Formal Verification of Distributed Algorithms using PlusCal-2

Sabina Akhtar

UNIVERSITÉ DE LORRAINE

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

Vérification Formelle d'Algorithmes Distribués en PlusCal-2

(Verification of Distributed Algorithms using PlusCal-2)

THÈSE

présentée et soutenue publiquement le 9 Mai 2012

pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Sabina AKHTAR

Composition du jury

Directeur de thèse :	Stephan MERZ – DR INRIA
Rapporteurs :	Béatrice BERARD – Professeur, LIP6 Paris Philippe QUEINNEC – MCF HDR, ENSEEIHT Toulouse
Examinateurs :	Hassan MOUTASSIR – Professeur, LIFC UFR ST Besancon Isabelle CHRISMENT – Professeur, Université de Lorraine Martin QUINSON – MCF, Université de Lorraine

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

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thloria.

À mon cœur, Rayaan et ma vie, Ehtesham.

Acknowledgments

The work presented in this thesis would not have been possible without the help of my thesis advisors Stephan Merz and Martin Quinson. I am sincerely thankful to them for being highly supportive and for their continuous guidance and wisdom throughout this journey.

I am very fortunate to have had opportunity to be part of the team VERIDIS at LORIA and I would like to thank my colleagues for all the time we spent together during these past years. I would also like to thank my parents, and my husband for their prayers and support.

Finally, I would like to thank the Higher Education Commission (HEC), Pakistan for sponsoring my work.

Abstract

Designing sound algorithms for concurrent and distributed systems is subtle and challenging. These systems are prone to deadlocks and race conditions, which occur in particular interleavings of process actions and are therefore hard to reproduce. It is often nontrivial to precisely state the properties that are expected of an algorithm and the assumptions on the environment under which these properties should hold. Formal verification is a key technique to model the system and its properties and then perform verification by means of model checking.

Formal languages like TLA$^+$ have the ability to describe complicated algorithms quite concisely, but algorithm designers often find it difficult to model an algorithm in the form of formulas. In this thesis, we present PlusCal-2 that aims at being similar to pseudo-code while being formally verifiable. PlusCal-2 improves upon Lamport's PlusCal algorithm language by lifting some of its restrictions and adding new constructs. Our language is intended for describing algorithms at a high level of abstraction. It resembles familiar pseudo-code but is quite expressive and has a formal semantics. Finite instances of algorithms described in PlusCal-2 can be verified through the TLC model checker. The second contribution presented in this thesis is a study of partial-order reduction methods using conditional and constant dependency relation.

To compute conditional dependency for PlusCal-2 algorithms, we exploit their locality information and present them in the form of independence predicates. We also propose an adaptation of a dynamic partial-order reduction algorithm for a variant of the TLC model checker. As an alternative to partial-order reduction based on conditional dependency, we also describe a variant of a static partial-order reduction algorithm for the TLC model checker that relies on constant dependency relation. We also present our results for the experiments along with the proof of correctness.

Contents

Introduction (en français)

Contents

1 Contexte scientifique

Les systèmes distribués et concurrents sont dans le courant dominant de la technologie de l'information [Lynch 1996] et la conception et la mise en œuvre de ces systèmes est une tâche très difficile. Ils sont source d'erreurs et peuvent avoir des problèmes comme les blocages, les conditions de course, etc qui pourraient avoir un impact dramatique sur la vie humaine, l'environnement ou des actifs importants. Ces systèmes doivent satisfaire une variété de qualités spécifiques, y compris la disponibilité, la sécurité, la fiabilité et la sécurité. Ainsi, ils doivent être vérifiées de manière extensive avant leur déploiement.

Il y a beaucoup de techniques formelles et informelles qui ont été proposées et sont utilisées pour la vérification de systèmes répartis, dont la vérification algorithmique qui est une technique formelle dont son nom indique que la vérification elle-même est réalisée algorithmiquement, contrairement à une vérification manuelle ou interactive. La vérification algorithmique nécessite un modèle du système et les spécifications du système, écrits dans un langage mathématique précis. Il faut également l'ensemble de propriétés, écrites sous la forme de formules qui sont nécessaires pour la vérification du système. Cette méthode est également connue sous le nom model-checking, qui vérifie qu'un modèle donné du système satisfait certaines propriétés [Clarke 1996a].

