$: C is still the configuration to extend, but e is now also to be avoided, thus added to D, while events in $J \setminus C$ are used as guides. Upon completion (line 14), U is intersected with $Q_{C,D}$ which includes all events in C and D as well as every event in U conflicting with some events in $C \cup D$, formally $Q_{C,D} := C \cup D \cup \bigcup_{e \in (C \cup D), e' \in \#_U^i(e)} [e']$.

Definition 7 (Alternatives [27]) *Let D and U be sets of events, and $C \subseteq U$ a configuration. An alternative to D after C in U is a configuration J such that: (i) $J \cup C$ is a configuration, (ii)* $\forall e \in D, \exists e' \in (C \cup J) : e' \in \#_U^i(e)$

In order to avoid sleep-set blocked executions (SSB) and obtain the optimality of DPOR, the function $Alt(C, D \cup \{e\})$ has to solve an NP-complete problem [34]: find a subset J of U that can be used for backtracking, conflicts with all $D \cup \{e\}$ thus necessarily leading to a configuration $C \cup J$ that is not already visited. In this case the condition $en(C) \subseteq D$ can then be replaced by $en(C) = \emptyset$ in line 5. Note that with a different encoding, Optimal DPOR must solve the same problem [1] as explained in [34]. In [34], a variant of the algorithm is proposed for the function Alt that computes *k-partial alternatives* rather than alternatives, *i.e.,* sets of events J conflicting with only k events in D, not necessarily all of them. Depending on k, (*e.g.* $k = \infty$ (or $k = |D| + 1$) for alternatives, $k = 1$ for source sets of [1]) this variant allows to tune between an optimal or a quasi-optimal algorithm that may be more efficient.

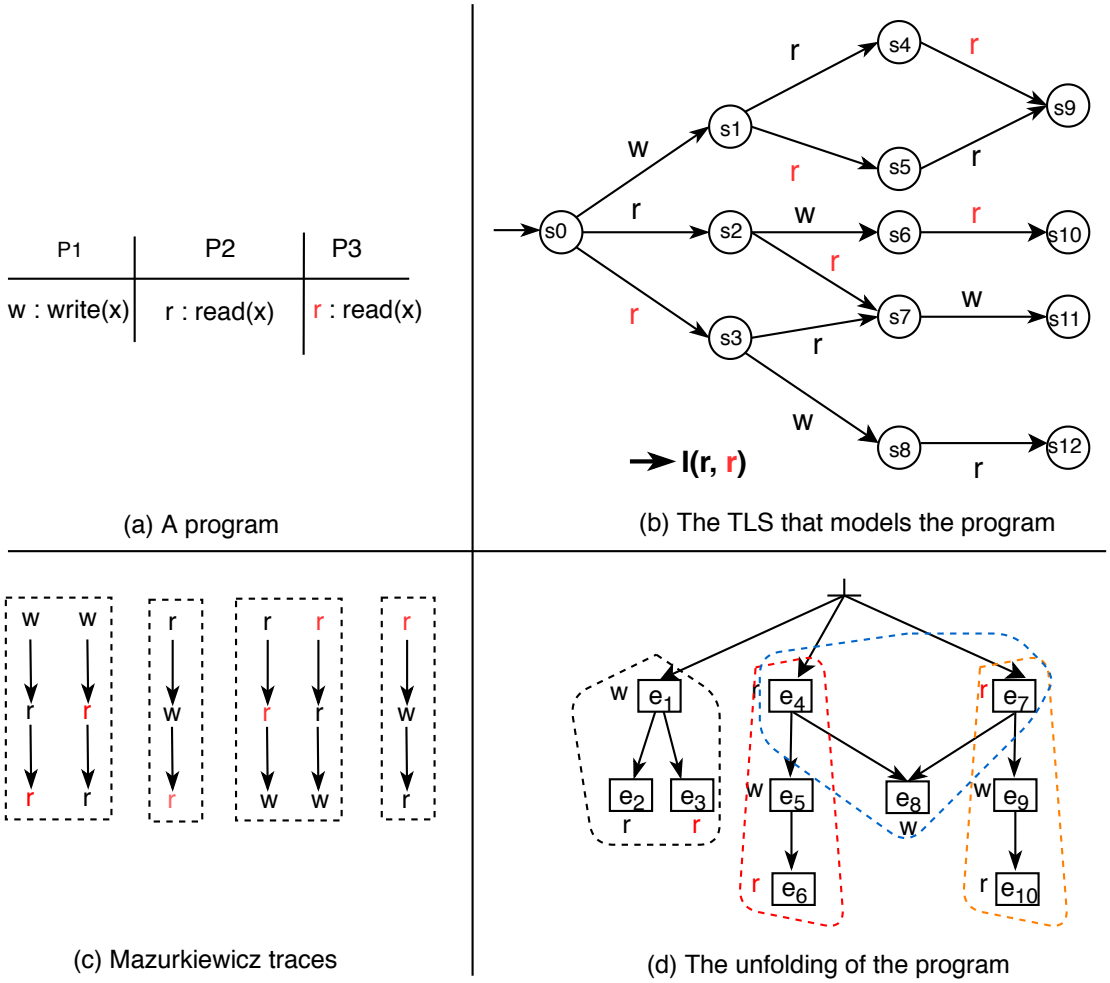
Definition 8 (k-partial alternative [34]) *Let D and U be sets of events, $C \subseteq U$ a configuration, $k \in Z$ (a non-negative integer). A configuration J is a k-partial alternative to D after C if there is some $\hat{D} \subseteq D$ such that $|\hat{D}| = k$ and J is an alternative to $\hat{D}$ after C.*

While an alternative needs to conflict with all events in D, a k-partial alternative requires conflicting with only k events. Obviously, by reducing the number of events that a k-partial alternative needs to conflict with, computing a k-partial alternative could be faster than computing an alternative. However, since conflicting with only k events, a k-partial alternative can guide the UDPOR exploration to revisit some Mazurkiewicz trace, leading to a sleep-set blocked execution. Thus, the k-partial alternative is a trade-off solution between accepting sleep-set blocked executions and having redundant explorations versus solving the NP-complete problem. Surprisingly, in some experiments

in [34], with low values of k, using k-partial alternative UDPOR is still optimal. The optimality may be explained by the fact that the calculation of k-partial alternatives was fortunate, since conflicting with only k events in D is enough to conflict with all other events. In [34], authors propose a new concept of *comb* that facilitates the computation of k-partial alternatives in polynomial time.

3.3 Conclusion

This chapter provides some basic notions related to interleaving semantics, namely labelled transition systems and its properties. Besides, concurrent semantics, namely event structures, are also discussed.



(a) A program

(b) The TLS that models the program

(c) Mazurkiewicz traces

(d) The unfolding of the program

Figure 3.4 – The connection between interleaving semantics and unfolding semantics.

Figure 3.4 summarizes the connection between interleaving semantics and unfolding semantics on an example. A concurrent program consists of three processes shown in Figure 3.4 a. This program can be modeled by a simple LTS (Figure 3.4 b) including six different executions belonging to four different Mazurkiewicz traces (Figure 3.4 c). The behaviors of the program can also be represented by a compact presentation, its unfolding semantics (Figure 3.4 d). In the unfolding, each maximal configuration (in dashed lines) corresponds to a Mazurkiewicz trace.

Besides introducing concurrent and interleaving semantics, UDPOR, an optimal Dynamic partial order reduction, is also described. While most standard dynamic partial order reduction methods are based on the interleaving semantics, UDPOR operates over a concurrent semantics named prime event structure. The next chapter will introduce a proposed abstract model for asynchronous distributed programs that we consider throughout the thesis as well as how we specify the model and prove independence relations.

CHAPTER 4

COMPUTATION MODEL OF ASYNCHRONOUS DISTRIBUTED PROGRAMS

This chapter aims to present our abstract model of distributed programs. The properties of the model, as well as the independence theorems that will be utilized to compute independence relations between actions, are also discussed. The abstract model is compact, it consists of only nine actions, but many MPI functions can be simulated by this abstract model in the SimGrid simulator. Determining dependent actions requires to specify the semantics of the programs. Thus, we formally describe the semantics of the abstract model in TLA+. Having a precise specification of the model, we can then reason about the dependency of actions.

4.1 Informal description of the model

In our model [38], we consider that an asynchronous distributed program P consists of a set of n actors Actors = $\{A_1, A_2, ...A_n\}$ that perform local actions, communicate asynchronously with each other, and share some resources. We assume that the program is terminating, which implies that all actions are terminating. All local actions are abstracted into a unique one *LocalComp*. Communication actions are of four types: *AsyncSend*, *AsyncReceive*, *TestAny*, and *WaitAny*. Actions on shared resources called *synchronizations* are of four types: *AsyncMutexLock*, *MutexUnlock*, *MutexTestAny*, and *MutexWaitAny*.

At the semantic level, P is a tuple $P = \langle \text{Actors}, \text{Network}, \text{Synchronization} \rangle$ where Network and Synchronization respectively describe the abstract objects, and the effects on these of the communication and synchronization actions. The Network subsys-

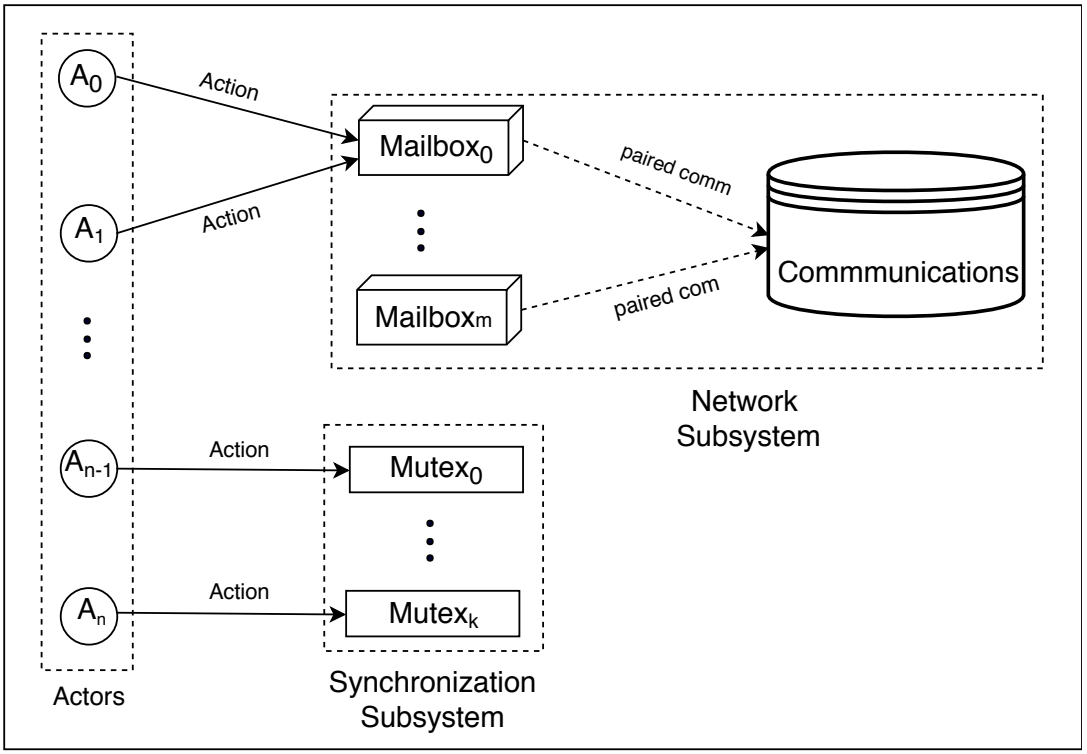


Figure 4.1 – Main elements of the model: Actors, Network and Synchronization.

tem provides facilities for the Actors to asynchronously communicate with each other, while the subsystem Synchronization allows the synchronization of actors on the access to shared resources.

Network subsystem. The state of the Network subsystem is defined as a pair: $\langle \text{Mailboxes}, \text{Communications} \rangle$, where Mailboxes is a set of mailboxes storing unpaired communications, while Communications stores only paired ones. Each communication c has a status in $\{\textit{send}, \textit{receive}, \textit{done}\}$, ids of source and destination actors, data addresses for those.

A mailbox is a rendez-vous point where *send* and *receive* communications meet. It is modelled as an unbounded FIFO queue that is either empty or stores communications with all same *send* or *receive* status, waiting for a matching opposite communication. When matching occurs, this paired communication gets a *done* status and is appended to the set Communications.

We now detail the effect in actor A_i of the communication actions on Mailboxes and Communications:

— c = *AsyncSend*$(m, data)$ drops an asynchronous *send* communication c to the mailbox m. If pending *receive* communications exist in the mailbox, c is paired with the oldest one c' to form a communication with *done* status in Communications, the *receive* communication is removed from m and the $data$ is copied from the source to the destination. Otherwise, a pending communication with *send* status is appended to m.

— c' = *AsyncReceive*(m', d) drops an asynchronous *receive* communication to mailbox m'; the way a *receive* communication is processed is similar to *send*. If pending *send* communications exist, c' is paired with the oldest one c to form a communication with *done* status in Communications, the *send* communication is removed from m, and the data of the *send* is copied to d. Otherwise, a pending communication with *receive* status is appended to m.

— *TestAny*(Com) tests a set of communications Com of A_i. It returns a boolean which value is $true$ if and only if a communication in Com with *done* status exists.

— *WaitAny*(Com) waits for a set of communications Com of A_i. The action is blocking until at least one communication in Com has a *done* status.

Example 4 *Figure 4.2 visualizes the main steps of sending data between two actors A_i and A_j. Actor A_i firstly drops a send communication to a particular mailbox. Suppose the mailbox is empty before the send communication arrives, then now only the send communication is stored in the mailbox. Secondly, actor A_j posts a receive communication to the same mailbox. The send and receive communications are paired to create a done communication in Communications, and then data is copied from A_i to A_j. When applying the reverse order in which the receive communication is posted to the mailbox before the send communication arrives, we also get the same result, creating a done communication and copying the data from A_i to A_j. Actor A_i and A_j can test if their communications are done by executing TestAny actions or wait for their communications by performing WaitAny actions.*

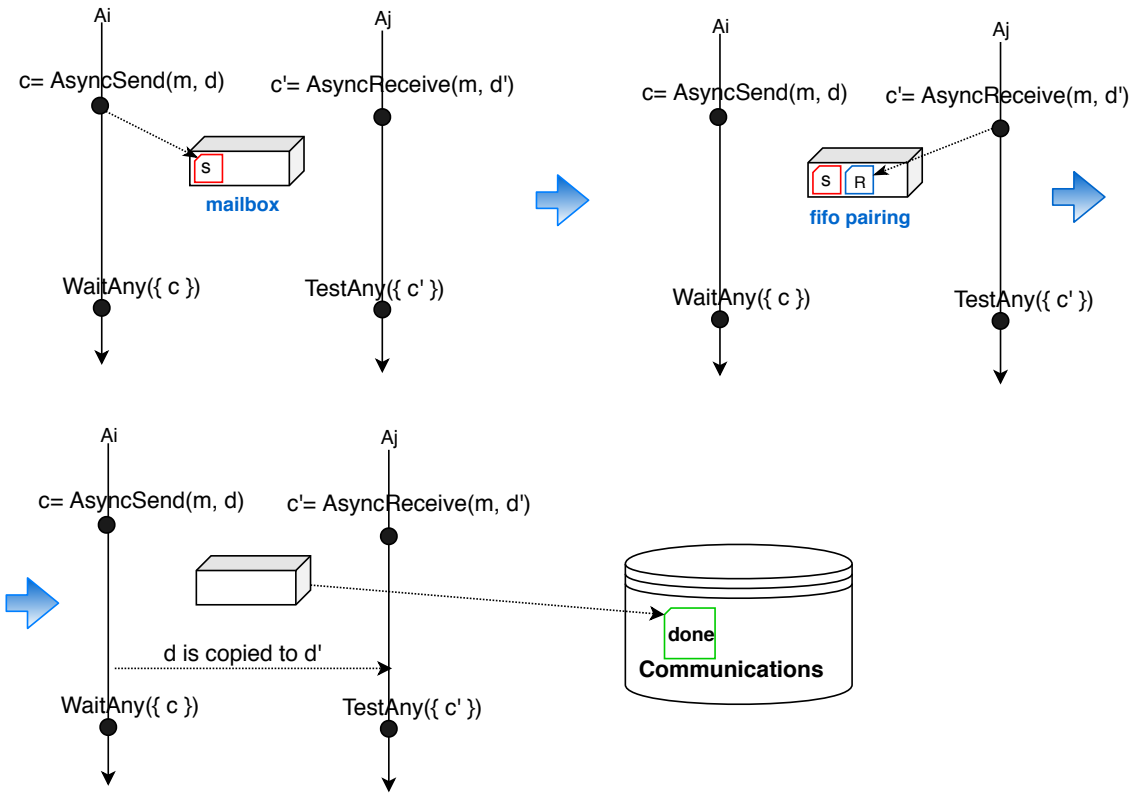


Figure 4.2 – Main steps of the communication

Synchronization subsystem. The Synchronization subsystem consists in a pair: $\langle \text{Mutexes}, \text{Requests} \rangle$ where Mutexes is a set of asynchronous mutexes used to synchronize the actors, and Requests is a vector indexed by actors ids of sets of requested mutexes. Each mutex m_j is represented by a FIFO queue of actors ids i who declared their interest on a mutex m_j by executing the action *AsyncMutexLock*(m_j) as depicted in Figure 4.3. A mutex m_j is *free* if its queue is empty, *busy* otherwise. The *owner* is the actor whose id is the first in the queue. In actor A_i, the effect of the synchronization actions on Mutexes and Requests is as follows:

— *AsyncMutexLock*(m_j) requests a mutex m_j with the effect of appending the actor id i to m_j's queue and adding j to Requests$[i]$. A_i is *waiting* until *owning* m_j but, unlike classical mutexes, waiting is not necessarily blocking.

— *MutexUnlock*(m_j) removes its interest to a mutex m_j by deleting the actor id i from the m_j's queue and removing j from Requests$[i]$.

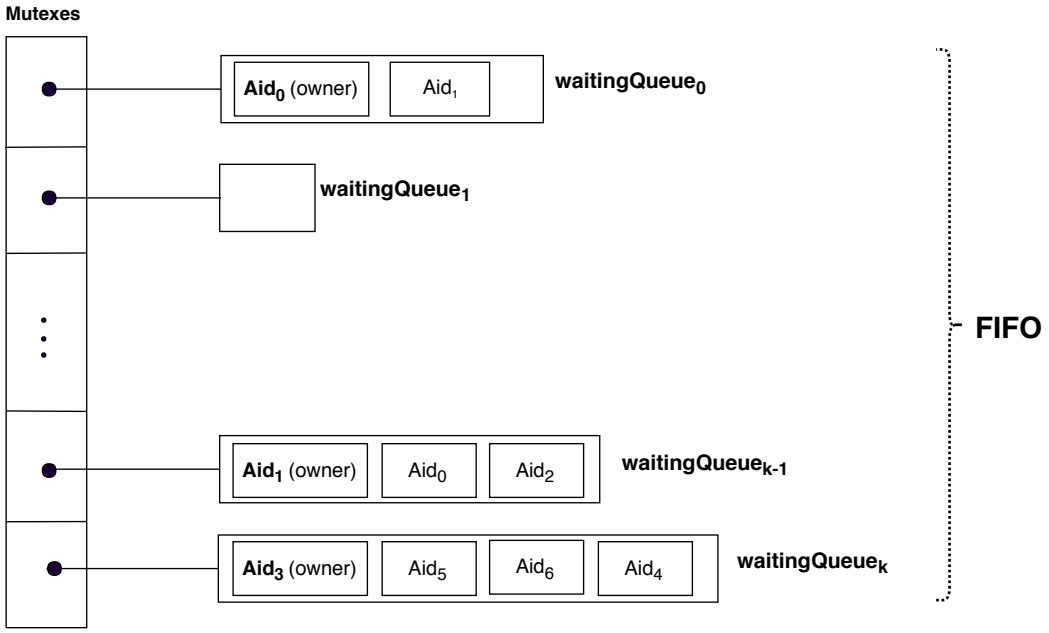


Figure 4.3 – Each mutex uses a waiting queue consisting of actor identifiers.

— *MutexTestAny*(M) checks if actor A_i owns some previously requested mutex m_j in set M (j is in Requests$[i]$), returning *true* if it is the case.

— *MutexWaitAny*(M) blocks until A_i owns some mutex m_j in M. Note that *MutexTestAny* (resp. *MutexWaitAny*) are similar to *TestAny* (resp. *WaitAny*) and could be merged. We keep them separate here for simplicity of explanations.

Example 5 *Figure 4.4 visualizes how synchronization actions work in a simple case. Two actors A_i and A_j want to access a shared resource that is protected by a mutex. The actors declare their interests on the mutex by executing AsyncMutexLock actions. Suppose actor A_i executes an AsyncMutexLock first, then it is the owner of the mutex. If actor A_j executes an AsyncMutexLock later, it is the second actor in the waiting queue of the mutex (its id is the second element in the waiting queue). Then A_i tests if it is the owner of the mutex by performing a MutexTestAny. Since the test returns $true$, A_i enters the shared resource. Meanwhile A_j is blocked because it waits to become the owner of the mutex by employing a MutexWaitAny action. After working with the shared resource, A_i executes a MutexUnlock to remove its interest on the mutex. Removing the id of A_i from the waiting queue entails A_j becomes the mutex owner, and A_j can execute the MutexWaitAny action. After that A_j can access the shared resource.*

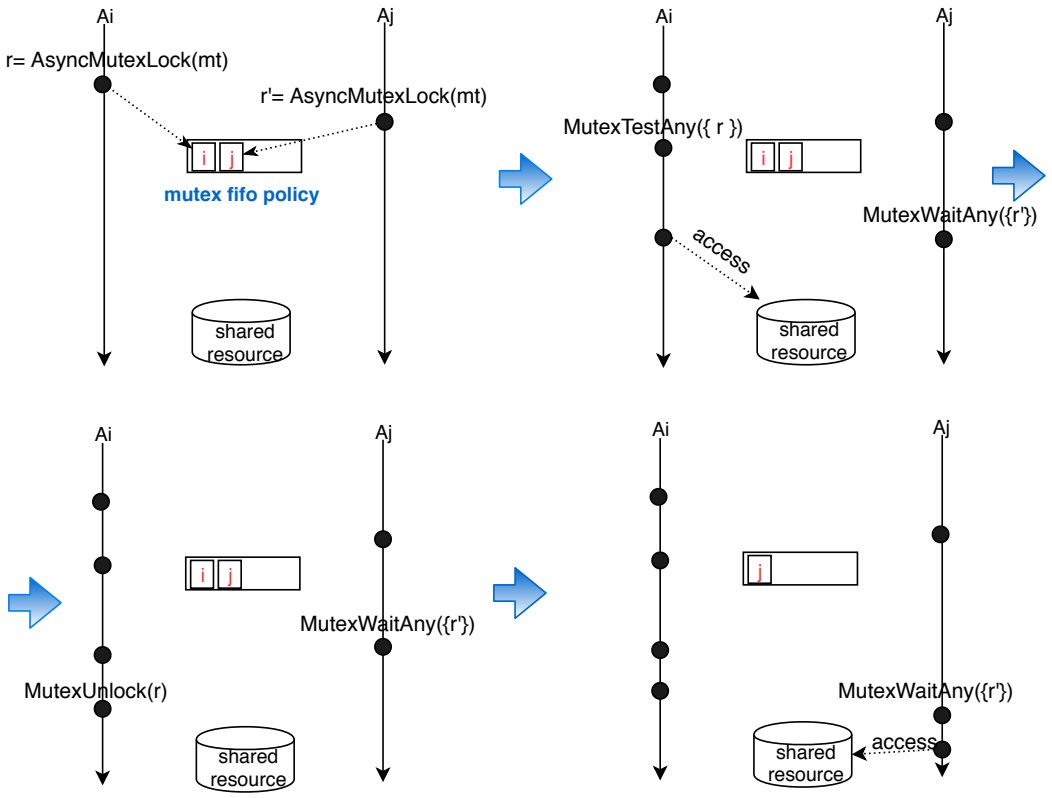


Figure 4.4 – Actors are synchronized by using synchronization actions

In addition to the above actions, a program can have local computations named *LocalComp* actions. Such actions do not intervene with shared objects (Mailboxes, Mutexes and Communications), and they can be responsible for I/O tasks.

We specified our model of asynchronous distributed systems in the formal language TLA+ [28]. Our TLA+ model, presented in the following, is also available online [1]. It focuses on how actions transform the global state of the system. An instance P of a program is described by a set of actors and their actions (representing their source code). Following the semantics of TLA+, and since programs are terminating, the interleaving semantics of a program P can be described by an acyclic LTS representing all its behaviors. Formally, the LTS of P is a tuple $\mathcal{T}_{\mathcal{P}} = \langle S, s_0, \Sigma, \rightarrow \rangle$ where Σ represents the actions of P; a state $s =< l, g >$ in S consists of the local state l of all actors (*i.e.* local variables, Requests) and the state g of all shared objects including Mutexes, Mailboxes and Communications. In the initial state s_0, all actors are in their initial local state, sets and FIFO queues are empty. A transition $t : s \xrightarrow{a} s' \in \rightarrow$ is defined if, ac-

1. https://github.com/pham-theanh/simixNetworks

cording to the TLA+ model, the action encoded by a is enabled at s and executing a transforms the state from s to s'.

Notice that when verifying a real program, we only observe its actions and assume that they respect the proposed TLA+ model and the independence relation discussed below. These assumptions are necessary to suppose that the LTS correctly models the actual program behaviors.

4.2 Model specification

In order to get the formal semantics of the abstract model, we use the TLA+ language [28]. TLA+ is a formal specification language for high-level modeling of concurrent systems. The language that first appeared in 1999 was developed by Leslie Lamport. The TLA+ specification of a system describes all possible actions (behaviors) of the system by using the Temporal Logic of Actions (TLA). So, each action is specified as a formula of TLA describing how the states of the system evaluate, in which a state is characterized by values of variables. The evolution of the system starts at the initial state (initial predicate) denoting the initial condition, and it transforms into a new state depending on what action is taken, in which the action is decided by the system's next-state relation. One of the main advantages of TLA+ is that a TLA+ specification can be model-checked by using the TLC model checker to verify given properties defined by users.

The figure 4.5 presents the data model of the abstract model. The state of the system is represented by six variables: $Communications$, $Mailboxes$, $Memory$, $Mutexes$, $nbComMbs$ and $Requests$. Communications is the set of all paired communications in the system (i.e. when a pair of send and receive requests have been matched). $Mailboxes$ is an array indexed by $MailboxesIds$. $Mailboxes[i]$ is a FIFO queue which stores send or receive communications (unpaired communications). $Memory$ is an array indexed by $Actors$ (a set of ids), and $Memory[i]$ is a memory local to the actor i used to store ids of communications and indexed by $Adresses$. $Mutexes$ is an array indexed by $MutexIds$, and $Mutexes[i]$ is a FIFO queue that remembers which actors have required the mutex i. $Requests$ is an array indexed by $Actors$; $Requests[i]$ is the set of mutexes (ids of mutexes) requested by the actor i. Lastly, $nbComMbs$ is a array indexed by $MailboxIds$, and it is used to set ids for communications concerning $Mailboxes[i]$.

MODULE $abstractModel$

EXTENDS $Integers$, $Naturals$, $Sequences$, $FiniteSets$, TLC
CONSTANTS $Actors$, $MailboxesIds$, $MutexIds$, $Addresses$
VARIABLES $Communications$, $Memory$, $Mutexes$, $Requests$, $Mailboxes$, $nbComMbs$
$NoActor \triangleq$ "NoActor"
$NoAddr \triangleq$ "NoAddress"
$Comm \triangleq [id$: STRING,
$status : \{$"send", "receive", "done"$\}$,
$src : Actors \cup \{NoActor\}$,
$dst : Actors \cup \{NoActor\}$,
$data\_src : Addresses \cup \{NoAddr\}$,
$data\_dst : Addresses \cup \{NoAddr\}]$

$$
\begin{aligned}
TypeInv \triangleq\ & \land\ Communications \in \text{SUBSET } Comm \\
& \land\ \forall\, c \in Communications : c.status = \text{"done"} \\
& \land\ Mailboxes \in [MailboxesIds \to Seq(Comm)] \\
& \land\ \forall\, mbId \in MailboxesIds : \forall\, i \in 1\,..\,Len(Mailboxes[mbId]) : \\
& \qquad \land\ Mailboxes[mbId][i].status \in \{\text{"send", "receive"}\} \\
& \qquad \land\ Mailboxes[mbId][i].status = Mailboxes[mbId][1].status \\
& \land\ Mutexes \in [MutexIds \to Seq(Actors)] \\
& \land\ \forall\, mId \in MutexIds : \forall\, id \in \text{DOMAIN } Mutexes[mId] : Mutexes[mId][id] \in Actors \\
& \land\ Requests \in [Actors \to Seq(MutexIds)] \\
& \land\ Memory \in [Actors \to [Addresses \to \text{STRING}]] \\
& \land\ nbComMbs \in Nat
\end{aligned}
$$

Figure 4.5 – TLA+ specification of the abstract model: data model.

AsyncSend specification. Figure 4.6 expresses the specification for the *AsyncSend* action. The actor aId sends a "send" communication to the mailbox $mbId$. If a pending "receive" communication already exists, they are combined to create a "done" paired communication in $Communications$ and data is copied from the source to the destination, otherwise a new communication with "send" status is created. Address $data\_addr$ of actor aId contains the data to transmit while memory address $comm\_addr$ of actor aId is assigned the id of the communication.

$$
\begin{array}{l}
AsyncSend(aId,\ mbId,\ data\_addr,\ comm\_addr) \triangleq \\
\quad \land\ aId \quad \in Actors \\
\quad \land\ mbId \in MailboxesIds \\
\quad \land\ data\_addr \quad \in Addresses \\
\quad \land\ \lor\ \land\ \lor\ Len(Mailboxes[mbId]) = 0 \\
\qquad\qquad \lor\ \land\ Len(Mailboxes[mbId]) > 0 \\
\qquad\qquad\quad \land\ Head(Mailboxes[mbId]).status = \text{“send”} \\
\qquad\quad \land\ \mathsf{LET}\ comm \triangleq \\
\qquad\qquad\qquad [id \mapsto\ converToString(nbComMbs[mbId], \text{“Mb”}, mbId), \\
\qquad\qquad\qquad\ status \mapsto \text{“send”},\ src \mapsto aId,\ dst \mapsto NoActor, \\
\qquad\qquad\qquad\ data\_src \mapsto data\_addr,\ data\_dst \mapsto NoAddr] \\
\qquad\qquad \mathsf{IN} \\
\qquad\qquad\quad \land\ Mailboxes' = [Mailboxes\ \mathsf{EXCEPT}\ ![mbId] = Append(Mailboxes[mbId],\ comm)] \\
\qquad\qquad\quad \land\ Memory' = [Memory\ \mathsf{EXCEPT}\ ![aId][comm\_addr] = comm.id] \\
\qquad\qquad\quad \land\ \mathsf{UNCHANGED}\ \langle Communications \rangle \\
\qquad\qquad\quad \land\ nbComMbs' = [nbComMbs\ \mathsf{EXCEPT}\ ![mbId] = nbComMbs[mbId] + 1] \\
\qquad \lor\ \land\ Len(Mailboxes[mbId]) > 0 \\
\qquad\quad \land\ Head(Mailboxes[mbId]).status = \text{“receive”} \\
\qquad\quad \land\ \mathsf{LET}\ comm \triangleq Head(Mailboxes[mbId]) \\
\qquad\qquad \mathsf{IN} \quad \land\ Communications' = \\
\qquad\qquad\qquad\ Communications \cup \{[comm\ \mathsf{EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad\qquad !.status = \text{“done”},\ !.src = aId, \\
\qquad\qquad\qquad\qquad\qquad\qquad !.data\_src = data\_addr]\} \\
\qquad\qquad\qquad\ \land\ Mailboxes' = [Mailboxes\ \mathsf{EXCEPT}\ ![mbId] = Tail(Mailboxes[mbId])] \\
\qquad\qquad\qquad\ \land\ Memory' = [Memory\ \mathsf{EXCEPT}\ ![comm.dst][comm.data\_dst] = \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Memory[aId][data\_addr]] \\
\qquad\qquad\qquad\ \land\ \mathsf{UNCHANGED}\ \langle nbComMbs \rangle \\
\quad \land\ \mathsf{UNCHANGED}\ \langle Mutexes,\ Requests \rangle
\end{array}
$$

Figure 4.6 – TLA+ specification of the abstract model: AsyncSend action.

AsyncReceive specification. Similarly, the actor aId posts a "receive" communication to the mailbox $mbId$. If there is a pending "send" communication in the same mailbox $mbId$, they are combined to create a "done" communication in $Communications$ and

data is copied from the source to the destination, otherwise a new communication is created with "receive" status. Figure 4.7 presents the specification of the *AsyncReceive* action.

$$
\begin{array}{l}
AsyncReceive(aId,\ mbId,\ data\_addr,\ comm\_addr)\ \triangleq \\
\quad \land\ aId\ \ \in Actors \\
\quad \land\ mbId \in MailboxesIds \\
\quad \land\ \lor\ \land\ \lor Len(Mailboxes[mbId]) = 0 \\
\qquad\qquad \lor\ \land Len(Mailboxes[mbId]) > 0 \\
\qquad\qquad\quad \land Head(Mailboxes[mbId]).status = \text{“receive”} \\
\qquad \land \text{LET } comm\ \triangleq \\
\qquad\qquad [id \mapsto converToString(nbComMbs[mbId], \text{“Mb”},\ mbId), \\
\qquad\qquad\ status \mapsto \text{“receive”},\ src \mapsto NoActor,\ dst \mapsto aId, \\
\qquad\qquad\ data\_src \mapsto NoAddr,\ data\_dst \mapsto data\_addr] \\
\qquad\ \text{IN} \\
\qquad\quad \land\ Mailboxes' = [Mailboxes\ \text{EXCEPT}\ ![mbId] = Append(Mailboxes[mbId],\ comm)] \\
\qquad\quad \land\ Memory' = [Memory\ \text{EXCEPT}\ ![aId][comm\_addr] = comm.id] \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle Communications \rangle \\
\qquad\quad \land\ nbComMbs' = [nbComMbs\ \text{EXCEPT}\ ![aId] = nbComMbs[aId] + 1] \\
\quad \lor\ \land Len(Mailboxes[mbId]) >\ 0 \\
\qquad \land\ Head(Mailboxes[mbId]).status = \text{“send”} \\
\qquad \land \text{LET } comm\ \triangleq\ Head(Mailboxes[mbId]) \\
\qquad\ \text{IN}\quad \land\ Communications' = \\
\qquad\qquad\quad Communications \cup \{[comm\ \text{EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad !.status = \text{“done”},\ !.dst = aId, \\
\qquad\qquad\qquad\qquad\qquad !.data\_dst = data\_addr]\} \\
\qquad\qquad\ \land\ Memory' = [Memory\ \text{EXCEPT}\ ![aId][data\_addr] = \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Memory[comm.src][comm.data\_src]] \\
\qquad\qquad\ \land\ Mailboxes' = [Mailboxes\ \text{EXCEPT}\ ![mbId] = Tail(Mailboxes[mbId])] \\
\qquad\qquad\ \land\ \text{UNCHANGED}\ \langle nbComMbs \rangle \\
\quad \land\ \text{UNCHANGED}\ \langle Mutexes,\ Requests \rangle
\end{array}
$$

Figure 4.7 – TLA+ specification of the abstract model: AsyncReceive action.

***WaitAny* action specification.** The actor aId waits for at least one communication from a given set $comm\_addrs$ (a list of addresses in the memory where ids of communications are stored) to complete. If at least one communication is already "done", there is nothing to do, else the function is blocking.

$$
\begin{array}{l}
WaitAny(aId,\ comm\_addrs) \triangleq \\
\quad \land\ aId \in Actors \\
\quad \land\ \exists\, comm\_addr \in comm\_addrs,\ comm \in Communications : \\
\qquad\qquad comm.id = Memory[aId][comm\_addr] \\
\quad \land\ \text{UNCHANGED}\ \langle Mutexes,\ Requests,\ Mailboxes,\ nbComMbs,\ Memory,\ Communications \rangle
\end{array}
$$

Figure 4.8 – TLA+ specification of *WaitAny* action.

***TestAny* specification.** Actor aId tests a set of communications $comm\_addrs$, and returns a boolean value at memory address $testResult\_Addr$. If there is at least one done communication, then the function returns value "true", otherwise it returns value "false". Besides, the function is never blocking.

$$
\begin{array}{l}
TestAny(aId,\ comm\_addrs,\ testResult\_Addr) \triangleq \\
\quad \land\ aId \in Actors \\
\quad \land\ \lor\ \land\ \exists\, comm\_addr \in comm\_addrs,\ comm \in Communications : \\
\qquad\qquad comm.id = Memory[aId][comm\_addr] \\
\qquad \land\ Memory' = [Memory\ \text{EXCEPT}\ ![aId][testResult\_Addr] = \text{“true”}] \\
\qquad \lor\ \land\ \neg\exists\, comm\_addr \in comm\_addrs,\ comm \in Communications : \\
\qquad\qquad comm.id = Memory[aId][comm\_addr] \\
\qquad \land\ Memory' = [Memory\ \text{EXCEPT}\ ![aId][testResult\_Addr] = \text{“false”}] \\
\quad \land\ \text{UNCHANGED}\ \langle Mutexes,\ Requests,\ Mailboxes,\ Communications,\ nbComMbs \rangle
\end{array}
$$

Figure 4.9 – TLA+ specification of *TestAny* action.

***AsyncMutexLock* specification.** The actor aId requests a lock on mutex mId. If it has no pending request on the mutex, a new request is created and added to $Requests[aId]$ and the id of the actor is appended to $Mutexes[mId]$.

$$
\begin{array}{l}
AsyncMutexLock(aId,\ mId)\ \triangleq \\
\quad \land\ aId\ \in Actors \\
\quad \land\ mId \in MutexIds \\
\quad \land\ req\_addr \in Addresses \\
\quad \land\ \neg isMember(aId,\ Mutexes[mId]) \\
\quad \land\ \ Requests' = [Requests\ \texttt{EXCEPT}\ ![aId] = Requests[aId] \cup \{mId\}] \\
\quad \land\ \ Mutexes' = [Mutexes\ \texttt{EXCEPT}\ ![mId] = Append(Mutexes[mId],\ aId)] \\
\quad \land\ \texttt{UNCHANGED}\ \langle Communications,\ Mailboxes,\ \ nbComMbs,\ \ Memory\rangle
\end{array}
$$

Figure 4.10 – TLA+ specification of *AsyncMutexLock* action.

***MutexUnlock* specification.** Actor aId releases mutex mId. It is either a normal unlock when aId owns the mutex, or a cancel of the pending request otherwise. In both cases all links between the mutex and the actor in $Mutexes[mId]$ and $Requests[aId]$ are removed.

$$
\begin{array}{l}
MutexUnlock(aId,\ mId)\ \triangleq \\
\land\ aId \in Actors \\
\land\ \lor\ \land\ isMember(aId,\ Mutexes[mId]) \\
\qquad \land\ Mutexes' = [Mutexes\ \texttt{EXCEPT}\ ![mId] = remove(aId,\ Mutexes[mId])] \\
\qquad \land\ Requests' = [Requests\ \texttt{EXCEPT}\ ![aId] = Requests[aId] \setminus \{mId\}] \\
\qquad \land\ \texttt{UNCHANGED}\ \langle Memory,\ Communications,\ Mailboxes,\ nbtest,\ nbComMbs\rangle \\
\quad \lor\ \land\ \neg isMember(aId,\ Mutexes[mId]) \\
\qquad \land\ \texttt{UNCHANGED}\ \langle Memory,\ Communications,\ Mailboxes,\ nbComMbs, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Requests,\ Mutexes\rangle
\end{array}
$$

Figure 4.11 – TLA+ specification of *MutexUnlock* action.

***MutexWaitAny* specification.** Actor aId waits for a set of lock requests $requests$. If the actor owns at least one mutex, this functions is enabled; otherwise, it is blocked.

$MutexWaitAny(aId,\ requests)\ \triangleq$
$\land\ aId \in Actors$
$\land \exists\, req \in\ requests : isHead(aId,\ Mutexes[req])$
$\land$ UNCHANGED $\langle Memory,\ Mutexes,\ Requests,\ Communications,\ Mailboxes,\ nbComMbs\rangle$

Figure 4.12 – TLA+ specification of *MutexWaitAny* action.

***MutexTestAny* specification.** Actor aId tests for a set of lock requests $requests$. If the actor owns at least one mutex, the "true" value is assigned to the memory of actor aId at address $testResult\_Addr$ otherwise a "false" value is assigned to $testResult\_Addr$.

$MutexTestAny(aId,\ requests,\ testResult\_Addr)\ \triangleq$
$\land\ aId \in Actors$
$\land\ testResult\_Addr \in Addresses$
$\land\ \ \lor\ \land \exists\, req \in\ requests : isHead(aId,\ Mutexes[req])$
$\land\ Memory' = [Memory$ EXCEPT $![aId][testResult\_Addr] =$ "true"$]$
$\lor\ \land \neg\exists\, req\ \ \in\ requests : isHead(aId,\ Mutexes[req])$
$\land\ Memory' = [Memory$ EXCEPT $![aId][testResult\_Addr] =$ "false"$]$
$\land$ UNCHANGED $\langle Mutexes,\ Requests,\ Communications,\ Mailboxes,\ nbComMbs\rangle$

Figure 4.13 – TLA+ specification of *MutexTestAny* action.

***LocalComp* specification.** A local computation of the actor aId can change the value of this actor's memory at any address.

$Local(aId)\ \triangleq$
$\land\ aId \in Actors$
$\land\ Memory' \in [Actors \to [Addresses \to \{\text{“ ”}\}\ ]]$
$\land$ UNCHANGED $\langle Communications,\ Mutexes,\ Requests,\ Mailboxes,\ nbComMbs\rangle$

Figure 4.14 – TLA+ specification of *LocalComp* action.

4.3 Persistence

The model presented in the previous section may appear unusual because the lock action on mutexes is split into an *AsyncMutexLock* and a *MutexWaitAny* while most works in the literature consider atomic locks. Our model does not induce any loss of generality since synchronous locks can trivially be simulated with asynchronous locks. One reason to introduce this specificity is that this entails the following lemma, that is the key to the efficiency of UDPOR in our model.

Lemma 1 *Let u be a prefix of an execution v of a program in our model. If an action a is enabled after u, it is either executed in v or still enabled after v.*

The lemma implies the following condition that can be formally expressed as follows [2]:

$$\begin{aligned}
&\forall\, action1, action2 \in \{AsyncMutexLock(_,_), MutexUnlock(_,_), MutexWaitAny(_,_),\\
&\quad MutexTestAny(_,_,_), AsyncSend(_,_,_,_), AsyncReceive(_,_,_,_), TestAny(_,_,_),\\
&\quad WaitAny(_,_,_), LocalComp(_)\} :\\
&\qquad \wedge\ action1 \neq action2\\
&\qquad \wedge\ (\textsf{ENABLED}\ \ action1 \wedge \textsf{ENABLED}\ action2)\\
&\qquad \Longrightarrow\ (action1 \implies (\textsf{ENABLED}\ action2)')
\end{aligned}$$

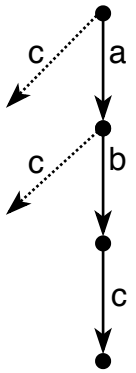


Figure 4.15 – Action c remains enabled until being executed.

Proof 1 *(Sketch) When a is a LocalComp, AsyncSend, AsyncReceive, TestAny, AsyncMutexLock, MutexUnlock, or MutexTestAny action, a cannot be disabled by any new action. Indeed, these actions are never blocking (e.g. AsyncMutexLock (m) comes down*

2. The lemma can be fully expressed in TLA+, but we keep like that for simplicity.

to the addition of an element in a FIFO, which is always enabled) and only depend on the execution of the action right before them by the same actor.