Le modèle du système est généralement représenté par un graphe de transitions d'états appelé une structure de Kripke [Clarke 1996a, Baier 2008]. Il est composé d'un ensemble d'états, un ensemble de transitions entre les états, et une fonction qui associe à chaque état un ensemble de propriétés vraies dans l'état donné. Une structure de Kripke peut être formellement définie comme suit:

Definition 1. *Une structure de Kripke M sur un ensemble AP de propositions est un quadruplet M =(S, I, R, L) où*

1. *S est un ensemble fini d'états.*

2. *I est l'ensemble des états initiaux.*

3. *$R \subseteq S \times S$ est une relation de transition, supposée totale.*

4. *$L{:}S \to 2^{AP}$ est une fonction qui associe à chaque état l'ensemble des propositions atomiques vraies dans cet état.*

Dans la littérature, une structure de Kripke est également désignée comme un système de transitions d'états. Il peut s'agir d'une structure déterministe où tous les états futurs sont totalement prévisibles ou d'une structure non-déterministe où une transition peut conduire à l'un parmi plusieurs etats successeurs possibles. Le non-déterminisme est un concept important pour les systèmes de transitions d'états pour modéliser le comportement imprévisible du système. Par exemple, une transition qui représente une instruction ou un bloc d'instructions dans un algorithme peut être composée d'une instruction if-then-else avec plus d'une condition. Ainsi, une telle transition peut conduire à un choix d'états successeurs possibles qui ne peut être résolu au moment de l'initialisation.

Une fois le système modélisé, on peut décrire les spécifications pour le modèle du système en utilisant un langage de spécification comme TLA$^+$. TLA$^+$ [Lamport 2002] est un langage de spécification complet basé sur la logique temporelle d'actions TLA [Lamport 1994] et la théorie des ensembles de Zermelo-Fraenkel. Il a été développé par Leslie Lamport, et il est utilisé pour spécifier le modèle du système.

TLA$^+$ n'est pas un langage de programmation mais plutôt un formalisme de spécification. Il requiert des connaissances sur la théorie mathématique des ensembles pour modéliser les systèmes. Dans la pratique, les utilisateurs qui écrivent des programmes ou des algorithmes n'ont pas de connaissances approfondies de ce formalisme. Ainsi, il devient difficile pour les utilisateurs d'utiliser ce langage. Afin de rendre TLA$^+$ accessible aux concepteurs d'algorithmes, Leslie Lamport a proposé PlusCal [Lamport 2009] qui est un langage algorithmique de haut niveau pour générer du code TLA$^+$ pour des algorithmes parallèles et distribués . PlusCal fournit des instructions très simples pour exprimer des algorithmes non-déterministes. PlusCal combine cinq caractéristiques importantes [Lamport 2006a]: des constructions simples de programmes traditionnels, des expressions extrêmement puissantes, du non-déterminisme, un moyen pratique pour décrire le grain de l'atomicité et le model-checking.

Le langage PlusCal fournit une plate-forme pour les concepteurs d'algorithmes pour générer des modèles pour leurs algorithmes en langage de spécification TLA$^+$. Ensuite, ils peuvent bénéficier de l'architecture sous-jacente qui est le langage de spécification TLA$^+$ avec son model-checker TLC [Yu 1999]. Le model-checking est une technique de vérification algorithmique qui décide si un modèle d'un système répond à un ensemble de propriétés caractérisant la correction du système, qui sont représentées par des formules logiques. Dans la pratique, les outils de model-checking exécutent les spécifications données et vérifient les propriétés spécifiées pour la correction des spécifications. Il existe diverses autres techniques, y compris

des techniques de démonstration de théorèmes qui fournissent une démonstration mathématique pour montrer la correction du système.

L'idée de base du model-checking est d'explorer toutes les exécutions possibles à partir de l'ensemble initial d'états pour le système donné. Cependant, le problème bien connu de l'explosion combinatoire de l'espace d'états limite la taille des instances qui peuvent être vérifiées de manière efficace. Dans la littérature, diverses méthodes ont été étudiées pour pallier à l'effet de l'explosion de l'espace d'états y compris la représentation symbolique de l'espace d'états, des stratégies efficaces de gestion de la mémoire, les méthodes de réduction par la symétrie, les méthodes de réduction par ordre partiel.