WaitAny and MutexWaitAny may seem more complex. If a is a WaitAny, being enabled after u means that one communication it refers to was paired. Similarly, if a is a MutexWaitAny, being enabled after u means that the corresponding actor is first in the FIFO of a mutex it refers to. In both cases, these facts cannot be modified by any subsequent action of any other actors, so a remains enabled until executed.

Intuitively, this lemma induces that once enabled, actions are never disabled by any subsequent action and remain enabled until executed. This does not hold for classical synchronous locks, as an enabled action *lock(m)* of an actor may be disabled by another actor locking the same mutex first.

4.4 Independence theorems

In order to use DPOR algorithms for our model of distributed programs, and in particular UDPOR that is based on the unfolding semantics, we need to define a valid independence relation for this model. This relation is formally expressed in TLA+ as so-called "independence theorems". We use the term "theorem" since the validity of the independence relation with respect to commutation should be proved [3]. We proved them manually and implemented them as rules in our model checker. Some independence theorems could be enlarged by finding other conditions inducing independence, but this requires expressing more detailed and complex conditions. We give these for simplicity. Intuitively, two actions in distinct actors are independent when they do not compete on shared objects, such as Mailboxes, Communications, or Mutexes. This is formally expressed in theorem 4.4.1. Notice that with the Lemma 1, the enableness of an action is never changed by firing any action in our model. Thus, the disabling condition in Eq3.1 page 41 (*i.e.,* one does not disable the other one) does not need to be checked when we prove the theorems.

Let $ReadVariables(a)$ denote the set of variables (or variable *parts* in case of sets) that are read while evaluating the transition a. Similarly, let $WriteVariables(a)$ denote the set of variable parts that are modified when the transition a is taken. Finally, we

3. What we prove is that actions commute, thus considering them as independent is valid.

note $Variables(a)$ the set of all variable parts involved in transition a. $Variables(a) = ReadVariables(a) \cup WriteVariables(a)$.

Theorem 4.4.1 *Any two actions $a1$ and $a2$ can only be dependent if they have some shared variables. Formally,* $(Variables(a1) \cap Variables(a2) = \emptyset) \Rightarrow I(a1, a2)$

Proof 2 *Whether $a2$ is enabled depends only on the state that is in $ReadVariables(a2)$. Thus if $ReadVariables(a2) \cap Variables(a1) = \emptyset$ then the execution of $a1$ has no impact on whether $a2$ is enabled. Eq3.1 is thus verified in this case.*

In addition, if $WriteVariables(a1) \cap WriteVariables(a2) = \emptyset$ then the order in which actions $a1$ and $a2$ are executed does not impact on the final state because the state modifications are separated. Eq3.2 is thus verified in this case.

Please note that this theorem only introduces a necessary condition to the dependence, but some actions may still involve the same variables and be independent under some conditions. However simple cases of independence will rely on Theorem 4.4.1 as we will now see.

Theorem 4.4.2 *Any pair of communication actions in distinct actors concerning distinct mailboxes are independent.*

$\forall\, act1,\, act2 \in Actors,\, mbId1,\, mbId2 \in MailboxesIds :$
$\land\, act1 \neq act2 \land mbId1 \neq mbId2 \land TypeInv \implies$
$\quad \land\, I(AsyncSend(act1,\, mbId1,\, \_,\, \_),\, AsyncSend(act2,\, mbId2,\, \_,\, \_))$
$\quad \land\, I(AsyncReceive(act1,\, mbId1,\, \_,\, \_),\, AsyncReceive(act2,\, mbId2,\, \_,\, \_))$
$\quad \land\, I(AsyncSend(act1,\, mbId1,\, \_,\, \_),\, AsyncReceive(act2,\, mbId2,\, \_,\, \_))$

Proof 3 *Because the two actions concern different mailboxes, and if they produce new communications, such communications are distinct. Thus, they do not share any variable and according to Theorem 4.4.1, they are independent.*

Theorem 4.4.3 *An AsyncSend is independent of an AsyncReceive of another actor.*

$\forall\, act1,\, act2 \in Actors : act1 \neq act2 \land TypeInv \implies$
$\quad I(AsyncSend(act1,\, \_,\, \_,\, \_),\, AsyncReceive(act2,\, \_,\, \_,\, \_))$

Proof 4 *Suppose we have two actions: a_s of $act1$, and a_r of $act2$. Both actions are enabled at a state s = ⟨ Communications, Mailboxes, nbComMbs, Memory, Mutexes,*

Requests⟩, the first action is c_s = AsyncSend, and second action is c_r = AsyncReceive. Let's firstly prove that Eq3.2 (page 41) is true.

(i) If two actions concern different mailboxes, according to Theorem 4.4.2, they are independent.

(ii) Otherwise, if they concern the same mailbox Mailboxes$[i]$, we have the following cases:

- *If the mailbox is empty, c_s and c_r will be paired together regardless of the order in the mailbox (FIFO queue). The final state after $a_s.a_r$ and after $a_r.a_s$ are thus identical.*
- *If there are some pending sends in the mailbox Mailboxes$[i]$, c_s is added to the tail of the mailbox while c_r is paired with the first pending send communication of the mailbox. In this case, also the final state after $a_s.a_r$ and after $a_r.a_s$ are identical.*
- *Conversely, if the are some pending receive communications in the mailbox, c_r is added to the tail of the mailbox while c_s is paired with the first pending receive communication of the mailbox. We also get the same final state with any order of execution.*

Concerning the second condition (Eq3.1), since the conditions ($e.g, mnId \in MaiboxesIds$) enabling AsyncSend and AsyncReceive actions are not affected by executing other actions, then the condition is satisfied.

Theorem 4.4.4 *Any pair of actions in {TestAny, WaitAny} in distinct actors is independent.*

$$\begin{aligned}\forall\, act1,\, act2 \in Actors : act1 \neq act2 \wedge TypeInv \implies \\ \wedge\, I(WaitAny(act1, \_),\, WaitAny(act2, \_)) \\ \wedge\, I(TestAny(act1, \_, \_),\, TestAny(act2, \_, \_)) \\ \wedge\, I(WaitAny(act1, \_),\, TestAny(act2, \_, \_))\end{aligned}$$

Proof 5 *Let's start with two TestAny actions. Let t_1 and t_2 be TestAny($\{Coms1\}$) action and TestAny($Coms2$) action, respectively. Suppose they are both enabled at state s. With any execution orders, t_1 returns a true boolean value if there is at least one done communication in $Coms1$, otherwise, it returns false. Similarly, t_2 also returns a boolean value decided by the status of communications in $Coms2$. So, the returned values do not depend on the execution orders but depend on the state of communications in $Coms1$ and $Coms2$ independently of $Coms1$ = $Coms2$ or not.*

Besides, executing one action does not change conditions (e.g., $aId \in Actors$*) enabling the other one. Hence, we have I(*t_1, t_2*). Similarly, we have the proof for two WaitAny actions, and a WaitAny action and a TestAny action.*

Theorem 4.4.5 *Any action in* {*TestAny*($Coms$), *WaitAny*($Coms$)} *is independent with any action of another actor in* {*AsyncSend*, *AsyncReceive*} *as soon as they do not both concern the first paired communication in the set* $Coms$.

$$\begin{aligned}
&\forall\, act1,\ act2 \in Actors,\ data,\ comm_addr,\ \ comms \in \text{SUBSET } Addresses :\\
&\quad \wedge\ act1 \neq act2 \wedge TypeInv\\
&\quad \wedge\ firstPaired(Coms) \neq\ Memory[act2][comm_addr] \implies\\
&\quad \wedge\ I(\,WaitAny(act1,\ Coms),\ AsyncSend(act2,\ _,\ _,\ comm_addr))\\
&\quad \wedge\ I(\,WaitAny(act1,\ Coms),\ AsyncReceive(act2,\ _,\ _,\ comm_addr))\\
&\quad \wedge\ I(\,TestAny(act1,\ Coms,\ _),\ AsyncSend(act2,\ _,\ _,\ comm_addr))\\
&\quad \wedge\ I(\,TestAny(act1,\ Coms,\ _),\ AsyncReceive(act2,\ _,\ _,\ comm_addr))
\end{aligned}$$

In the above formal expression, we suppose that function $firstPaired(Coms)$ returns the id of the first paired communication in the set $Coms$.

Proof 6 *Let's start with a TestAny (*$Coms$*) (called action* a_1*) and an AsyncSend (m, _) (called action* a_2*). Suppose they are enabled at a particular state and do not both concern the first done communication* c *in* $Coms$*. There are two cases: (i) if there is no done communication in* $Coms$*, action* a_1 *returns a "false" value. (ii) if there are some done communications in* $Coms$*,* a_1 *needs only the first done communication (the first paired AsyncSend, AsyncReceive) in* $Coms$ *to return "true". Similarly, the effect of action* a_2 *only depends on the state of the mailbox* m*, regardless of the order in which the actions are executed. Furthermore, both actions cannot be disabled by executing the other action. Hence, they are independent. The proof for other pairs of actions can be done similarly.*

Theorem 4.4.6 *Any synchronization action is independent of any communication action of a distinct actor.*

$$\begin{aligned}
&\forall\, act1,\ act2 \in Actors, \forall\ action1 \in \{AsyncMutexLock(act1,\ _), MutexUnlock(act1,\ _),\\
&MutexWait(act1,\ _), MutexTest(act1,\ _,\ _)\}, \forall\ action2 \in \{AsyncSend(act2,\ _,\ _,\ _),\\
&AsyncReceive(act2,\ _,\ _,\ _), TestAny(act2,\ _,\ _), WaitAny(act2,\ _,\ _)\} :\\
&act1 \neq act2 \wedge TypeInv \implies\ I(action1, action2)
\end{aligned}$$

Proof 7 *According to Theorem 4.4.1, they are independent because they concern different variables.*

Theorem 4.4.7 *Any pair of synchronization actions of distinct actors concerning distinct mutexes are independent.*

$$
\begin{aligned}
&\forall\, act1,\, act2 \in Actors,\, mt1,\, mt2 \in MutexIds,\, requests \in \text{SUBSET } MutexIds : \\
&\lor act1 \neq act2 \land mt1 \neq mt2 \land TypeInv \implies \\
&\qquad \land I(AsyncMutexLock(act1,\, mt1),\, AsyncMutexLock(act2,\, mt2)) \\
&\qquad \land I(AsyncMutexLock(act1,\, mt1),\, MutexUnlock(act2,\, mt2)) \\
&\qquad \land I(MutexUnlock(act1,\, mt1),\, MutexUnlock(act2,\, mt2)) \\
&\lor act1 \neq act2, requests \in \text{SUBSET } Requests[act2] \land \neg\exists\, req \in requests :\ req = mt1 \implies \\
&\qquad \land I(AsyncMutexLock(act1,\, mt1),\, MutexTestAny(act2,\, requests,\, \_)) \\
&\qquad \land I(AsyncMutexLock(act1,\, mt1),\, MutexWaitAny(act2,\, requests)) \\
&\qquad \land I(MutexUnlock(act1,\, mt1),\, MutexUnlock(act2,\, mt2)) \\
&\qquad \land I(MutexUnlock(act1,\, mt1),\, MutexWaitAny(act2,\, requests)) \\
&\qquad \land I(MutexUnlock(act1,\, mt1),\, MutexTestAny(act2,\, requests,\, \_))
\end{aligned}
$$

Proof 8 *According to Theorem 4.4.1, they are independent because they concern different variables.*

Theorem 4.4.8 *An AsyncMutexLock is independent with a MutexUnlock of another actor.*

$$
\begin{aligned}
&\forall\, act1,\, act2 \in Actors : act1 \neq act2 \land TypeInv \implies \\
&\qquad I(AsyncMutexLock(act1,\, \_),\, MutexUnlock(act2,\, \_))
\end{aligned}
$$

Proof 9 *Suppose that an action AsyncMutexLock and an action MutexUnlock are executed by actor $act1$ and actor $act2$, respectively. If they relate to different mutexes, they are independent by the Theorem 4.4.1. Conversely, if they concern the same mutex m, we firstly examine the execution order where AsyncMutexLock executes before MutexUnlock. If executing AsyncMutexLock the id of $act1$ is added to $Mutexes[m]$, and the id of m is added to the request set of the actor $act1$. After that, firing MutexUnlock results in removing the id of mutex m and the id of actor $act2$ from the request set of $act2$ and $Mutexes[m]$, respectively. Similarly, with the reverse order, we get the same outcome. Besides, they do not change the conditions making each other become enabled. Thus, they are independent.*

Theorem 4.4.9 *Any pair of actions in {MutexWaitAny, MutexTestAny} of distinct actors is independent.*

$$\begin{aligned}
&\forall\, act1,\ act2 \in Actors : act1 \neq act2 \wedge TypeInv \implies \\
&\quad \wedge I(MutexTestAny(act1,\ _,\ _),\ MutexTestAny(act2,\ _,\ _)) \\
&\quad \wedge I(MutexWaitAny(act1,\ _),\ MutexWaitAny(act2,\ _)) \\
&\quad \wedge I(MutexWaitAny(act1,\ _),\ MutexTestAny(act2,\ _,\ _))
\end{aligned}$$

Proof 10 *Let's first prove the theorem for two MutexWaitAny actions. If both actions are enabled in a particular state, with any order of execution, they only modify the local state of their actors (changing program counters to the next actions), and do not modify any shared data structure (*e.g., *mutexes, mailboxes). So, swapping them does not change their effects. Besides, whether they are enabled or not depends only on the state of the communications they wait on regardless of the execution of the other action. For those reasons they are independent. We can prove it similarly for two MutexTestAny actions, or a MutexWaitAny action and MutexTestAny action.*

Theorem 4.4.10 *Let $firstTwoOwners(m)$ be the set containing first two actors in mutex m. Let $firstExecutedUnlock(M)$ be the first mutex m in M concerned by a MutexUnlock. Consider the action a_1 is a MutexUnlock (m) of actor Act1 and a_2 is a MutexTestAny(M) or MutexWaitAny(M) executed by actor act_2. Then a_1 and a_2 are independent if at least one of the actors act_1 and Act_2 is not in $firstTwoOwners(m)$ for any m in M, or m is not $firstExecutedUnlock(M)$.*

$$\begin{aligned}
&\forall\, act1,\ act2 \in Actors,\ m \in MutexIds,\ requests \in \text{SUBSET } MutexIds : \\
&\quad \wedge act1 \neq act2 \wedge TypeInv \\
&\quad \wedge\ \vee \neg\exists\, m' \in M : m = m' \\
&\qquad \vee act1, act2 \notin firstTwoOwners(m) \\
&\qquad \vee m \neq firstExecutedUnlock(M) \\
&\implies \qquad \wedge I(MutexUnlock(act1,\ m),\ MutexWaitAny(act2,\ M)) \\
&\qquad\qquad\ \ \wedge I(MutexUnlock(act1,\ m),\ MutexTestAny(act2,\ M,\ _))
\end{aligned}$$

In the above expression, we suppose that function $firstTwoOwners(mt)$ returns *true* if the two actors are the first two actors in mutex mt, otherwise it returns *false*. Function $firstExecutedUnlock(M)$ gives the first mutex m in M concerned by a *MutexUnlock*.

Proof 11 *Let's start with a MutexTestAny (M) (called action a_1) and an MutexUnlock (m) (called action a_2). Suppose they are enabled at a particular state.*

- *If they do not concern the same mutex, then according to theorem 4.4.1 they are independent.*
- *If they concern the same mutex, there are two cases:*

(i) If at least one of them is not in $firstTwoOwners(m)$. Executing a_1 returns "true" if the actor of a_1 is the first actor in some mutex in M, otherwise it returns "false". Executing a_2 is removing the actor (actually id of the actor) of action a_2 from $Mutexes[m]$. Changing the execution order, we get the same output because the effects of both actions do not depend on any execution order.

(ii) If they are both in $firstTwoOwners(m)$. If a_2 is not equal to $firstExecutedUnlock(M)$. In any execution order, MutexTestAny (M) always returns a "true" value, since there is some MutexUnlock already executed, resulting in a_1 is the first actor in some mutex in M. If we change the execution order, we get the same overall outcome. From the above reasons, they are independent. For other pairs of actions, we can prove similarly.

Theorem 4.4.11 *An AsyncMutexLock is independent of any MutexWaitAny and MutexTestAny of another actor.*

$$\forall\, act1,\, act2 \in Actors : act1 \neq act2 \wedge TypeInv \implies$$
$$\wedge\, I(AsyncMutexLock(act1, \_),\, MutexWaitAny(act2, \_))$$
$$\wedge\, I(AsyncMutexLock(act1, \_),\, MutexTestAny(act2, \_, \_))$$

Proof 12 *Let's prove for a AsyncMutexLock and a MutexWaitAny. If both actions are enabled, their effects do not depend on the order of execution. Indeed, after executing AsyncMutexLock, the id of the mutex is added to the request set of the actor that fires AsyncMutexLock while the id of the actor is enqueued to the mutex. Besides, firing MutexWaitAny does not change any variables. From the above arguments, we can conclude that they are independent. For the pair of AsyncMutexLock and MutexTestAny, we can prove similarly.*

Theorem 4.4.12 *A LocalComp is independent with any other action of another actor.*

$$\forall\, action1 \in \{AsyncMutexLock(act1, \_), MutexUnlock(act1, \_), MutexWait(act1, \_),$$
$$MutexTestAny(act1, \_, \_), AsyncSend(act1, \_, \_, \_), AsyncReceive(act1, \_, \_, \_),$$
$$TestAnyAny(act1, \_, \_), WaitAny(act1, \_, \_), localComp(act1)\},$$
$$\forall\, action2 \in \{LocalComp(act2)\} : act1 \neq act2 \wedge TypeInv$$
$$\implies I(action1,\, action2)$$

Proof 13 *According to Theorem 4.4.1 they must be independent because they concern different variables.*

4.5 Encoding MPI programs

In this section, we will introduce some information about MPI as well as some of its basic functions. Then the way we encode MPI programs through the abstract model will also be presented briefly.

4.5.1 Introduction to MPI programs

The Message Passing Interface Standard (MPI) [15] is a standard message passing library defining the syntax and semantics of core routines that will be used to write message-passing programs in C, C++, and Fortran. The first version (version 1.0) of MPI was released in June 1994 after some modifications of a draft MPI standard. Currently, MPI has version 3.1, there has been many changes compared to the first version along with having many functions formally specified. There are several library implementations of MPI, for example, MPICH, Intel MPI, and OpenMPI.

Many high-performance computing applications have been implemented by using MPI libraries. By providing a variety of useful functions and efficient ways for writing distributed programs, they may still be the first choice for HPC application developers in the coming years. Figure 4.16 displays an MPI program written in C. Each MPI program comprises some autonomous processes deployed on different computers. It inherits the properties of distributed applications such as processes communicating with others by exchanging messages via calling MPI communication functions. For example, the function MPI_Send is used to send a message from a particular process to a destination process while receiving a message can be done by calling a function $MPI\_Receive$. To increase performance as well as flexibility, MPI provides a range of communication modes, including point-to-point communications, collective communications, one-sided communications. Which function to use depends on the computation requirements as well as users experience. The remainder of the section introduces some point-to-point communication primitives that are closely similar to the primitives in our abstract model of distributed programs.

Point-to-Point Communication Point-to-point communication is a fundamental form of message-passing communication. Two nodes are involved: one sends a message to the other. There are several MPI functions defined in this mode, and they can be divided

```
#include "mpi.h"
#include <stdio.h>
int main( int argc, char *argv[] )
{
int rank, world_size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &world_size );
printf( "Hello, i am %d of %d \n", rank, world_size );
MPI_Finalize();
return 0;
}
```

Figure 4.16 – A simple MPI program

into two groups: blocking and non-blocking. A process may wait for a blocking function until success while it will not wait to complete a non-blocking function, executing the next action immediately.

— **MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)**: it is a non-blocking function for sending data where *buf* is the initial address of the data buffer that will be transferred to the destination node, *count* (an integer) is the number of elements in send buffer, *datatype* is the datatype of the data in buf, *dest* is the rank of destination node, *tag* is the message tag, and *comm* is the MPI communicator, finally *request* is the communication request.

— **MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)**: a non-blocking function for receiving data where *buf* is the initial address of the memory where the receiving data will be stored, and *source* is the rank of the source process sending the data that will be received, *count* is the maximum number of elements in receive buffer. *MPI_Irecv* function can receive data from any source process by setting *source* to **MPI_ANY_SOURCE**. Other parameters have the same semantics as described for *MPI_Isend*.

— **MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**: similar to *MPI_Isend* , it is used to send data, but while *MPI_Isend* is a non-blocking function, *MPI_Send* is a blocking one. The param-

eters in MPI_Send have the same semantics as presented for *MPI_Isend*. Note that here buf does not refer to the runtime buffer in the next section (a motivating example).

— **MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)**: a blocking function where *status* stores information about the receive operation after it completes. Other parameters have the same semantics as described in *MPI_Irecv*.

To check the status of a non-blocking *MPI_Isend* and *MPI_Irecv*, one can use *MPI_Test* or *MPI_Testany*.

— **MPI_Test(MPI_Request * request, int *flag, MPI_Status * status)** : MPI_TEST returns *flag = true* if the operation identified by *request* is completed. In that case, the information on the completed operation is stored in the *status* object. Otherwise, the function returns *flag = false*, and the value of the status object is undefined.

— **MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status)**: tests an array of requests (array_of_requests) with *size = count*. If at least one request is completed then *flag* is set to true and returns in *index* the index of the completed request, otherwise *flag* is set to false.

To wait for a blocking *MPI_Isend* and *MPI_Irecv*, one can use *MPI_Wait*, *MPI_Waitany*, or *MPI_Waitall*.

— **MPI_Wait(MPI_Request *req, MPI_Status *status)**: waits for the completion of the request identified by the parameter *req*. An instance of MPI_Status that keeps information of the request is returned in *status* as soon as the request *req* is completed.

— **MPI_Waitany(int count, MPI_Request array_of_requests, int *index, MPI_Status *status)** : waits for an array of requests (array_of_requests). The action is blocked until at least one communication in the array is completed. Parameter *index* stores the index of the first completed request in *array_of_requests* while *status* keeps the status of the completed request.

— **MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status array_of_statuses[])**: waits for an array of requests (array_of_requests). The action is blocked until all communications in the array is completed.

4.5.2 Encoding

We aim at leveraging UDPOR while verifying MPI programs. Since our computational model is more abstract than MPI, this section presents how MPI primitives can be encoded in our model.

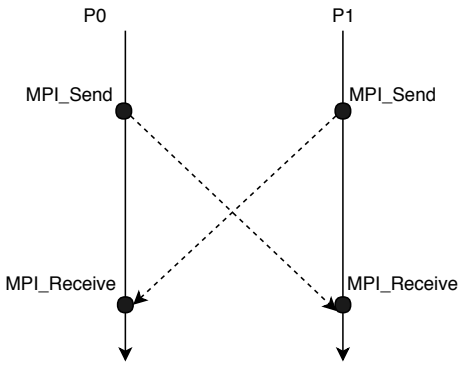


Figure 4.17 – An MPI program with a potential deadlock

A motivating example We begin with a simple example shown in Figure 4.17. Suppose we have an MPI program consisting of two processes $P0$ and $P1$. Both processes try to send a message to the other by executing a MPI_Send function. After performing sending functions, they both execute MPI_Recv to receive the message from the other. At first glance, we may think that running that program can encounter a deadlock since there is a dependency cycle (MPI_send of the first process waits for MPI_Recv of the second one while MPI_Send of the second process waits for the MPI_Recv of the first one). Interestingly, depending on presence or absence of buffering (finite or zero buffering) in MPI nodes, the deadlock may or may not appear.

The semantics of MPI_Send functions are ambiguously defined. A MPI_Send is an asynchronous call under zero buffering (messages have no buffering) while under infinite buffering (messages are copied into a runtime buffer) it does not wait for a matching MPI_Recv to be completed. Therefore, the scenario in Figure 4.17 may or may not have a deadlock. Our programming model can encode MPI functions in the both infinite buffering and zero buffering semantics.

Encoding MPI programs Based on the semantics of MPI functions and the abstract model, we can represent MPI functions by using the primitives of the abstract model.

Table 4.18 presents the correspondence between some MPI functions and the actions in the abstract model. There is a minor difference between the two modes in the table. While in infinite buffering mode, an MPI_Send is represented by an *AsyncSend* action, it is encoded by a pair composed of an *AsyncSend* action and a *WaitAny* action in the zero buffering mode.

MPI functions	**Infinite buffering**	**Zero buffering**
MPI_Send	*AsyncSend*	*AsyncSend* + *WaitAny*
MPI_Isend		*AsyncSend*
MPI_Recv	*AsyncReceive* + *WaitAny*	*AsyncReceive* + *WaitAny*
MPI_Irecv	*AsyncReceive*	*AsyncReceive*
MPI_Test	*TestAny*	*TestAny*
MPI_Testany		
MPI_Wait	*WaitAny*	*WaitAny*
MPI_Waitany		
MPI_Win_lock	*AsyncMutexLock* + *MutexWaitAny*	*AsyncMutexLock* + *MutexWaitAny*
MPI_Win_unlock	*MutexUnlock*	*MutexUnlock*

Figure 4.18 – Representations of some basic MPI functions in the two modes.

Actor0
c = AsyncSend(mb1)
c1 = AsyncReceive(mb0)
WaitAny({c1})

Actor1
c2 = AsyncSend(mb0)
c3 = AsyncReceive(mb1)
WaitAny({c3})

Actor0
c = AsyncSend(mb1)
WaitAny({c})
c1 = AsyncReceive(mb0)
WaitAny({c1})

Actor1
c2 = AsyncSend(mb0)
WaitAny({c2})
c3 = AsyncReceive(mb1)
WaitAny({c3})

Figure 4.19 – Encoding of a MPI program in two modes infinite and zero buffering.

We now easily encode the MPI program in Figure 4.17. Each process in the program is represented by an actor. There are two mailboxes $mb0$ and $mb1$. Actor0 posts a send communication to $mb1$ and a receive one to $mb0$. Meanwhile in the opposite direction, Actor1 sends a communication to $mb0$ and a receive one to $mb1$. Figure 4.19 demonstrates the encoding of the MPI program in the two modes (zero buffering mode in the right and infinite buffering mode in the left). There is only one difference between the two modes related to the encoding of MPI_Send function. Obviously, the zero buffering mode version encounters a deadlock since both actors are blocked; the communications they wait for can not become done communications because there are no matching communications for them to pair with in the mailboxes.

P0	**Actor0**
MPI_Irecv (from P1, &reqs[0]);	c = AsyncReceive (mb0);
MPI_Irecv (from P1, &reqs[1]);	c1 = AsyncReceive (mb0);
MPI_Waitall (2, reqs);	WaitAny({ c });
MPI_Send (to P1);	WaitAny({ c1 });
	c3 = AsyncSend (mb1);
	WaitAny({ c3 });
P1	**Actor1**
MPI_Isend (to P0, &reqs[0]);	c4 = AsyncSend (mb0);
MPI_Isend (to P2, &reqs[1]);	c5 = AsyncSend (mb2);
MPI_Waitall (2, reqs, statuses);	WaitAny({ c4 });
MPI_Recv (from P0);	WaitAny({ c5 });
MPI_Recv (from P0);	c6 = AsyncReceive (mb1);
	WaitAny(c6);
	c7 = AsyncReceive (mb1);
	WaitAny({ c7 });
P2	**Actor2** c8 = AsyncReceive (mb2);
MPI_Recv (from P1);	WaitAny({ c8 });

Figure 4.20 – An MPI program (in the left) and its encoding (in the right).

Consider now a more complex program shown in the left of the Figure 4.20. In

the zero buffering mode, with any executing orders, the program encounters deadlocks because $P0$ and $P1$ wait for each other. The encoding of that program (in zero buffering mode) is shown in the right of Figure 4.20. In the encoding, the $MPI\_Waitall$ function is simulated by using two $WaitAny$ actions. In the general case, a function $MPI\_Waitall$ that waits a set of n requests can be simulated by n $WaitAny$ actions where each action waits for an individual request. The corresponding actions encode other MPI functions in the program.

Simple MPI programs can be easily encoded through the abstract model. However, encoding complex programs with many types of MPI functions becomes more complicated. For instance, encoding an MPI function may need a block of actions in the abstract model, or creating mailboxes for communications should be efficient. The primitives of our abstract model closely match the ones provided by SimGrid's simulation kernel. SimGrid provides a programming environment for the simulation of MPI applications, namely SMPI. SMPI runs MPI applications on top of that kernel thanks to a re-implementation of the MPI primitives on top of the simulation kernel and its primitives. So, it can simulate 160 MPI functions [13]. Besides, 60 MPI proxy apps that are considered as representatives of massive HPC applications are supported by the SMPI implementation [4].

4.6 Conclusion

In this chapter we presented our abstract model of asynchronous distributed programs. From the formal point of view, a significant advantage of the abstract model is that it can cover a large class of MPI applications based on a very small amount of kernel primitives. After defining the abstract model, TLA+ is used to specify the abstract model. Independence theorems between the actions in the model are also formalized in TLA+ and informally proved. The theorems will be used as rules to compute independence between actions in concrete MPI programs in model checking. Besides, specifying the abstract model, the chapter also demonstrates an important property of the model, namely persistence. All the actions in the abstract model are persistent. In the next chapter, we will see that persistence is essential in the efficiency of UDPOR.

4. https://framagit.org/simgrid/SMPI-proxy-apps

CHAPTER 5

ADAPTING UDPOR

Recall that the behavior of a concurrent program can be represented compactly by using its unfolding semantics (section 3.2). UDPOR operates over the unfolding semantics and explores a reduced LTS without redundant explorations of Mazurkiewicz traces.

Algorithm 3: Unfolding-based POR exploration

```
1  Set U := ∅
2  call Explore(∅, ∅, ∅)
3  Procedure Explore(C, D, A)
4      Compute ex(C), and add all events in ex(C) to U
5      if en(C) ⊆ D then
6          Return
7      if (A = ∅) then
8          chose e from en(C) \ D
9      else
10         choose e from A ∩ en(C)
11     Explore(C ∪ {e}, D, A \ {e})
12     if ∃J ∈ Alt(C, D ∪ {e}) then
13         Explore(C, D ∪ {e}, J \ C)
14     U := U ∩ Q_{C,D}
```

In order to get optimality (*i.e.* exploring exactly one interleaving per each Mazurkiewicz trace), the function $Alt(C, D \cup \{e\})$ has to solve an NP-complete problem. Besides, computing the extensions $ex(C)$ of a configuration C may also be costly in general. It is, for example, an NP-complete problem for Petri Nets since all sub-configurations must be enumerated. Fortunately, this algorithm can be specially tuned for sub-classes. In particular for the programming model of [34] it is tuned in an algorithm working in

time $O(n^2 log(n))$, using the fact that events have a maximum of two causal predecessors, thus limiting the subsets to consider. This chapter tunes the algorithm to our abstract model, using the fact that the amount of causal predecessors of events is also bounded. Besides, the chapter also shows how to incrementally compute $ex(C)$ to avoid recomputations.

5.1 Computing extensions efficiently

This section mandates some additional notations. Given a configuration C and an extension with action a, let $pre(a)$ denote the action right before a in the same actor, while $preEvt(a, C)$ denotes the event in C associated with $pre(a)$ (formally $e = preEvt(a, C) \iff e \in C, \lambda(e) = pre(a)$). Given a set of events $F \subseteq E$, $Depend(a, F)$ means that a depends on all actions labeling events in F.

The definition of $ex(C)$ (set of extensions of a configuration C)

$$ex(C) = \{e \in E \setminus C : \lceil e \rceil \subseteq C\} \tag{5.1}$$

can be rewritten using the definitions of section 3.1.3 as follows:

$$ex(C) = \{e = \langle a, H \rangle \in E \setminus C : a = \lambda(e) \wedge H = \lceil e \rceil \wedge H \in 2^C \cap conf(E) \wedge a \in enab(H)\}. \tag{5.2}$$

Fortunately, it is not necessary to enumerate all subsets H of C, that are in exponential number, to compute this set. According to the unfolding construction in Definition 6, an event $e = \langle a, H \rangle$ only exists in $ex(C)$ if the action a is dependent on the actions of all maximal events of H. This gives:

$$\begin{aligned} ex(C) = \{e = \langle a, H \rangle \in E \setminus C : a = \lambda(e) \wedge H = \lceil e \rceil \wedge H \in 2^C \cap conf(E) \\ \wedge a \in enab(H) \wedge Depend(a, maxEvents(H))\}. \end{aligned} \tag{5.3}$$

Now $ex(C)$ can be simplified and decomposed by enumerating Σ, yielding to:

$$ex(C) = \bigcup_{a \in \Sigma} \{\langle a, H\rangle : H \in S_{a,C}\} \setminus C$$
$$where\ S_{a,C} = \{H \in conf(E) : H \subseteq C \wedge a \in enab(H) \wedge Depend(a, maxEvents(H))\}. \tag{5.4}$$

The above formulation of $ex(C)$ iterates on all actions in Σ.

However, interpreting Lemma 1 on page 64 (an enabled action cannot be disabled by executing any other action) for configurations H and C with $H \subseteq C$ entails that an action is enabled at $state(H)$ can not be disabled by executing any action in C. Thus, for two configurations H and C with $H \subseteq C$, an action a in $enab(H)$ is either in $actions(C)$ or in $enab(C)$.

Therefore, $ex(C)$ can be rewritten by restricting a to $actions(C) \cup enab(C)$:

$$ex(C) = (\bigcup_{a \in actions(C) \cup enab(C)} \{\langle a, H\rangle : H \in S_{a,C}\}) \setminus C \tag{5.5}$$

Now, since a configuration is uniquely characterized by its set of maximal events, instead of enumerating possible configurations $H \in S_{a,C}$, we can enumerate maximal sets $K = maxEvents(H)$. Hence, $ex(C)$ can be written as

$$ex(C) = (\bigcup_{a \in actions(C) \cup enab(C)} \{\langle a, config(K)\rangle : K \in S^{max}_{a,C}\}) \setminus C \tag{5.6}$$

with $S^{max}_{a,C} = \{K \in 2^C :$ K is maximal $\wedge\ a \in enab(config(K)) \wedge Depend(a, K))\}$ and K is maximal if $(\nexists e, e' \in K : e < e')$.

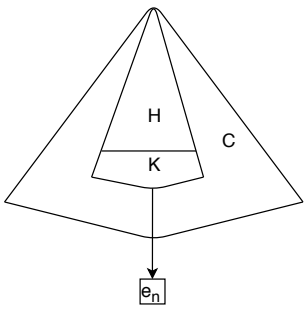


Figure 5.1 – A configuration C, extended by a new event e_n, with history H of maximal events K.

One can then specialize the computation of $ex(C)$ according to the type of action a. The remainder of this section now illustrates how to compute such extensions according to the type of action a. We will see that an event can only depend on very few and easily identifiable maximal events, and can thus be implemented efficiently.

5.1.1 General properties

Given a configuration C, we have some general properties that will be used for computing extensions as follows:

+ All *AsyncSend*$(m, \_)$ events in C are causally related. Indeed, because their actions are dependent (two *AsyncSend* actions that concern the same mailbox are dependent), those events cannot be concurrent. Besides, they are in the same configuration, they cannot conflict. For those reasons, they are causally related.
+ Similarly, we have that all *AsyncReceive*$(m, \_)$ are causally related.
+ All *AsyncMutexLock*(mtx) events in C are causally related. Because their actions are dependent (they concern the same mutex mtx), they cannot be concurrent. Besides, since they are in the same configuration, they cannot conflict. Thus, they are causally related.

5.1.2 Computing extensions for *AsyncSend* actions.

Let C be a configuration, and a an action of type $c =$ *AsyncSend*$(m, \_)$ of an actor A_i. We want to compute the set $S^{max}_{a,C}$ of sets K of maximal events from which a depends.

According to independence theorems (see theorem 4.4.2, theorem 4.4.3, theorem 4.4.5, theorem 4.4.6, and theorem 4.4.12), a can only depends on the following actions: $pre(a)$, all *AsyncSend*$(m, \_)$ actions of distinct actors A_j which concern the same mailbox m, and all *WaitAny* (resp. *TestAny*) actions that wait (resp. test) a *AsyncReceive* which concerns the same communication c. Considering this, we now examine the composition of maximal events sets K in $S^{max}_{a,C}$, and prove that its cardinality is bounded by 3.

First, according to the general properties, two events labelled by *AsyncSend*$(m, \_)$ actions cannot co-exist in K (they are causally related), formally $\nexists e, e' \in K : \lambda(e), \lambda(e')$ are *AsyncSend*$(m, \_)$ (see Example 6 for illustration).

Second, if a *WaitAny*(Com) action concerns communication c, there are two cases: (i) either c is not the first *done* communication in Com, then *WaitAny*(Com) and the action a are independent; or (ii) c is the first *done* communication in Com and *WaitAny* is enabled only after a (see Example 7 for illustration). Thus the only possibility for a maximal event to be labelled by a *WaitAny* is when $pre(a)$ is a *WaitAny* of the same actor. We can then write: $\nexists e \in K : \lambda(e)$ is *WaitAny* $\wedge \lambda(e) \neq pre(a)$.

Third, all *AsyncReceive* events for the mailbox m are causally related in configuration C, and c can only be paired with one of them, say c'. Thus a can only depend on actions *TestAny*(Com') such that $c' \in Com'$ and c and c' form the first *done* communication in Com', and all those *TestAny* events are ordered. Thus, there is at most one event e labelled by *TestAny* in K such that $\lambda(e) \neq pre(a)$ (see Example 8 for illustration).

To conclude, K contains at most three events: $preEvt(a, C)$, an event labelled with an *AsyncSend* action on the same mailbox, and a *TestAny* for some matching *AsyncReceive* communication. There is thus only a cubic number of such sets, which is the worse case among considered action types as will be seen in the sequel.

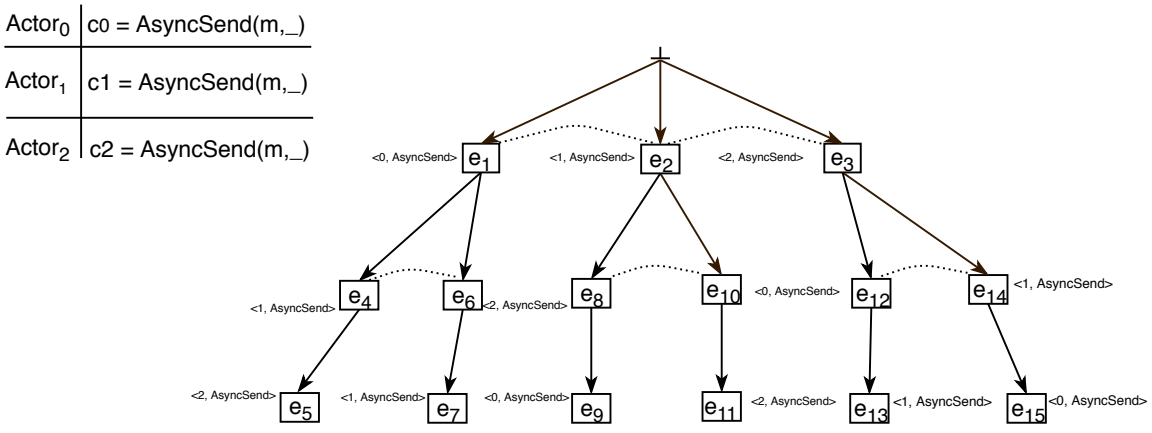


Figure 5.2 – The pseudo-code of a distributed program and its unfolding

Example 6 *Figure 5.2 presents the pseudo-code of a distributed program and its unfolding. The program consists of three actors, and each actor sends an AsyncSend request to mailbox m. Altogether the unfolding includes 15 AsyncSend events, but*

each event has at most one successor which is an AsyncSend event. Besides, if two AsyncSend events are not causally related, they will conflict. For example, event e_4 *and event* e_6 *are not in the same configuration, and they conflict; event* e_6 *and event* e_8 *belong to different configurations and also conflict. All AsyncSend events in the same configuration are causally related. For example, events* e_1*,* e_4*, and* e_5 *are in the same configuration and are causally related. Similarly, events* e_2*,* e_8*, and* e_9 *belong to the same configuration and are causally related.*

Example 7 *Let's study the distributed program and its unfolding in Figure 5.3 to support the claim that a WaitAny event cannot be a direct ancestor of an AsyncSend event of another actor. In the program, the action WaitAny of the first actor depends on the first AsyncReceive of the second actor since they concern the same first done communication (obtained by pairing* c_0 *and* c_1*). Obviously* c_0 *cannot be "done" without* c_1*. Hence, the action WaitAny is only enabled after the first AsyncReceive.*

We are now considering the WaitAny action of the second actor. Although the WaitAny action and the second AsyncSend action of the first actor concern the same communication, but they are independent because they do not concern the first done communication in the set $\{c_1, c_1'\}$*. The first done communication in* $\{c_1, c_1'\}$ *is the communication created by the combination between* c_0 *and* c_1*. Therefore, the first AsyncSend of the* $Actor_0$ *is dependent on such a WaitAny action. However, without executing the first AsyncSend of* $Actor_0$*, such a done communication cannot be created. That is why the action WaitAny can only be enabled after the action AsyncSend. And we can see that in the unfolding there is no AsyncSend whose direct ancestor is a WaitAny event.*

$Actor_0$: c_0 = `AsyncSend(m,_)`
c_0'= `AsyncSend(m,_)`
`WaitAny({`c_0`})`

$Actor_1$: c_1= `AsyncReceive(m,_)`
c_1'= `AsyncReceive(m,_)`
`WaitAny({`c_1, c_1'`})`

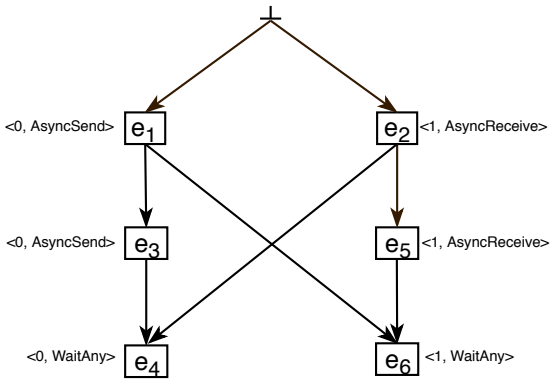


Figure 5.3 – The pseudo-code of a distributed program and its unfolding.