Dans cette thèse, nous nous sommes concentrés sur la performance des méthodes de réduction par ordre partiel pour le model-checker TLC. L'idée principale de la réduction par ordre partiel est de restreindre l'exploration de l'espace d'états tels que les entrelacements redondants de transitions sont évités, d'où la préservation de la correction de la vérification. Cette méthode repose sur l'indépendance et la commutativité de transitions de différents processus.

2 Motivation pour le langage PlusCal-2

PlusCal [Lamport 2006b] est un langage de haut niveau pour décrire des algorithmes parallèles et répartis. Il est destiné aux concepteurs d'algorithmes qui ne veulent pas maîtriser la complexité de la rédaction de spécifications TLA$^+$ à part entière, mais qui souhaitent néanmoins tirer parti des capacités de vérification du model-checker TLC.

Nous avons modélisé différents algorithmes dans le langage PlusCal notamment l'algorithme proposé par Naimi et Trehel [Naimi 1996]. Il s'agit d'un algorithme distribué d'exclusion mutuelle qui maintient deux structures de données distribuées: une liste des processus qui sont en attente pour l'accès à la section critique, et un arbre de processus dont la racine est le processus à la fin de la file d'attente (ou le dernier processus à accéder à sa section critique si la file d'attente est vide). Son modèle dans le langage PlusCal est montré en annexe 1.1.1 et les spécifications TLA$^+$ correspondantes sont présentées en annexe 1.1.2.

Dans cette section, nous décrivons un certain nombre de problèmes que nous avons remarqué lors de l'utilisation PlusCal pour la modélisation des algorithmes parallèles et répartis.

Besoin de comprendre TLA$^+$ et la compilation. Dans la pratique, PlusCal ne peut être utilisé sans une bonne compréhension du formalisme TLA$^+$ et même de la traduction produite par le compilateur PlusCal. Par exemple, pour définir les propriétés de l'algorithme de Naimi et Trehel l'utilisateur doit comprendre les spécifications TLA$^+$ générées, et écrire les propriétés en termes des variables TLA$^+$ introduites par le compilateur. Bien que le compilateur essaie de préserver les noms de variables, ceci n'est pas possible si des variables du même nom sont déclarées

dans différentes procédures. Les variables locales des processus sont converties en des tableaux de TLA$^+$, et l'utilisateur doit être conscient de ceci lors de l'annotation du modèle TLA$^+$. Les propriétés d'équité, nécessaires à la vérification des propriétés de vivacité, doivent être définies en termes des actions TLA$^+$ générées par le compilateur. Enfin, l'utilisateur doit fournir un fichier de configuration au model-checker TLC, qui définit l'instance finie qui sera vérifiée.

Hiérarchie plate de processus et absence de règles de portée. Les processus de PlusCal doivent tous être déclarés au plus haut niveau et ne peuvent être imbriqués. Cependant, de nombreux algorithmes sont plus naturellement exprimés à l'aide d'une hiérarchie de processus. Cela est particulièrement vrai pour les algorithmes distribués, où plusieurs threads (partagant de la mémoire locale) peuvent coexister dans des nœuds physiquement distribués qui communiquent de façon asynchrone sur un réseau. Par exemple, l'algorithme de Naimi-Trehel aurait été représenté facilement avec des processus imbriqués. On pourrait introduire des sous-processus pour le processus Site, l'un pour gérer les messages reçus et l'autre pour envoyer une demande pour l'utilisation de la section critique. En outre, l'utilisateur est responsable de l'attribution des identités des processus et de s'assurer qu'elles soient uniques. Bien que ce point est un ennui mineur, il constitue une source potentielle d'erreurs qui sont facilement négligés.

Une question connexe est l'absence de règles de portée dans PlusCal. Même si des variables locales peuvent être déclarées au sein des processus comme nous l'avons fait pour le processus Site de l'algorithme de Naimi-Trehel, ces variables sont en fait librement accessibles pour la lecture et l'écriture par des processus ou procédures différents. Au-delà d'être très probable une limitation lors de la modélisation des algorithmes et une cause d'erreurs, le manque d'une hiérarchie de processus et des variables de portée locale rend également très difficile la mise en œuvre d'optimisations en matière de vérification, telles que des réductions basées sur les ordres partiels.