Example 8 *This example supports the statement: an AsyncSend event has at most one direct TestAny predecessor that belongs to another actor. Indeed, in Figure 5.4, the unfolding of a distributed program has several AsyncSend events and TestAny events, but all the AsyncSend events satisfy the condition.*

There are four maximal configurations: $C_1 = \{e_1, e_2, e_4, e_{10}, e_{11}\}$, $C_2 = \{e_1, e_4, e_5, e_6, e_8\}$, $C_3 = \{e_1, e_4, e_5, e_7, e_9\}$, *and* $C_4 = \{e_2, e_3, e_{12}, e_{13}, e_{14}\}$. *In all the maximal configurations, the action AsyncSend only concerns one communication. For example, in configuration* C_1, *it only combines with AsyncReceive of the first actor to form a done communication. Similarly, in the configuration* C_4 *the AsyncSend action and the AsyncReceive of* $Actor_2$ *build another done communication. Besides, in different configurations, the AsyncSend action is dependent on different TestAny actions, but the statement is still satisfied. Indeed, in configurations* C_2 *and* C_3, *the AsyncSend action is dependent on both TestAny actions of* $Actor_0$, *but each AsyncSend event (e.g.,* e_6*) has at most one TestAny event in its predecessors.*

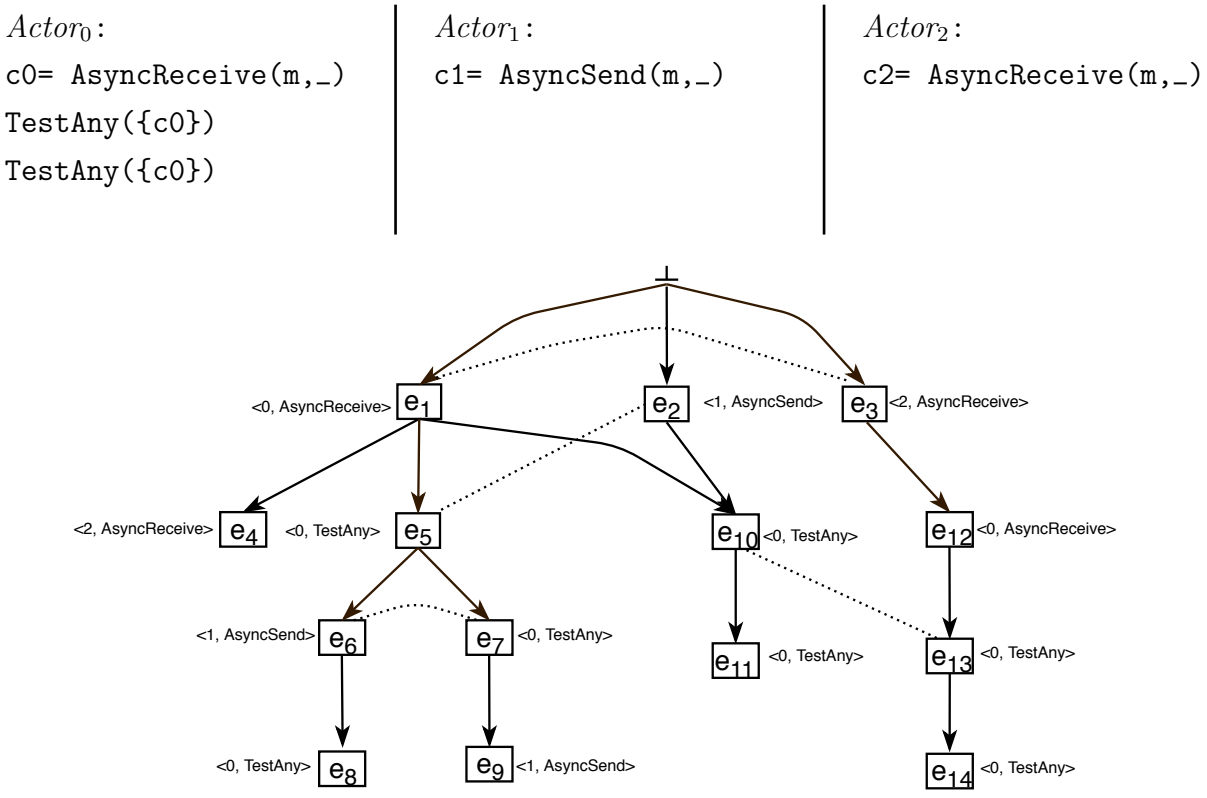


Figure 5.4 – The pseudo-code of a distributed program and its unfolding.

Algorithm 4: createAsyncSendEvt(a, C)

1 create e' : = $\langle a, config(preEvt(a, C))\rangle$, and $ex(C) := ex(C) \cup \{e'\}$
2 **foreach** $e \in C$ *s.t.* $\lambda(e) \in \{$*AsyncSend*$(m, \_)$, *TestAny*$(Com)\}$
3 *where* Com *contains a matching* $c' =$ *AsyncReceive*$(m, \_)$ *with* a **do**
4 $K := \emptyset$;
5 **if** $\neg(e < preEvt(a, C)$ **then** $K := K \cup \{e\}$;
6 **if** $\neg(preEvt(a, C) < e)$ **then** $K := K \cup \{preEvt(a, C)\}$;
7 **if** $D_K(a, \lambda(e))$ **then** create e' = $\langle a, config(K)\rangle$, and $ex(C) := ex(C) \cup \{e'\}$;
8 **foreach** $e_s \in C$ *s.t.* $\lambda(e_s) =$ *AsyncSend*$(m, \_)$ **do**
9 **foreach** $e_t \in C$ *s.t.* $\lambda(e_t) =$ *TestAny*(Com)
10 *where* Com *contains a matching* $c' =$ *AsyncReceive*$(m, \_)$ *with* a **do**
11 $K := \emptyset$;
12 **if** $\neg(e_s < preEvt(a, C))$ *and* $\neg(e_s < e_t)$ **then** $K := K \cup \{e_s\}$;
13 **if** $\neg(e_t < preEvt(a, C)$ *and* $\neg(e_t < e_s)$ **then** $K := K \cup \{e_t\}$;
14 **if** $\neg(preEvt(a, C) < e_s)$ *and* $\neg(preEvt(a, C) < e_t)$ **then**
$K := K \cup \{preEvt(a, C)\}$;
15 **if** $D_K(a, \lambda(e_t))$ **then** create e' : = $\langle a, config(K)\rangle$, and
$ex(C) := ex(C) \cup \{e'\}$;

Remark 1 *Note that in the above algorithm we use the notion* D_K *since in our model, dependency can be context sensitive (i.e. depend on the configurations, see below in section 5.3). Thus,* $D_K(a, b)$ *denotes the fact that actions* a *and* b *are dependent in the configuration* $config(K)$*.*

Algorithm 4 generates all events in $ex(C)$ labelled by an *AsyncSend* action a. It first creates a new event e' labelled by the action a (line 1), and event $preEvt(a, C)$ is the unique predecessor of that event. Since a is dependent on $pre(a)$ and if a is an *AsyncSend* action, it becomes enabled after executing $pre(a)$. So that is why e' is created. Next the algorithm iterates on all events e in $\{$*AsyncSend*$(m, \_)$, *TestAny*$(Com)\}$ (line 2), then building set K. Line 5 and line 6 ensure that K includes only concurrent events. If a is dependent on the action of e, a new event is created. To simplify the presentation we always check the dependence between a and $\lambda(e)$ (line 7), but if e is *AsyncSend*$(m, \_)$ the condition is always satisfied. The rest of the algorithm (from line 9) is mainly dedicated to create new events with three direct predecessors. Firstly the

candidate K is built, and ensuring that K is composed of only concurrent events (line 12 to line 15). Then if there is a dependence between a and $\lambda(e_t)$ then a new event e' is created. The algorithm only checks the dependence between a and $\lambda(e_t)$ because for other events in K (*i.e.* $preEvt(a, C)$ and *AsyncSend*$(m, \_)$ events) their actions are always dependent on a.

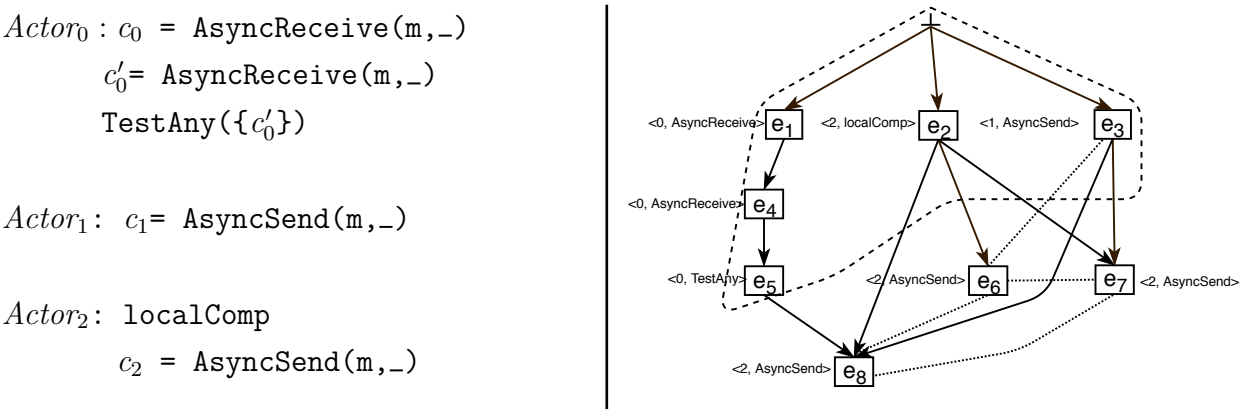


Figure 5.5 – A program, and a configuration C with extensions by action AsyncSend.

Example 9 *We illustrate the Algorithm 4 by the example shown in Figure 5.5. Suppose we want to compute the extensions of* C *=* $\{e_1, e_2, e_3, e_4, e_5\}$ *associated with* a*, the action* $c2 =$ *AsyncSend*$(m, \_)$ *of* $Actor_2$*. First* $e_6 = \langle$*AsyncSend*$, \{e_2\}\rangle \in ex(C)$ *because* $preEvt(a, C) = e_2$*. We then iterate on all AsyncSend events in* C*, combining them with* e_2 *to create the maximal event set* K *(lines 2-8 in the algorithm). We only have one AsyncSend event* e_3*. Since* $\neg(e_2 < e_3)$ *and* $\neg(e_3 < e_2)$*, we form a first set* $K = \{e_2, e_3\}$*, and then create* $e_7 = \langle$*AsyncSend*$, \{e_2, e_3\}\rangle$*. Next, all TestAny events that concern the mailbox* m *should be considered. Events* e_2 *and* e_5 *can be combined to form a new maximal event set* $K = \{e_2, e_5\}$*, but since* a *and* $\lambda(e_5)$ *are not related to the same communication,* $D(a, \lambda(e_5))$ *is not satisfied and no event is created. Finally, combinations of* e_2 *with an AsyncSend event and a TestAny event are examined (lines 9-17 in the algorithm). We then get* $K = \{e_2, e_5, e_3\}$*, and* e_8 *is added to* $ex(C)$ *since* $D(a, \lambda(e_5))$ *holds in the configuration* $config(\{e_2, e_5, e_3\})$*.*

5.1.3 Computing extensions for *AsyncReceive* actions.

Since an *AsyncReceive* action and an *AsyncSend* action are completely symmetrical, thus we can also conclude that K contains at most three events: $preEvt(a, C)$, and an event labelled with an action *AsyncReceive* on the same mailbox, and a *TestAny* for a particular matching *AsyncSend* communication. Thus, similar to computing extensions for *AsyncSend* actions, only a cubic number of such sets exist which is the worse case among considered action types.

Algorithm 5: createAsyncReceiveEvt(a, C)

create $e' := < a, config(preEvt(a, C)) >$, and $ex(C) := ex(C) \cup \{e'\}$
foreach $e \in C$ *s.t.* $\lambda(e) \in \{$*AsyncReceive*$(m, \_)$, *TestAny*$(Com)\}$
where Com *contains a matching* $c' =$ *AsyncSend*$(m, \_)$ *with* a **do**
    $K := \emptyset$;
    if $\neg(e < preEvt(a, C)$ **then** $K := K \cup \{e\}$;
    if $\neg(preEvt(a, C) < e)$ **then** $K := K \cup \{preEvt(a, C)\}$;
    if $D_K(a, \lambda(e))$ **then** create $e' = \langle a, config(K)\rangle$ and $ex(C) := ex(C) \cup \{e'\}$;
foreach $e_r \in C$ *s.t.* $\lambda(e_r) =$ *AsyncReceive*$(m, \_)$ **do**
    foreach $e_t \in C$ *s.t.* $\lambda(e_t) =$ *TestAny*(Com)
    where Com *contains a matching* $c' =$ *AsyncSend*$(m, \_)$ *with* a **do**
        $K := \emptyset$;
        if $\neg(e_r < preEvt(a, C))$ *and* $\neg(e_r < e_t)$ **then** $K := K \cup \{e_r\}$;
        if $\neg(e_t < preEvt(a, C)$*and* $\neg(e_t < e_r)$ **then** $K := K \cup \{e_t\}$;
        if $\neg(preEvt(a, C) < e_r)$ *and* $\neg(preEvt(a, C) < e_t)$ **then**
          $K := K \cup \{preEvt(a, C)\}$;
        if $D_K(a, \lambda(e_t))$ **then** create $e' = \langle a, config(K)\rangle$, and
          $ex(C) := ex(C) \cup \{e'\}$;

Algorithm 5 generates all events in $ex(C)$ labelled by an *AsyncReceive* action a. It has the same structure as the previous algorithm. The main difference is that while the former algorithm iterates on *AsyncSend* events the later one iterates on *AsyncReceive* events.

5.1.4 Computing extensions for *WaitAny* actions.

Given a configuration C, and a an action of type *WaitAny*(Com) of an actor A_i. This section presents how to compute the set $S^{max}_{a,C}$ of sets K of maximal events from which a depends.

According to independence theorems (see theorem 4.4.5, theorem 4.4.6, and theorem 4.4.12), a can only depend on the following actions: $pre(a)$, *AsyncSend*$(m, \_)$ actions or *AsyncReceive*$(m, \_)$ actions of a distinct actor A_j which concern the first done communication in Com. We now examine the composition of maximal events sets K in $S^{max}_{a,C}$.

Suppose $c =$ *AsyncSend*$(m, \_)$ is the first done communication in Com, and is paired with $c' =$ *AsyncReceive*$(m, \_)$ of some actor A_j. Thus, action a and the action *AsyncReceive*$(m, \_)$ are dependent. Besides, c becomes done after pairing with c', and the action *WaitAny*(Com), a blocking action, becomes enabled after the pairing. In addition, according to the general properties (section 5.1.1), all *AsyncSend*$(m, \_)$ events in C are causally related, and all *AsyncReceive*$(m, \_)$ events in C are also causally related, thus there exists only one order to pair *AsyncSend*$(m, \_)$ events with *AsyncReceive*$(m, \_)$ events to form done communications in the configuration C. So there is a unique $c' =$ *AsyncReceive*$(m, \_)$ in C that can be paired with c. For those reasons, there is at most one event e labelled by an *AsyncReceive*$(m, \_)$ action in K such that $\lambda(e) \neq pre(a)$. Similarly if $c =$ *AsyncReceive*$(m, \_)$ then there is at most one event e labelled by *AsyncSend*$(m, \_)$ in K such that $\lambda(e) \neq pre(a)$.

To conclude, K contains at most two events: $preEvt(a, C)$, an event labelled with an *AsyncSend* or *AsyncReceive* that concerns the first done communication in Com.

Algorithm 6 simply computes all events associated with action a. If a is enabled at $state(config(\{preEvt(a, C)\}))$ then it creates a new event characterized by a and $config(\{preEvt(a, C)\})$. After that, it iterates on all events e in the configuration C. If e is an *AsyncReceive*$(m, \_)$ or an *AsyncSend*$(m, \_)$ event, then e can possibly be a direct ancestor of a new event labelled by a. If e and $preEvt(a, C)$ are concurrent, both events are added to a set K. Otherwise, one of them, the event that is not an ancestor of the other, is added to K (line 6 to line 7 of the algorithm). If there is a dependence between a and the action of e (*i.e.* they concern the first done communication), then a new event e' is created.

Algorithm 6: $createWaitEvt(a, C)$

```
1  if a is enabled at state(config({preEvt(a, C)})) then
2  |  create e' := ⟨a, config({preEvt(a, C)})⟩; ex(C) := ex(C) ∪ {e'};
3  foreach event e in C do
4  |  if λ(e) is AsyncReceive(m, _) or AsyncSend(m, _) then
5  |  |  K := ∅;
6  |  |  if ¬(e < preEvt(a, C)) then K := K ∪ {e};
7  |  |  if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)};
8  |  |  if D_K(a, λ(e)) then
9  |  |  |  create e' := ⟨a, config(K)⟩;
10 |  |  |  ex(C) := ex(C) ∪ {e'};
```

Example 10 *Let us look at the example presented in Figure 5.6. The distributed program consists of two actors, the first one executes an AsyncSend*$(m, \_)$*, a* $localComp$ *action and a WaitAny action, the second one only performs an AsyncReceive*$(m, \_)$ *action. Suppose we have to compute extensions for the WaitAny action in* $ex(C)$ *with* C *= {*e_1, e_2, e_3*}. Searching all AsyncReceive events in* C*, we only find* e_2*. Since* $\neg(e_2 < e_3)$*, we add* e_2 *to the set* K*. Besides, we have* $\neg(e_3 < e_2)$*, thus* e_3 *is added to* K*, forming* $K = \{e_2, e_3\}$*. We can then create* $e_4 = \langle$*WaitAny*$, \{e_2, e_3\}\rangle$ *because* $D(a, \lambda(e_2))$*.*

$Actor_0 : c_0$= *AsyncSend*$(m, \_)$
`localComp`
WaitAny$(\{c\})$

$Actor_1 : c_1$= *AsyncReceive*$(m, \_)$

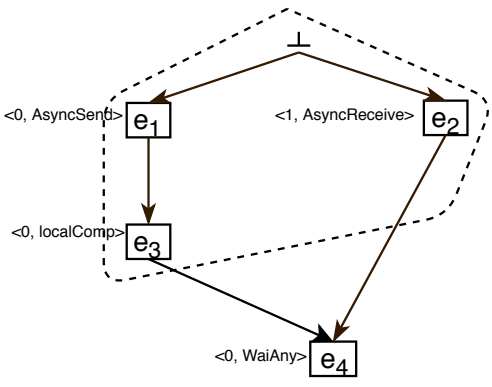


Figure 5.6 – A program and a configuration C with extensions by action WaitAny.

5.1.5 Computing extensions for *TestAny* actions.

There is only one difference between a *TestAny* action and a *WaitAny* action, that is, while the *TestAny* action is a non-blocking action (*i.e.,* its enableness depends only on its previous action), the *WaitAny* action is blocking one (*i.e.,* its enableness depends not only on its previous action but also the status of communications that it waits for). Thus, we can use almost the arguments as presented in computing extensions for *WaitAny* actions. So, given a configuration C, and a an action of type *TestAny*(Com) of an actor A_i. We can conclude that K contains at most two events: $preEvt(a, C)$, and an event labelled with an action *AsyncSend* or *AsyncReceive* that concerns the first done communication in Com.

Algorithm 7 is used to compute all events labelled by action a, belonging to $ex(C)$. It first iterates on all events e in the configuration C. If e is $preEvt(a, C)$, a new event e' = $\langle a, config(preEvt(a, C))\rangle$ is created (lines 3-4). Event e' is created because a and $pre(a)$ are dependent, and a is a non-blocking action, enabled after executing $pre(a)$. Next if e is an *AsyncReceive*$(m, \_)$ or *AsyncSend*$(m, \_)$ event, then e can possibly be a direct ancestor of a new event labelled by a. If e and $preEvt(a, C)$ are concurrent, both events are added to a set K. Otherwise, one of them is added to K (lines 8-9). If there is a dependence between a and the action of e (*i.e.* they concern the first done communication), then the algorithm creates a new event e'.

Algorithm 7: $createTestEvt$(a,C)

```
1  foreach event e in C do
2      if λ(e) = pre(a) then
3          K := {preEvt(a, C)};
4          create e' := ⟨a, config(K)⟩;
5          ex(C) := ex(C) ∪ {e'};
6      else if λ(e) is AsyncSend(m, _) or AsyncReceive(m, _) then
7          K := ∅;
8          if ¬(e < preEvt(a, C)) then K := K ∪ {e};
9          if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)};
10         if D_K(a, λ(e)) then
11             create e' := ⟨a, config(K)⟩;
12             ex(C) := ex(C) ∪ {e'};
```

$Actor_0$: c_0 = ***AsyncReceive***$(m, \_)$
WaitAny$(\{c\})$
c'_0= ***AsyncReceive***$(m, \_)$

$Actor_1$: c_1= ***AsyncSend***$(m, \_)$
c'_1= ***AsyncSend***$(m, \_)$
TestAny$(\{c'_1\})$

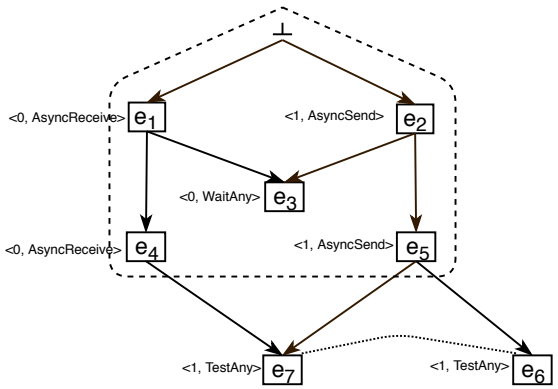


Figure 5.7 – A program, and a configuration C with extensions by action TestAny.

Example 11 *Let us illustrate how the Algorithm 7 works through the example given in Figure 5.7. Let* $C = \{e_1, e_2, e_3, e_4, e_5\}$ *and the action* a *is a TestAny action of* $Actor_1$*. Checking every event* e *in* C*, we have that* $\lambda(e_5) = pre(a)$*, then creating event* $e_6 = \langle$***TestAny***$, \{e_5\}\rangle$*. In C, there are two AsyncReceive events. First considering event* e_1*, we can create* $K = \{e_1, e_5\}$ *since* $\neg(e_1 < e_5)$ *and* $\neg(e_5 < e_1)$*. However, the condition* $D(a, \lambda(e_1))$ *is not satisfied, so we can not create a new event characterized by the action* a *and a configuration* $config(K)$*. Secondly,* e_4 *is examined, because* $\neg(e_5 < e_4)$ *and* $\neg(e_4 < e_5)$ *we build* $K = \{e_4, e_5\}$*. Since,* $D(a, \lambda(e_4))$ *holds, event* $e_7 = \langle$***TestAny***$, \{e_4, e_5\}\rangle$ *is created.*

5.1.6 Computing extensions for *AsyncMutexLock* actions.

Given a configuration C, and a an action of type ***AsyncMutexLock***(m_j) of an actor A_i, this section presents how to compute the set $S^{max}_{a,C}$ of sets K from which a depends.

According to independence theorems (see 4.4.6, theorem 4.4.7, theorem 4.4.8, theorem 4.4.11, theorem 4.4.12), action a only depends on the following actions: $pre(a)$, and ***AsyncMutexLock***(m_j) actions of distinct actors A_k which concern the same mutex m_j. In addition, according to the general properties (section 5.1.1), two events labelled with ***AsyncMutexLock***(m_j) actions cannot co-exist in K (they must be causally related), formally $\nexists e, e' \in K : \lambda(e), \lambda(e')$ are ***AsyncMutexLock***(m_j). We can conclude that, K contains at most two events: $preEvt(a, C)$, and an event labelled with an action ***AsyncMutexLock***(m_j) that concerns the same mutex as a.

Algorithm 8: $createLockEvt$(a,C)

```
1  foreach event e in C do
2      if λ(e) = pre(a) then
3          K := {preEvt(a, C)};
4          create e' := ⟨a, config(K)⟩;
5          ex(C) := ex(C) ∪ {e'}
6      else if λ(e) is AsyncMutexLock(m_j) then
7          K := ∅
8          if ¬(e < preEvt(a, C)) then K := K ∪ {e};;
9          if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)} ;
10         create e' := ⟨a, config(K)⟩;
11         ex(C) := ex(C) ∪ {e'};
```

Algorithm 8 is used to compute all events labelled by action a, belonging to $ex(C)$. It first iterates on all events e in the configuration C. If e is $preEvt(a, C)$, a new event e' = $\langle a, config(preEvt(a, C))\rangle$ is created (lines 3-5). Event e' is created because a and $pre(a)$ are dependent, and a is enabled after executing $pre(a)$. Next, if event e is an *AsyncMutexLock*(m_j) event, then e can possibly be a direct ancestor of a new event labelled by a. If e and $preEvt(a, C)$ are concurrent, both events are added to the set K. Otherwise, one of them, the event not ancestor of the other, is added to K (line 8-9). After that, the algorithm creates a new event e'.

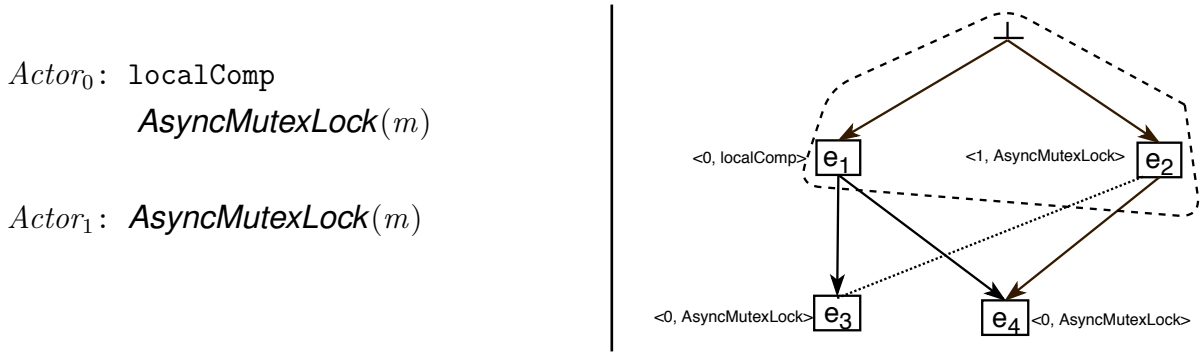


Figure 5.8 – A program and a configuration C with extensions by *AsyncMutexLock*.

Example 12 *Let's zoom in on the example presented in Figure 5.8. The distributed pro-*

gram consists of two actors, the first one executes two actions: a LocalComp action and an AsyncMutexLock(m) *action, the second one performs an action AsyncMutexLock*(m)*. Suppose we have to compute events labelled by action AsyncMutexLock*(m) *of* $Actor_0$ *in the extension of a configuration* C *= {*e_1, e_2*}. Iterating on all events* e *in* C*, we have* e_1 *=* $preEvt$(*AsyncMutexLock*$(m), C$)*, creating* $e_3 = \langle$*AsyncMutexLock (m),* $\{e_1\}\rangle$*. Next we find that* e_2 *is an AsyncMutexLock*(m)*. Since* $\neg(e_1 < e_2)$ *and* $\neg(e_2 < e_1)$*, both* e_1 *and* e_2 *are added to the set* K*. And then we can create* $e_4 = \langle$*AsyncMutexLock*$(m), \{e_1, e_1\}\rangle$*.*

5.1.7 Computing extensions for *MutexUnlock* actions.

Let C be a configuration, and a an action of type *MutexUnlock*(m) of an actor A_i. This section presents how to compute the set $S^{max}_{a,C}$ of sets K of maximal events from which a depends.

According to independence theorems (see 4.4.6, theorem 4.4.7, theorem 4.4.8, theorem 4.4.10, and theorem 4.4.12) a depends on $pre(a)$. It can also depends on all *MutexTestAny*(M), and *MutexWaitAny*(M) actions of distinct actors A_j if $m \in M$ and the two actors are the first two actors in the waiting queue of the mutex m. Recall that each mutex maintains a FIFO queue to store actors interests. If A_i is one of the first two actors in the queue, there is only one A_j such that it is also one of the first two actors. For that reason there is only one *MutexTestAny*(M) or *MutexWaitAny*(M) of some A_j depending on a, so there is at most one event e labelled by *MutexTestAny*(M) or *MutexWaitAny*(M) such that $\lambda(e) \neq pre(a)$.

To conclude, K contains at most two events: $preEvt(a, C)$, and an event labelled with an action *MutexTestAny*(M) or *MutexWaitAny*(M) on the same mutex.

Algorithm 9 is used to compute all events labelled by a *MutexUnlock* action, and belonging to $ex(C)$. Similar to Algorithm 8, it first iterates on all events e in the configuration C. If e is $preEvt(a, C)$, a new event e' = $\langle a, config(preEvt(a, C))\rangle$ is created (lines 3-4 in the algorithm). If event e is a *MutexTestAny*(M) or *MutexWaitAny*(M) event (lines 7-8), a set K is created. If e and $preEvt(a, C)$ are concurrent, both events are added to set K. Otherwise, one of them, the event not ancestor of the other, is added to K. If there is a dependence between a and the action of e, then the algorithm creates a new event e' = $\langle a, config(K)\rangle$, adding e' to $ex(C)$.

Algorithm 9: $createUnlockEvt$(a,C)

```
1  foreach event e in C do
2      if λ(e) = pre(a) then
3          K := {preEvt(a, C)};
4          create e' := ⟨a, config(K)⟩; ex(C) := ex(C) ∪ {e'}
5      else if λ(e) is MutexTestAny(M) or MutexWaitAny(M) then
6          K := ∅;
7          if ¬(e < preEvt(a, C)) then K := K ∪ {e};
8          if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)} ;
9          if D_K(a, λ(e)) then
10             create e' := ⟨a, config(K)⟩;
11             ex(C) := ex(C) ∪ {e'};
```

$Actor_0$: `localComp`
AsyncMutexLock(m)
MutexUnlock(m)

$Actor_1$: `localComp`
AsyncMutexLock(m)
MutexTestAny$(\{m\})$

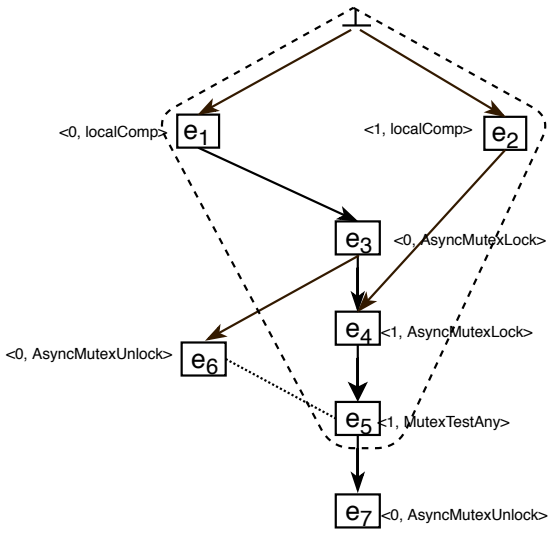


Figure 5.9 – A program, and a configuration C with extensions by MutexUnlock.

Example 13 *We illustrate the Algorithm 9 by the example of Figure 5.9. Suppose we want to compute the extensions of* C *=* $\{e_1, e_2, e_3, e_4, e_5\}$ *associated with* a*, the action MutexUnlock*(m) *of* $Actor_0$*. By iterating all events in* C*, we find that* $e_3 = preEvt(a, C)$*, then creating* e_6*. The configuration* C *has only one MutexTestAny event, event* e_5*. Note that* $Actor_0$ *and* $Actor_1$ *are the first two owners of the mutex* m*, then action* a *and action MutexTestAny are dependent. Besides, since* $e_3 < e_5$*, only event* e_5 *is added to the set*

K. *Thus, we create* $e_7 = \langle a, \{e_5\}\rangle$.

5.1.8 Computing extensions for *MutexWaitAny* actions.

Let C be a configuration, and a an action of type *MutexWaitAny*$(\{M\})$ of an actor A_i. This section presents how to compute the set $S^{max}_{a,C}$ of sets K of maximal events from which a depends.

Let M_1 be the subset of M composed of mutexes m' having A_i as one of the first two actors in their waiting queues. First, according to independence theorems (see theorem 4.4.6, theorem 4.4.7, theorem 4.4.10, theorem 4.4.12), a depends on a *MutexUnlock*(m) action of a distinct actor A_j if m is the first mutex in M_1 concerned by a *MutexUnlock*, and A_j is one of the first two actors in the waiting queue of the mutex m. Second, similar to the above section (section 5.1.7), if A_i is one of the first two actors in the queue, there is only one A_j such that it is also one of the first two actors in that queue. Thus, there is at most one event e in K labelled by *MutexUnlock*(m) such that $\lambda(e) \neq pre(a)$. In addition, a also depends on $pre(a)$. To conclude, K contains at most two events: $preEvt(a, C)$, and possibly an event labelled with an action *MutexUnlock*(m) such that $m \in M$.

Algorithm 10: $createMutexWaitEvt$(a,C)

```
1  foreach event e in C do
2      if λ(e) = pre(a) and A_i is the owner of some mutex m ∈ M then
3          K := {preEvt(a, C)};
4          create e' := ⟨a, config(K)⟩;
5          ex(C) := ex(C) ∪ {e'};
6      else if λ(e) is MutexUnlock then
7          K := ∅;
8          if ¬(e < preEvt(a, C)) then K := K ∪ {e};
9          if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)} ;
10         if D_K(a, λ(e)) then
11             create e' := ⟨a, config(K)⟩; ex(C) := ex(C) ∪ {e'} ;
```

Algorithm 10 trivially generates all events in $ex(C)$ labelled by an *MutexWaitAny*$(\{M\})$ action of actor A_i. It first iterates on all events e in the configuration C. If $\lambda(e)$ is $pre(a)$ and A_i is the owner of some mutex $m \in M$ then it creates a new event e' =

$\langle a, config(preEvt(a, C))\rangle$. If e is a *MutexUnlock* event, then it builds a set K such that all events in K are concurrent. If the condition $D(a, \lambda(e))$ holds then it creates a new event $e' = \langle a, config(K)\rangle$, adding it to $ex(C)$.

$Actor_0$: *LocalComp*
AsyncMutexLock(m)
MutexUnlock(m)

$Actor_1$: *LocalComp*
AsyncMutexLock(m)
MutexWaitAny(m)

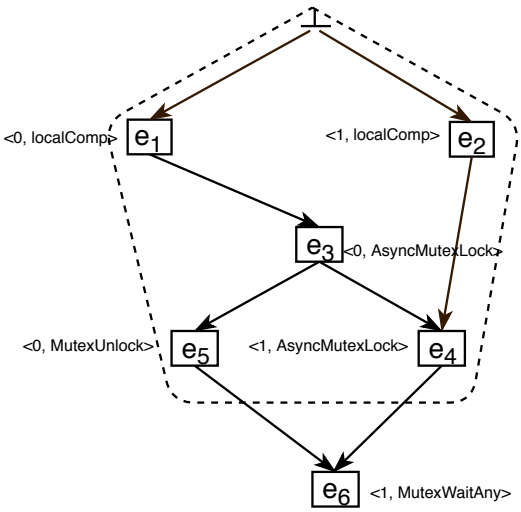


Figure 5.10 – A program, and a configuration C with extensions by *MutexWaitAny*.

Example 14 *We illustrate the Algorithm 10 by the example of Figure 5.10. Suppose we want to compute the extensions of C = {e_1, e_2, e_3, e_4, e_5} associated with action a, the action MutexWaitAny(m) of actor $Actor_1$. The configuration C has only one Mutex-Unlock event, event e_5. We also have that $Actor_0$ and $Actor_1$ are the first two owners of the mutex m, then action a and action MutexUnlock are dependent. Because event e_4 and e_5 are concurrent, both e_4 and e_5 are added to the set K. Thus, we create a new event $e_6 = \langle a, \{e_4, e_5\}\rangle$ in $ex(C)$.*

5.1.9 Computing extensions for *MutexTestAny* actions.

Let C be a configuration, and a an action of type *MutexTestAny*$(\{M\})$ of an actor A_i. This section presents how to compute the set $S^{max}_{a,C}$ of sets K of maximal events from which a depends. Compared to *MutexWaitAny* actions, *MutexTestAny* actions are not blocking, the main difference is that a *MutexTestAny* action a becomes enabled after executing its previous action ($pre(a)$) while a *MutexWaitAny* action becomes enabled if its actor is the owner of some mutex. We can use almost the arguments in the pre-

vious section to compute extensions labelled by *MutexTestAny* actions. Thus, we can conclude that K contains at most two events: $preEvt(a, C)$, an event labelled with an action *MutexUnlock*(m) such that $m \in M$.

Algorithm 11: createMutexTestEvt(a, C)

```
1  foreach event e in C do
2      if λ(e) = pre(a) then
3          K := {preEvt(a, C)};
4          create e' := ⟨a, config(K)⟩; ex(C) := ex(C) ∪ {e'};
5      else if λ(e) is MutexUnlock then
6          K := ∅;
7          if ¬(e < preEvt(a, C)) then K := K ∪ {e};
8          if ¬(preEvt(a, C) < e) then K := K ∪ {preEvt(a, C)};
9          if D_K(a, λ(e)) then
10             create e' := ⟨a, config(K)⟩; ex(C) := ex(C) ∪ {e'};
```

Algorithm 11 creates all events in $ex(C)$ labelled by a *MutexTestAny* action. It first iterates on all events e in the configuration C. If e is $preEvt(a, C)$ then it creates a new event $\langle a, config(K)\rangle$. If e is a *MutexUnlock* event, it builds a set K such that all events in K are concurrent. When the condition $D(a, \lambda(e))$ is satisfied, it creates a new event $e' = \langle a, config(K)\rangle$, and then adds that event to $ex(C)$.

$Actor_0$: `localComp`
AsyncMutexLock(m)
MutexUnlock(m)

$Actor_1$: `localComp`
AsyncMutexLock(m)
MutexTestAny$(\{m\})$

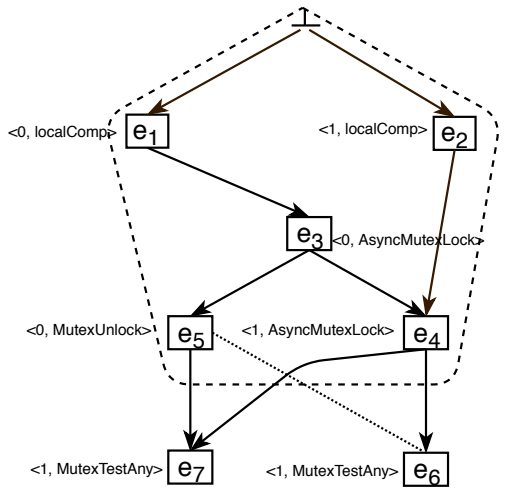


Figure 5.11 – A program, and a configuration C with extensions by *MutexTestAny*.

Example 15 *We illustrate the Algorithm 11 by the example of Figure 5.11. Suppose we want to compute the extensions of* C *=* $\{e_1, e_2, e_3, e_4, e_5\}$ *associated with* a*, the action MutexTestAny*$(\{m\})$ *of* $Actor_1$*. Since event* e_4 *is* $preEvt(a, C)$*, we create* e_6*. Next,* e_5 *is a MutexUnlock event in configuration* C*, and* e_4 *and* e_5 *are concurrent then we build* $K = \{e_4, e_5\}$*. We also have that* $Actor_0$ *and* $Actor_1$ *are the first two owners of the mutex* m*. Hence, action* a *and action MutexUnlock are dependent. Thus, we create* $e_7 = \langle a, \{e_4, e_5\}\rangle$*.*

5.1.10 Computing extensions for *LocalComp* actions.

Let C be a configuration, and a an action of type *LocalComp* of an actor A_i, computing extensions for action a is trivial since a is only dependent on the previous action of the same actor ($pre(a)$), and it becomes enabled after executing $pre(a)$. Hence, we can make a conclusion about the set K of maximal events from which a depends: K includes only $preEvt(a, C)$. Algorithm 12 computes all events in $ex(C)$ labelled by action a.

Algorithm 12: $createLocalCompEvt$(a,C)

```
1 foreach event e in C do
2     if λ(e) = pre(a) then
3         K := {preEvt(a, C)};
4         create e' := ⟨a, config(K)⟩;
5         ex(C) := ex(C) ∪ {e'};
```

5.2 Computing extensions incrementally

In the UDPOR exploration algorithm, after extending a configuration C' by adding a new event e, one must compute the extensions of $C = C' \cup \{e\}$, thus resulting in redundant computations of events. Thanks to the persistence property, almost all such recomputations can be eliminated. The next theorem provides an incremental computation of extensions.

Theorem 5.2.1 *Suppose* $C = C' \cup \{e\}$ *where* e *is the last event added to* C *by the*

Algorithm 2. We can compute $ex(C)$ *incrementally as follows:*

$$ex(C) = (ex(C') \cup \bigcup_{a \in enab(C)} \{< a, H >: H \in S_{a,C}\}) \setminus \{e\} \tag{5.7}$$

where $S_{a,C} = \{H \in 2^C \cap conf(E) : a \in enab(H) \wedge Depend(a, maxEvents(H))\}$.

With the definition of $S_{a,C}$ as above, recall that (see equation 5.5)

$$ex(C) = (\bigcup_{a \in actions(C) \cup enab(C)} \{\langle a, H\rangle : H \in S_{a,C}\}) \setminus C \tag{5.8}$$

Similarly we have:

$$ex(C') = (\bigcup_{a \in actions(C') \cup enab(C')} \{\langle a, H'\rangle : H' \in S_{a,C'}\}) \setminus C' \tag{5.9}$$

We have the following lemma that will be used to prove theorem 5.2.1.

Lemma 2 *Suppose* $C = C' \cup \{e\}$ *where* e *is the last event added to* C *by the Algorithm 2. We have the following relation:*

$$\bigcup_{a \in actions(C)} \{< a, H >: H \in S_{a,C}\} = \bigcup_{a \in actions(C)} \{< a, H' >: H' \in S_{a,C'}\} \tag{5.10}$$

where $S_{a,C} = \{H \in 2^C \cap conf(E) : a \in enab(H) \wedge Depend(a, maxEvents(H))\}$ *and* $S_{a,C'} = \{H' \in 2^{C'} \cap conf(E) : a \in enab(H') \wedge Depend(a, maxEvents(H'))\}$.

Proof 14 *(⊇) This inclusion is obvious since* $C \supseteq C'$*, and thus* $S_{a,C} \supseteq S_{a,C'}$.
(⊆) Suppose there exists some event $e_n =< a, H >$ *belonging to the left but not the right set. If* $a = \lambda(e_n) = \lambda(e)$*, then* $H \in S_{a,C} \cap S_{a,C'}$*, so* e_n *is in both sets, resulting in a contradiction. If* $a = \lambda(e_n) \neq \lambda(e)$*, there are two cases: (i) either* $e \notin H$ *then* $H \in S_{a,C'}$ *and* e_n *belongs to the right set, a contradiction. Or (ii)* $e \in H$*, then* $\lambda(e_n) \in actions(C) \setminus \{\lambda(e)\} = actions(C')$*, thus there is another event* $e' \in C'$ *such that* $\lambda(e') = \lambda(e_n)$*, then* e' *cannot belong to* H *(one action* a *cannot appear twice in* $\lceil e_n \rceil$*). Besides,* e *is the last event explored in* C*, thus* a *depends on* $\lambda(e)$ *by Definition 6. Then,* e' *conflicts with* e*, contradicting their membership to the same configuration* C *(see Figure 5.12). This proves (5.10).* ■

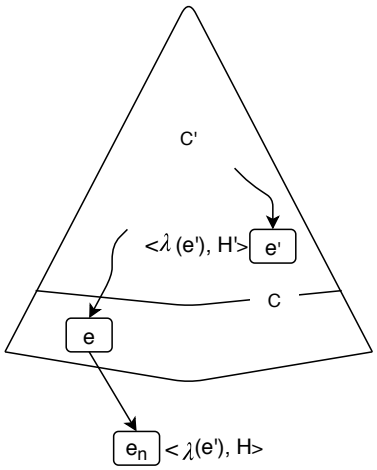


Figure 5.12 – Since $D(\lambda(e), \lambda(e'))$, events e and e' are in conflict.

We now can prove theorem 5.2.1 as follows:

Proof 15 *We have:*

$$ex(C) = (\bigcup_{a \in actions(C) \cup enab(C)} \{\langle a, H\rangle : H \in S_{a,C}\}) \setminus C \tag{5.11}$$

and

$$ex(C') = (\bigcup_{a \in actions(C') \cup enab(C')} \{\langle a, H'\rangle : H' \in S_{a,C'}\}) \setminus C' \tag{5.12}$$

Exploring e from C' leads to C, which entails that $\lambda(e)$ belongs to $enab(C')$ and $actions(C') \cup \{\lambda(e)\} = actions(C)$, thus the range of a in $ex(C')$ which is $actions(C') \cup enab(C')$ can be rewritten $actions(C) \cup (enab(C') \setminus \{\lambda(e)\})$, and the equation 5.12 changes as follows:

$$ex(C') = (\bigcup_{a \in actions(C) \cup (enab(C') \setminus \lambda(e))} \{\langle a, H'\rangle : H' \in S_{a,C'}\}) \setminus C'$$

First, separating $action(C)$ from the rest in both $ex(C)$ and $ex(C')$, and according to lemma 2, we have that :

$$\bigcup_{a \in actions(C)} \{< a, H >: H \in S_{a,C}\} = \bigcup_{a \in actions(C)} \{< a, H' >: H' \in S_{a,C'}\} \tag{5.13}$$

Second, since $C' \subseteq C$, according to Lemma 1, $(enab(C') \setminus \{\lambda(e)\}) \subseteq enab(C)$. We

thus have:

$$\bigcup_{a \in enab(C')\setminus\{\lambda(e)\}} \{< a, H' >| H' \in S_{a,C'}\} \subseteq \bigcup_{a \in enab(C)} \{< a, H >| H \in S_{a,C}\} \quad (5.14)$$

Now, using equations (5.13) and (5.14), $ex(C)$ *can be rewritten as follows:*

$$\begin{aligned} ex(C) = (& \bigcup_{a \in actions(C) \cup (enab(C')\setminus\lambda(e))} \{\langle a, H' \rangle : H' \in S_{a,C'}\} \\ & \cup \bigcup_{a \in enab(C)} \{\langle a, H \rangle : H \in S_{a,C}\}) \setminus (C' \cup \{e\}) \end{aligned} \quad (5.15)$$

But since no event in $\bigcup_{a \in enab(C)}\{\langle a, H\rangle : H \in S_{a,C}\}$ *is in* $(C' \cup \{e\})$*, equation (5.15) can be rewritten as Equation (5.7) in Theorem 5.2.1.* ■

Algorithm 13: Computing extensions for a configuration

```
1  ex(C) := ex(C') \ {e}
2  foreach action a ∈ enab(C) do
3      if (λ(e) = AsyncSend) then call createAsyncSendEvt(a, C);
4      if (λ(e) = AsyncReceive) then call createAsyncReceiveEvt(a, C);
5      if (λ(e) = TestAny) then call createAsyncTestEvt(a, C);
6      if (λ(e) = WaitAny) then call createAsyncWaitEvt(a, C);
7      if (λ(e) = AsyncMutexLock) then call createLockEvt(a, C);
8      if (λ(e) = MutexUnlock) then call createUnlockEvt(a, C);
9      if (λ(e) = MutexTestAny) then call createMutexTestEvt(a, C);
10     if (λ(e) = MutexWaitAny) then call createMutexWaitEvt(a, C);
11     if (λ(e) = LocalComp) then call createLocalCompEvt(a, C);
```

Algorithm 13 is a complete algorithm computing the extension for a configuration C. It calls the algorithms presented in the previous sections. After assigning $ex(C) := ex(C')\setminus\{e\}$, it iterates on all actions a enabled at the state of configuration C, and then a function is called based on the type of a.

5.3 Computing dependence relations

As presented previously, computing the extensions of a configuration can be specialized according to the type of action; most of the presented algorithms computing

extensions require determining the dependence between actions. Usually checking dependence/independence of actions is a trivial task. For example, if we consider that two write actions, or a write action and a read action are dependent if they concern the same variable, then by examining the memory locations where the actions access and what they do with that memory one can get their relation. In our model, in order to compute dependencies based on independence theorems, and for some pairs of actions, we must rely on events that have been performed before. This section illustrates how dependencies are computed.

In [34], authors consider a computation model consisting of only three kinds of actions: lock, unlock, and local actions. Two lock actions that require the same mutex are dependent, or a lock action and an unlock action involving the same mutex are dependent. Hence, checking the dependence between two actions of programs in such a model is simple. Besides, the relation between actions is independent of the context (context-insensitive dependence) in the sense that if two actions are dependent, they will be dependent in all configurations (executions) in the unfolding of the program.

In our computation model, there are some dependence relations of actions that are context-insensitive (*e.g.,* two *AsyncSend* actions concerning the same mailbox, two *AsyncReceive* concerning the same mailbox, and two *AsyncMutexLock* requiring the same mutex). Such kind of relations is easy to determine by checking which mailboxes or mutexes they involve.

Besides context-insensitive dependence, some dependencies are context-sensitive: two dependent actions in a particular configuration can become independent actions in another configuration. Consider the example shown in Figure 5.13, a distributed program composed of three actors. Let's zoom in on the unfolding of the program. In configuration $C_1 = \{e_1, e_2, e_5, e_7, e_8\}$ (the configuration in the red line), action *AsyncSend* of $Actor_1$ and action *TestAny* of $Actor_0$ are dependent because they concern the first done communication in mailbox m. Besides, in that configuration, action *TestAny* of $Actor_2$ is independent with action *AsyncSend* of $Actor_1$. However, in another configuration $C_2 = \{e_2, e_3, e_9, e_{10}, e_{11}\}$, action *AsyncSend* is dependent on *TestAny* of $Actor_2$ and independent with *TestAny* of $Actor_0$. Obviously some dependencies of pairs of communication actions are context-sensitive.

$Actor_0$:	$Actor_1$:	$Actor_2$:
`c0= AsyncReceive(m,_)`	`c1= AsyncSend(m,_)`	`c2= AsyncReceive(m,_)`
`TestAny({c})`		`TestAny({c2})`

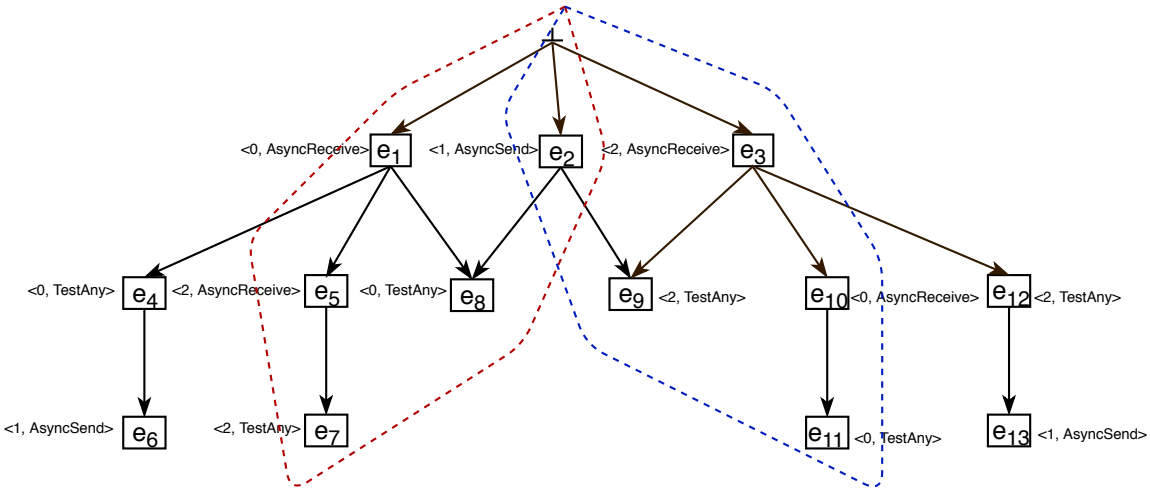


Figure 5.13 – The pseudo-code of a distributed program and the it's unfolding.

$Actor_0$:	$Actor_1$:	$Actor_2$:
`r0 = AsyncMutexLock(mtx)`	`r1 = AsyncMutexLock(mtx)`	`r2 = AsyncMutexLock(mtx)`
`MutexUnlock(mtx)`	`MutexTest({r1})`	`MutexUnlock(mtx)`

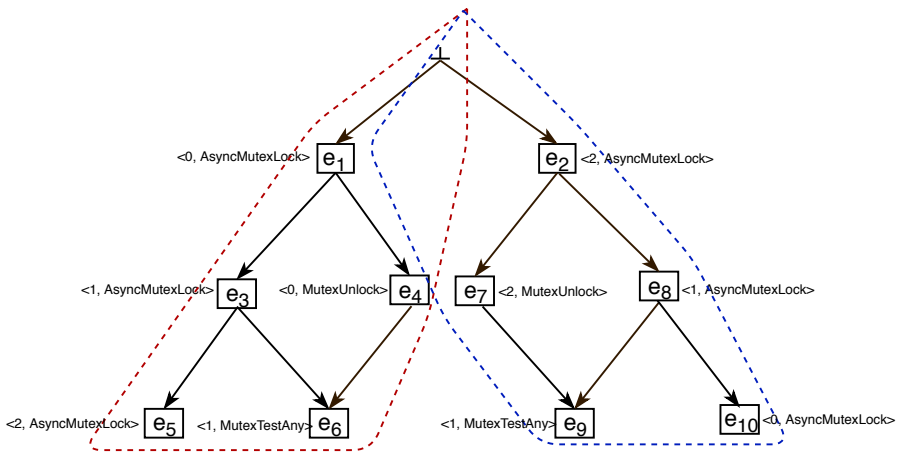


Figure 5.14 – The pseudo-code of a distributed program and two configurations.

Let's consider another example presented in Figure 5.14. A distributed program consists of three actors executing some actions. To keep it simple, we illustrate only two maximal configurations marked with dashed lines. In the first configuration (marked by the red line), action *MutexTestAny* of $Actor_1$ and *MutexUnlock* of $Actor_0$ are dependent, while in the second configuration (marked by the blue line) they are independent.

Through the two above examples, we can conclude that with communication actions the relations between an *AsyncSend* action or an *AsyncReceive* and a *TestAny* or a *WaitAny* are context-sensitive. Similarly, with synchronization actions, the relations between a *MutexUnlock* and a *MutexTestAny* or *MutexWaitAny* are also context-sensitive. Therefore, one of the critical technical challenges is how to determine the dependence between actions in case of context-sensitive dependence.

For communication actions, context-sensitive dependencies arise because different configurations may have different combinations of *AsyncSend* and *AsyncReceive* communications to produce done communications. According to the independence theorems, an *AsyncSend*(m) or an *AsyncReceive*(m) and a *TestAny*$(\{C\})$ or a *WaitAny*$(\{C\})$ are dependent if they concern the first done communication $c \in C$. Two actions concerning a first done communication in a particular configuration may not be involved in a first done communication in another configuration. Thus, checking dependence entails specifying which *AsyncSend*, *AsyncReceive* communications are paired to produce done communications. Similarly, context-sensitive relations of synchronization actions come from the fact that changing the execution order of *AsyncMutexLock* actions involved in a mutex leads to different owners of that mutex. So, first two actors in the waiting queue of the mutex in different configurations are not always identical. Thus, dependence can be checked by checking which actors are being stored in the waiting queues of mutexes and in which order. The next sections demonstrate how dependence can be detected between two actions involved by a context-sensitive relation.

5.3.1 Computing dependencies for communication actions

As presented, checking context-sensitive dependencies between two communication actions entails examining done communications. Consider the example in Figure 5.15, a distributed program with three actors. In the figure we draw a part of the unfolding of the program. Let us zoom in on events e_7, e_9, they have the corresponding maximal event sets: (their fathers) $K_{e_7} = \{e_5, e_6\}$ and $K_{e_9} = \{e_2, e_3, e_8\}$, respectively.

$Actor_0$:	$Actor_1$:	$Actor_2$:
c_0 = *AsyncSend*(m)	c_1 = *AsyncReceive*(m)	`LocalComp`
c_0' = *AsyncSend*(m)	c_1' = *AsyncReceive*(m)	c_2 = *AsyncReceive*(m)
c_0'' = *AsyncSend*(m)		
`TestAny(`$\{c_0'\}$`)`		

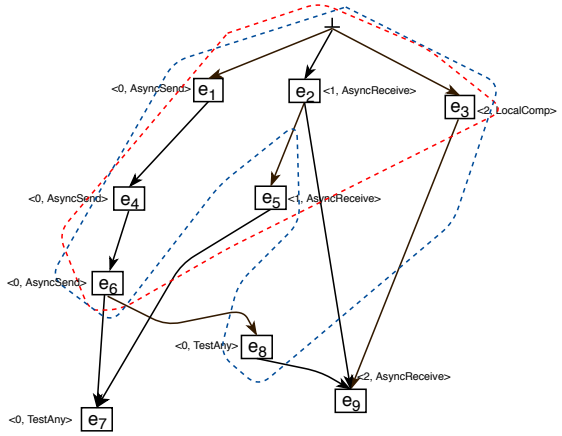


Figure 5.15 – The pseudo-code of a distributed program and two configurations.

Besides, $\lambda(e_7)$ is enabled at the state of configuration $C1 = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ while $\lambda(e_9)$ is enabled at the state of configuration $C2 = \{e_1, e_2, e_3, e_4, e_6, e_8\}$. We have that $D(\lambda(e_5), \lambda(e_7))$ because they concern the first done communication in $\{c_0'\}$ that is a pairing between c_0' and c_1'. However, how do we know that c_0' and c_1' are paired to form the first done communication? One can rely on the information stored at $state(C1)$ to know how the first done communication is created.

Things become more complicated for the case of e_9. Event e_9 is created because $\lambda(e_9)$ is dependent on all actions of events in K_{e_9} = {e_2, e_3, e_8} in which $\lambda(e_2)$ (action *AsyncReceive*(m) of $Actor_1$) and $\lambda(e_9)$ (*AsyncReceive*(m) of $Actor_2$) concern the same mailbox, $\lambda(e_3)$ = $pre(\lambda(e_9))$ (*LocalComp* of $Actor_2$) while $\lambda(e_8)$ (*TestAny* of $Actor_0$) and $\lambda(e_9)$ concern a first done communication in $\{c_0'\}$. The same question is why $\lambda(e_8)$ and $\lambda(e_9)$ concern the first done communication in $\{c_0'\}$ (*i.e.,* the done communication is the result of pairing c_0' and c_2). Using information which is stored at $state(C2)$ is not helpful since c_2 has not previously posted to mailbox m at that state.

Let us zoom in closer on the histories of events e_7 and e_9. At $state(C1)$, there are three *AsyncSend* and two *AsyncReceive* communications posted. Communication c_0' is the second *AsyncSend* communication posted to mailbox m after c_0. Hence, c_0' is paired with c_1' (the second *AsyncReceive* communication posted to the mailbox) and

then $D(\lambda(e_7), \lambda(e_5))$. Regarding event e_9, we have three *AsyncSend* communications and only one *AsyncReceive* communication posted to mailbox m at $state(C2)$. Because c'_0 is the second *AsyncSend* communication, it will be matched with c_2 after executing $\lambda(e_9)$. Thus, $\lambda(e_8)$, and $\lambda(e_9)$ concern the same communication and then $D(\lambda(e_8), \lambda(e_9))$.

From the above example, we see that counting the number of *AsyncReceive* and *AsyncSend* communications can helps us determine whether a *TestAny* (or *WaitAny*) action and an *AsyncSend* (or *AsyncReceive*) action are dependent or not. Let $nbSend(m, H)$ and $nbReceive(m, H)$ denote the number of *AsyncSend*$(m, \_)$ events and number of *AsyncReceive*$(m, \_)$ events in configuration H, respectively. Checking dependence between a *TestAny* (or a *WaitAny*) action and an *AsyncSend* (or an *AsyncReceive*) action can leverage on Lemma 3.

Lemma 3 *Let C be a configuration, a be a TestAny ({c}) (or WaitAny ({c})) action such that a is enabled at $state(C)$, and c is an AsyncSend communication that is posted to the mailbox m by an event e: c = AsyncSend$(m, _)$. Action a is dependent on the action of some event in e' : c' = AsyncReceive$(m, _)$ (in C) if $nbSend(m, \lceil e \rceil) = nbReceive(m, \lceil e' \rceil)$.*

Proof. *Suppose $n = nbSend(m, \lceil e \rceil) = nbReceive(m, \lceil e' \rceil)$. According to general properties (in page 82), in the configuration C, all AsyncSend$(m, _)$ events are causally related, and all AsyncReceive$(m, _)$ events are also causally related. Beside, remember that mailbox m is a FIFO queue. Hence, in the mailbox m, the first n AsyncSend communications posted to m by n AsyncSend$(m, _)$ events in $\lceil e \rceil$ are combined with the first n AsyncReceive communications that are issued by n AsyncReceive$(m, _)$ events in $\lceil e' \rceil$. For those reasons, c ($n+1$th AsyncSend communication posted to m by e) and c' ($n+1$th AsyncReceive communication issued by e') are combined to create a done communication, and thus a and $\lambda(e')$ are dependent.*

Algorithm 14: $ChekingDependence(a, e, C)$

```
1 if λ(e) is an AsyncReceive (m,_) action then
2 |   e1 := the event whose action issues the communication c;
3 |   return (nbSend(m, ⌈e1⌉) == nbReceive(m, ⌈e⌉));
4 else
5 |   return false;
```

Given a *TestAny*$(\{c\})$ (resp. *WaitAny*$(\{c\})$) action a that is enabled at $state(C)$ and tests (resp. waits) a *AsyncSend*$(m, \_)$ communication. Algorithm 14 checks whether a depends on the action of some event e in the configuration C or not. We can easily lift up this idea to check dependencies of other cases of communication actions (*e.g.,* c is an *AsyncReceive* communication, a is an *AsyncSend* or *AsyncReceive* action).

5.3.2 Computing dependencies for synchronization actions

Let C be a configuration, a a *MutexTestAny*$(\{mtx\})$ action of actor A_i enabled at $state(C)$. Suppose that we must find all events $e \in C$ such that $D(a, \lambda(e))$. According to independence theorems (see theorem 4.4.10), action a only depends on some action *MutexUnlock*(mtx) b of actor A_j if A_i and A_j are the first two actors in mtx. So, if at state C, A_i is not the first actor in mtx then there is no *MutexUnlock*(mtx) action b in C such that $D(a, b)$. Otherwise, the last executed *MutexUnlock*(mtx) of some actor A_j is dependent on a. Because, after executing that action, A_i becomes the first owner of the mutex. Obviously, before executing it, A_i and A_j are the first two actors in the mutex mtx.

$Actor_0$:
```
r0 = AsyncMutexLock(mtx)
MutexUnlock(mtx)
```

$Actor_1$:
```
r1 = AsyncMutexLock(mtx)
MutexTest({r1})
MutexUnlock(mtx)
```

$Actor_2$:
```
r2 = AsyncMutexLock(mtx)
MutexTest({r2})
```

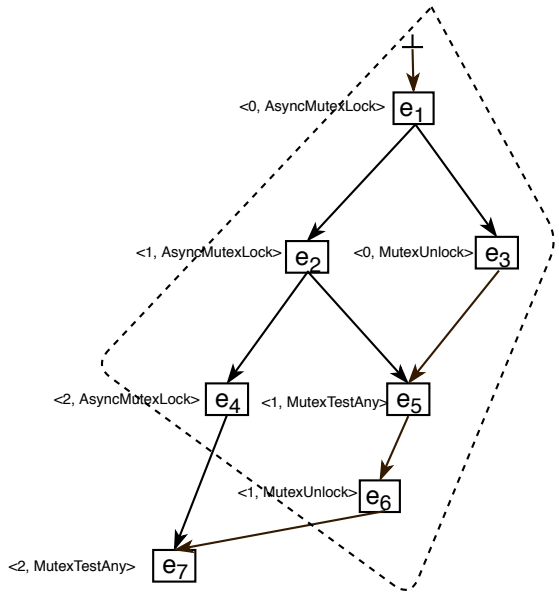


Figure 5.16 – a program and a configuration C.

Let us start with the example presented in Figure 5.16 to illustrate the above idea. In

the example, a part of the unfolding of a distributed program is illustrated. At the state of a configuration $C = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, action *MutexTestAny*$(\{r_2\})$ is enabled. Suppose we must find all actions *MutexUnlock* in C depending on action *MutexTestAny*$(\{r_2\})$. We see that *MutexUnlock* of $Actor_1$ is the last executed *MutexUnlock* concerning the mutex mtx. Thus, *MutexUnlock* of $Actor_1$ and *MutexTestAny*$(\{r_2\})$ are dependent, and then e_7 is created.

Algorithm 15: $ChekingDependence(a, e, C)$

```
1 if the actor of action a is the owner of mtx then
2 |   if λ(e) is the last executed action concerning mtx then
3 |   |   return true;
4 |   else
5 |   |   return false;
6 else
7 |   return false;
```

Let a be a *MutexTestAny*(mtx) action that is enabled at the state of a configuration C. Let e be an *MutexUnlock*$(\{mtx\})$ event in C, Algorithm 15 checks whether a and $\lambda(e)$ are dependent or not. For other pairs of actions (*e.g., MutexWaitAny* and *MutexUnlock*), we proceed similarly.

5.4 Experiments

We implemented the quasi-optimal version of UDPOR with k-partial alternatives [34] in a prototype adapted to the distributed programming model of chapter 4, *i.e.* with its independence relation. The computation of k-partial alternatives is essentially inspired by [34]. Recall that the algorithm reaches optimality when $k = |D| + 1$ (recall that we avoid to explore events in D), while $k = 1$ corresponds to Source DPOR [1]. The prototype is still limited, not connected to the SimGrid environment because SimGrid needs an API to integrate new exploration and reduction algorithms. Thus, it can only be experimented on simple examples. Besides, because of the fact that it has not been integrated into SimGrid, the input of the prototype is the encoding of an MPI program, not a real MPI program.

Benchmarks	#P	Deadlock	Exhaustive search		UDPOR	
			#Traces	Time	#Traces	Time
wait-deadlock	2	yes	2	<0.01	1	<0.01
send-recv-ok	2	no	24	0.03	1	<0.01
sendrecv-deadlock	3	yes	105	0.06	1	0.01
complex-deadlock	3	yes	36	0.03	1	<0.01
waitall-deadlock	3	yes	1458	1.2	1	<0.01
no-error-wait-any-src	3	no	21	0.02	1	0.01
any-src-waitall-deadlock	3	no	105	0.05	1	0.01
any-src-can-deadlock3	3	yes	999	0.65	2	0.03
DTG	5	yes	-	TO	2	0.07
RMQ-receiving	4	no	20064	8.15	6	0.2
	5	no	-	TO	24	2.52
	6	no	-	TO	120	47
Master-worker	3	no	1356444	1038	2	0.2
	4	no	-	TO	6	2.5
	5	no	-	TO	24	60

Table 5.1 – Comparing exhaustive exploration and UDPOR. TO: timeout after 30 minutes; #P: number of processes; Deadlock: deadlock exists; #Traces: number of traces

Comparison We first compare optimal UDPOR with an exhaustive stateless search on several benchmarks (see Table 5.1). The first 8 benchmarks come from Umpire_Tests [1], while DTG and RMQ-receiving belong to [24] and [41], respectively. The last benchmark is an implementation of a simple Master-Worker pattern. We expressed them in our programming model and explored their state space with our prototype. The experiments were performed on an HP computer, Intel Core i7-6600U 2.60GHz pro-

1. `http://formalverification.cs.utah.edu/ISP-Tests/`

cessors, 16GB of RAM, and Ubuntu version 18.04.1. Table 5.1 presents the number of explored traces and running time for both an exhaustive search and optimal UDPOR. In all benchmarks, UDPOR outperforms the exhaustive search. For example, for RMQ-receiving with four processes, the exhaustive search explores more than 20000 traces in around 8 seconds, while UDPOR explores only six traces in 0.2 seconds. Besides, UDPOR is optimal, exploring only one execution per Mazurkiewicz trace. For example, in RMQ-receiving with five processes, with only four *AsyncSend* actions that concern the same mailbox, UDPOR explores exactly 24 (=4!) non-equivalent executions. Similarly, the DTG benchmark has only two dependent *AsyncSend* actions, thus two non-equivalent traces. Furthermore, deadlocks are also detected in the prototype.

Variations on k We also tried to vary the value of k in the computation of k-partial alternatives. When k is decreased, one gains in efficiency in computing alternatives, but loses optimality by producing more traces. It is then interesting to analyze whether this can be globally more efficient than optimal UDPOR. Similar to [34], we observed that in some cases, fixing small values of k may improve efficiency. Let us comment on some instances:

— For RMQ-receiving with five processes, $k = 7$ is optimal (2.5s), but reducing to $k = 4$ still produces 24 traces (thus is optimal) a bit more quickly (2.3 seconds), while for $k = 3$, it is not optimal, generating one SSB (the number of traces is 25), but the run time reduces to only 2 seconds. When $k = 2$, both the number of traces and the run time grows quickly, it timeouts (after 10 minutes).

— For RMQ-receiving with six processes, $k = 11$ is optimal, and the run time is 47 seconds as presented in the above table. When $k = 5$, it is still optimal and takes 34 seconds to cover the 120 traces. One SSB is produced when $k = 4$ (121 traces are traversed), but the run time drops to only 28 seconds. Besides, it timeouts if $k = 3$.

— The run time of the Master-worker benchmark with five processes also varies when changing the value of k. $k = 7$ is optimal (60s), but if $k = 5$ it is still optimal, exploring the same traces (24 traces) in 57 seconds. When $k = 4,$ the run time goes down to 51 seconds, and there is no SSB.

Note that with our simple prototype, we do not yet make experiments with concrete programs (*e.g.* MPI programs), for which the run time may diverge. We expect to make

it in the future and then experiment the algorithms in more depth. However, we believe that the results are already significant and that UDPOR is effective for asynchronous distributed programs.

5.5 Conclusion

This chapter intensively discusses how to compute extensions of a configuration efficiently. Given a configuration C and an action a, to compute $ex(C)$ we could use a naive algorithm iterating all subsets of C, building history candidates for new events labelled by action a. However this algorithm leads to an exponential number of subsets. Using the fact that each configuration (candidate) H is identified by its maximal events $maxEvents(H)$, instead of building H we build the maximal event set K of H. We have proved that the number of events in K is bounded. If a is a *LocalComp* action, $|K|$ is always equal to 1. If action a is *AsyncSend* or *AsyncReceive*, in the worse case $|K| = 3$, and for the others types of actions (*i.e., TestAny*, *WaitAny*, *AsyncMutexLock*, *MutexTestAny*, *MutexWaitAny*), $|K| = 2$. Thus, extensions can be computed in polynomial time. Table 5.2 summarizes possible values of K according to the type of action a.

Besides presenting how to compute extensions efficiently, this chapter also demonstrates a theorem avoiding recomputation of many events. The UDPOR algorithm is recursive, and after extending a configuration C by adding a new event e the algorithm computes $ex(C \cup \{e\})$. Thus many events that are already computed in $ex(C)$ are recomputed in $ex(C \cup \{e\})$. Thanks to the persistence property of the model, the theorem eliminates most of this redundant computations and shows that considering only actions that are enabled at $state(C)$ is enough.

The UDPOR algorithm, together with our ideas, are implemented in a prototype. Experiments conducted on several benchmarks in the chapter show significant reductions obtained by the prototype when verifying MPI programs. We also run k-partial alternatives with different values of k. The results obtained by experiments once again confirm that in some cases, with low values of k, we still obtain optimal results, and k-partial alternatives can be faster than optimal DPOR if the benchmark contains few SSBs. However, this chapter also opens some works to be done. For example, we need to compare the prototype with state-of-the-art verification tools for MPI programs, as well as proposing an automated method to propose good values of k in calculating

k-partial alternative.

Type of action	Description of K
AsyncSend	K $\subseteq$ {$preEvt(a, C)$, *AsyncSend*, *TestAny* }
AsyncReceive	K $\subseteq$ {$preEvt(a, C)$, *AsyncReceive*, *TestAny* }
TestAny	K $\subseteq$ { $preEvt(a, C)$, *AsyncSend* (or *AsyncReceive*) }
WaitAny	
AsyncMutexLock	K $\subseteq$ {$preEvt(a, C)$, *AsyncMutexLock* }
MutexUnlock	K $\subseteq$ {$preEvt(a, C)$, *MutexTestAny* }
MutexTestAny	K $\subseteq$ {$preEvt(a, C)$, *MutexUnlock* }
MutexWaitAny	
LocalComp	K $\subseteq$ {$preEvt(a, C)$}

Table 5.2 – Possible values of K according to the type of action a

CHAPTER 6

CONCLUSION AND PERSPECTIVES

6.1 Conclusion

Using model checking techniques to verify concurrent programs is still a challenge for the formal methods community because of an intractable problem, that is, the state space explosion. Since the model checking method was born, much attention has been paid to mitigate this problem, typically studies of POR techniques. In the last few years, we have seen the birth of two new efficient methods outperforming the POR techniques, namely UDPOR and optimal DPOR. They are considered optimal since completely avoiding redundant explorations. Besides, distributed programs are in the mainstream of information technology. When writing HPC programs, MPI libraries are usually chosen by developers because they provide many useful functions and manners for developing such kind of applications. For that reason, many HPC applications that run on super-computing platforms are written by using MPI libraries. HPC applications can bring intensive performances, but verifying them is an error-prone task. So, they have attracted interest from researchers in the model checking community, typically Ganesh Gopalakrishnan and his colleagues. Similar to verifying other concurrent programs, model checking of MPI programs also faces the state space explosion problem since concurrency and non-determinism that mainly result in state space explosion are main features of MPI applications where many processes operate simultaneously.

The main goal of the thesis aims at adapting a state of the art method that is Unfolding-based DPOR to model-check real MPI programs in the setting of the SimGrid simulator. Figure 6.1 depicts the workflow that we use in order to reach the main target of the thesis, verifying MPI applications. Let's summarize some mains steps in the thesis.

Abstract model In the thesis, we defined a compact abstract model with very few kernel actions. At the semantic level, the abstract model consists of a set of actors and

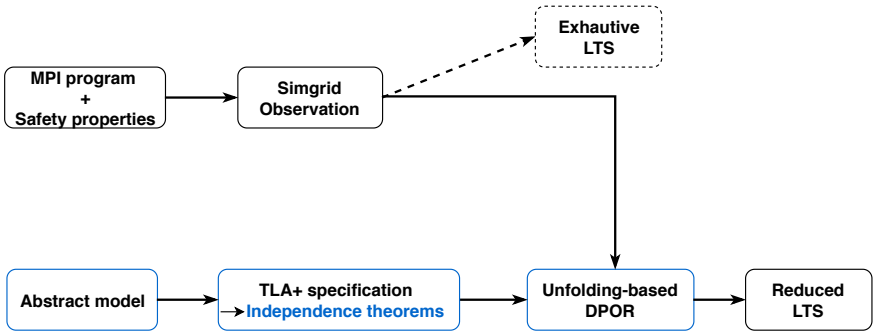


Figure 6.1 – Workflow.

two subsystems, namely the synchronization subsystem and the communication subsystem. Actors can execute three kinds of actions, namely local computation actions, synchronization actions, and communication actions. The synchronization subsystem is composed of some FIFO mailboxes where communications meet and pair, and a Communications object that stores all done communications in the system. The synchronization subsystem includes some mutexes used for actors synchronization tasks. From a formal point of view, a significant advantage of the abstract model is that with a very small amount of kernel actions, it can be used to encode a large class of MPI programs. So, instead of formalizing a huge number of MPI functions that is an error-prone task, specifying the semantics of these actions to reason about the dependency of actions becomes easy.

Specification of the abstract model As presented, Partial order reduction methods reduce the state space of a program by exploiting the commutation of independent actions. Determining whether actions are independent requires a precise formal semantics of the program. So, after defining it abstractly, the abstract model is specified in TLA+ in order to obtain its semantics. Then, independence theorems are proved based on the formal TLA+ specification, and they are later used as an input of UDPOR to compute independence relations between actions of programs to be verified.

Adapting UDPOR Experiments already showed that UDPOR efficiently combats the state space explosion of concurrent programs consisting of several processes that communicate through mutexes. However, to the best of our knowledge, it has not been applied to distributed programs where message passing is employed. This work is the

first attempt to verify message passing programs by using UDPOR. However, verifying such programs requires efficiently computing extensions of configurations. By careful observation of the relationship between events in a configuration, and using the persistence property of the abstract model, we proposed methods to efficiently compute the extensions in the context of our model. The main flavor of the methods is that when computing events for a given action a and a configuration C, we point out that a can only depend on the actions of very few and easily identifiable events in C. Thus we can compute new events efficiently.

Implementation All our proposed algorithms are implemented in a prototype, and some benchmarks have also been experimented. Although the prototype now works with simple benchmarks since it has not been yet integrated into the SimGrid environment. These experiments have implications in demonstrating that UDPOR is fully applicable to ensure that MPI programs do not have unwanted behaviors. Besides, the implementation of the quasi-optimal version of UDPOR with k-partial alternatives and varying the value of k in some benchmarks reveals that using k-partial alternatives in message passing programs can also gain benefits: UDPOR can still be optimal with a low value of k; or it can have redundant explorations, but the run time decreases.

6.2 Perspectives

Our work opens a number of perspectives. In this section, we discuss the most promising ones.

Integration As presented, the prototype now has not been integrated into SimGrid since it needs an API that in turn needs rather intrusive refactoring of the framework. In the future, we aim at extending our model of asynchronous distributed systems, while both preserving good properties, and implementing UDPOR in the SimGrid model checker and verify real MPI applications. Once done, we should experiment UDPOR more deeply, and compare it with state of the art tools on more significant benchmarks, get a more precise analysis about the efficiency of UDPOR compared to simpler DPOR approaches, analyze the impact of quasi-optimality on efficiency.

Improving performance We now check conflicts between two events by scanning the local configuration of the two events and checking for immediate conflicts between the causes. This is rather slow. In [34], the authors proposed an efficient method for deciding causality and conflicts between the events of an unfolding of a data-race-free program that uses mutexes. We believe that the method can be extended for our model to improve the performance of UDPOR.

Refining the independence relation A simple solution to have a better reduction of UDPOR is to refine the independence relation: the more precise, the less Mazurkiewicz traces exist, thus the more efficient could be UDPOR. For example, we now have that two *AsyncSend* actions are dependent if they concern the same mailbox. However, if they send the same data, they should be independent, or if the program future behavior and the property do not depend on the value of the data, then they can be considered independent. We believe that this solution can helps us get good reductions in many cases in practice.

Liveness property We now assume that the state space of the MPI program that we consider is acyclic. However, UDPOR can work with non-acyclic state spaces by using a cut-off technique [27]. Besides, we have a great advantage since SimGrid is equipped with a state equality tool allowing to check if two application states are equal. Hence, we can extend the prototype to verify non-acyclic state spaces. Besides, like most other studies on POR, our work now only focuses on safety property. One direction to extend UDPOR is to check LTL-X properties (LTL properties do not contain the next operator (X)). We believe that there are promising results to get in this direction, but it will require a lot of efforts.

Parallelization and distribution Another direction that can be tried to get better performance of UDPOR is to parallelize or distribute UDPOR. One simple scheme is that each trace (alternative) can be explored by a thread/process. We will need a scheduler to cleverly assign alternatives to threads/processes and avoid the situation that two threads/process explore the same alternative, but we think that's a promising way.

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas, « Optimal dynamic partial order reduction », *in*: *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, San Diego, CA, USA, January*, 2014.

[2] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey, « Context-Sensitive Dynamic Partial Order Reduction », *in*: *Computer Aided Verification - 29th International Conference, Heidelberg, Germany, July*, 2017.

[3] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio, « Constrained Dynamic Partial Order Reduction », *in*: *30th International Conference on Computer Aided Verification, Oxford, UK, July*, 2018.

[4] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania, « Bounded model checking of software using SMT solvers instead of SAT solvers », *in*: *STTT* 11.*1* (2009).

[5] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas, « Optimal Dynamic Partial Order Reduction with Observers" », *in*: *Tools and Algorithms for the Construction and Analysis of Systems, Thessaloniki, Greece, April*, Springer, 2018.

[6] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, MIT Press, 2008, ISBN: 978-0-262-02649-9.

[7] Armin Biere, « Bounded Model Checking », *in*: *Handbook of Satisfiability*, 2009.

[8] Stanislav Böhm, Ondrej Meca, and Petr Jancar, « State-Space Reduction of Non-deterministically Synchronizing Systems Applicable to Deadlock Detection in MPI », *in*: *Formal Methods - 21st International Symposium, Limassol, Cyprus, November*, 2016.

[9] Henri Casanova, Arnaud Legrand, and Martin Quinson, « SimGrid: A Generic Framework for Large-Scale Distributed Experiments », *in*: *Tenth International Conference on Computer Modeling and Simulation, Cambridge, UK, April*, 2008.

[10] Edmund Melson Clarke and E. Allen Emerson, « Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic », *in*: *25 Years of Model Checking - History, Achievements, Perspectives*, 2008.

[11] Edmund Melson Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith, « Counterexample-Guided Abstraction Refinement », *in*: *Computer Aided Verification, 12th International Conference, CAV, Proceedings, Chicago, IL, USA, July*, 2000.

[12] Edmund Melson Clarke, Orna Grumberg, and David E. Long, « Model Checking and Abstraction », *in*: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, 1992.

[13] Augustin Degomme, Arnaud Legrand, George S. Markomanolis, Martin Quinson, Mark Stillwell, and Frédéric Suter, « Simulating MPI Applications: The SMPI Approach », *in*: *IEEE Transactions on Parallel and Distributed Systems* 28.*8* (2017).

[14] Cormac Flanagan and Patrice Godefroid, « Dynamic partial-order reduction for model checking software », *in*: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Long Beach, California, USA, January*, vol. 40, 2005.

[15] Message Passing Forum, *MPI: A Message-Passing Interface Standard*, tech. rep., Knoxville, TN, USA, 1994.

[16] Patrice Godefroid, « Model Checking for Programming Languages using Verisoft », *in*: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, January*, 1997.

[17] Patrice Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, vol. 1032, Lecture Notes in Computer Science, Springer, 1996, ISBN: 3-540-60761-7.

[18] Patrice Godefroid and Didier Pirottin, « Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract) », *in*: *Proc. Computer Aided Verification, 5th International Conference, CAV, Elounda, Greece, June*, 1993.

[19] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky, « Formal Analysis of MPI-based Parallel Programs », *in*: *Communication of the ACM* 54.*12* (Dec. 2011), ISSN: 0001-0782.

[20] Marion Guthmuller, Gabriel Corona, and Martin Quinson, « System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications », Research Report, 2015.

[21] Anthony Hall, « Seven Myths of Formal Methods », *in*: *IEEE Softw.* 7.*5* (Sept. 1990), ISSN: 0740-7459.

[22] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller, « MUST: A Scalable Approach to Runtime Error Detection in MPI Programs », *in*: *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*, Springer Berlin Heidelberg, 2010, ISBN: 978-3-642-11261-4.

[23] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare, « Dynamic Symbolic Verification of MPI Programs », *in*: *22nd International Symposium on Formal Methods, FM'18, Oxford, UK*, 2018.

[24] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare, « Dynamic Symbolic Verification of MPI Programs », *in*: *22nd International Symposium on Formal Methods, Oxford, UK, July*, 2018.

[25] Roger Kowalewski and Karl Fürlinger, « Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications », *in*: *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Proceedings*, Springer International Publishing, 2016.

[26] Bettina Krammer, K. Bidmon, M.S. Müller, and M.M. Resch, « MARMOT: An MPI analysis and checking tool », *in*: *Advances in Parallel Computing* 13 (2004), Parallel Computing, pp. 493 –500.

[27] César Rodríguez , Marcelo Sousa , Subodh Sharma , Daniel Kroening, « Unfolding-based Partial Order Reduction », *in*: *26th International Conference on Concurrency Theory, Madrid, Spain, September*, 2015.

[28] Leslie Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley, 2002, ISBN: 0-3211-4306-X.

[29] Kenneth McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993, ISBN: 0792393805.

[30] Willem Visser, Peter C. Mehlitz, « Model Checking Programs with Java PathFinder », *in*: *Proceedings of Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August*, 2005.

[31] Stephan Merz, Martin Quinson, and Cristian Rosa, « SimGrid MC: Verification Support for a Multi-API Simulation Platform », *in*: *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS, and 31st IFIP WG 6.1 International Conference, FORTE, Proceedings, Reykjavik, Iceland, June*, 2011.

[32] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill, « CMC: A Pragmatic Approach to Model Checking Real Code », *in*: *5th Symposium on Operating System Design and Implementation,Boston, Massachusetts, USA, December*, 2002.

[33] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu, « Finding and Reproducing Heisenbugs in Concurrent Programs », *in*: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008.

[34] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci, « Quasi-Optimal Partial Order Reduction », *in*: *30th International Conference on Computer Aided Verification, Oxford, UK, July*, 2018.

[35] Robert Palmer, Steve Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M Kirby, « Gauss: A Framework for Verifying Scientific Computing Software », *in*: *Electronic Notes in Theoretical Computer Science* 144.*3* (2006).

[36] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby, « Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs », *in*: *Proceedings of the ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, London, United Kingdom, 2007.

[37] Doron A. Peled, « All from One, One for All: on Model Checking Using Representatives », *in*: *Computer Aided Verification, 5th International Conference, CAV, Proceedings, Elounda, Greece, June*, 1993.

[38] The Anh Pham, Thierry Jéron, and Martin Quinson, « Unfolding-Based Dynamic Partial Order Reduction of Asynchronous Distributed Programs », *in*: *Formal Techniques for Distributed Objects, Components, and Systems - 39th IFIP WG 6.1 International Conference, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, Kongens Lyngby, Denmark, June*, 2019.

[39] The Anh Pham, Thierry Jéron, and Martin Quinson, « Verifying MPI Applications with SimGridMC », *in*: *Proceedings of the 1st International Workshop on Software Correctness for HPC Applications, Denver, CO, USA, November*, 2017.

[40] Jean-Pierre Queille and Joseph Sifakis, « Specification and verification of concurrent systems in CESAR », *in*: *International Symposium on Programming, 5th Colloquium, Proceedings,Torino, Italy, April*, 1982.

[41] Cristian Daniel Rosa, Stephan Merz, and Martin Quinson, « A Simple Model of Communication APIs - Application to Dynamic Partial Order Reduction », *in*: *10th International Workshop on Automated Verification of Critical Systems, Düsseldorf, Germany, Septembre*, 2010.

[42] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou, « PARCOACH: Combining static and dynamic validation of MPI collective communications », *in*: *International Journal of High Performance Computing Applications* (2014).

[43] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky, « A Sound Reduction of Persistent-Sets for Deadlock Detection in MPI Applications », *in*: *Formal Methods: Foundations and Applications, Natal, Brazil, September*, 2012, pp. 194–209.

[44] Stephen F. Siegel, « Model Checking Nonblocking MPI Programs" », *in*: *8th International Conference on Verification, Model Checking, and Abstract Interpretation, Nice, France, January*, Springer Berlin Heidelberg, 2007.

[45] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers, « CIVL: The Concurrency Intermediate Verification Language », *in*: *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, Texas, USA, November, ACM, 2015.

[46] Stephen F. Siegel and Timothy K. Zirkel, « Automatic Formal Verification of MPI-based Parallel Programs », *in*: *SIGPLAN Not.* (2011).

[47] C.R. Spitzer, *Digital avionics handbook: development and implementation. Avionics*, CRC Press, 2007, ISBN: 9780849384417.

[48] Maarten van Steen and Andrew S. Tanenbaum, « A brief introduction to distributed systems », *in*: *Computing* 98.*10* (2016), ISSN: 1436-5057.

[49] Antti Valmari, « Stubborn sets for reduced state space generation », *in*: *International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June*, 1989.

[50] Anh Vo, S. Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky, « A Scalable and Distributed Dynamic Formal Verifier for MPI Programs », *in*: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA,* IEEE Computer Society, 2010.

[51] Anh Vo, Sarvani S. Vakkalanka, Michael Delisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur, « Formal Verification of Practical MPI Programs », *in*: *ACM SIGPLAN Notices* 44.*4* (2009).

[52] Jim Woodcock and Jim Davies, *Using Z: Specification, Refinement, and Proof*, Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996, ISBN: 0-13-948472-8.

[53] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald, « Formal Methods: Practice and Experience », *in*: *ACM Comput. Surv.* 41.*4* (2009).

[54] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou, « MODIST1: Transparent Model Checking of Unmodified Distributed Systems », *in*: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, April*, 2009.

[55] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby, « Distributed dynamic partial order reduction », *in*: *STTT* 12.*2* (2010), pp. 113–122.