Spécification limitée d'atomicité. En modélisant des algorithmes concurrents et distribués, il est important de spécifier "le grain d'atomicité", c'est-à-dire les blocs d'instructions que l'on peut considérer comme étant exécutés sans intercaler avec les instructions d'autres processus. Tandis que l'atomicité à trop gros grain peut cacher les erreurs qui résultent d'entrelacements inattendus, in modèle à grain d'atomicité trop fin donne lieu à l'explosion combinatoire de l'espace d'états lors de la vérificaiton et est inutile tant que seulement le calcul local est impliqué. PlusCal utilise une convention simple mais puissante : l'utilisateur décore des instructions avec des étiquettes pour spécifier où l'intercalation peut intervenir de manière atomique. Toutes les instructions entre deux étiquettes sont exécutées. Cependant, les étiquettes servent aussi à d'autres buts pour la compilation et PlusCal impose un certain nombre de règles d'étiquetage qui limitent la liberté du spécificateur de déterminer ce qui doit être considéré comme groupes d'instructions atomiques.

Limitations techniques. Le compilateur PlusCal doit être simple afin que les utilisateurs puissent lire le modèle TLA$^+$ produit par le compilateur PlusCal, comme nous l'avons mentionné ci-dessus. La simplicité du compilateur impose certaines limites contre-intuitives au langage PlusCal. Par exemple, bien que des ensembles soient la construction de base pour représenter des données, PlusCal ne contient pas de construction primitive pour itérer sur tous les éléments d'un ensemble. Le programmeur doit introduire une variable auxiliaire pour l'itération et garder trace des éléments qui ont déjà été traités. Sans une attention particulière, ces variables auxiliaires peuvent aggraver l'explosion de l'espace d'états pendant la vérification du modèle. Une autre restriction technique est la règle de PlusCal qu'il ne peut y avoir qu'une seule attribution par variable au sein d'un bloc atomique.

Nous avons d'abord essayé d'étendre le compilateur existant de PlusCal pour contourner ces limitations. Cependant, il est vite apparu qu'il était nécessaire de redéfinir le langage et la reconception du compilateur afin de surmonter les lacunes les plus graves. Alors que nous avons maintenu les idées de base et les structures de PlusCal dans notre nouvel langage PlusCal-2, nous avons privilégié une conception propre sur la compatibilité ascendante en cas de conflit.

3 Motivation pour les méthodes de réduction

Nous avons modélisé différents algorithmes dans PlusCal-2, y compris l'algorithme pour l'élection d'un leader qui a été proposé par Dolev, Klawe, et Rodeh [Dolev 1982]. Nous avons décrit l'algorithme dans le langage PlusCal-2 comme indiqué dans l'annexe 1.2, pour générer les spécifications en TLA$^+$ et le vérifier à l'aide de TLC.

Le compilateur PlusCal-2 a généré avec succès les spécifications en TLA$^+$. Ensuite, nous les avons transmis au model-checker TLC pour vérifier certaines propriétés. Le tableau 3 montre les résultats de la vérification pour différents nombres de processus. Il montre clairement que le model-checking et confronté à un problème d'explosion d'espace d'états avec un nombre croissant de processus. Cette question nous a motivé à chercher une technique qui pourrait réduire ce problème. Beaucoup de recherches ont déjà été menées pour s'attaquer au problème d'explosion d'espace d'états, par exemple, les méthodes de réduction par ordre partiel qui utilisent la relation d'indépendance entre les actions atomiques du système. Cette idée nous a inspiré d'utiliser notre compilateur PlusCal-2 pour générer ces relations d 'indépendance pour les actions dans les spécifications TLA$^+$.

TLC utilise la méthode de recherche en largeur d'abord pour explorer tous les états possibles ce qui restreint l'utilisation de la relation d'indépendance conditionnelle. Ainsi, nous proposons une variante du model-checker TLC avec une adaptation de l'algorithme de réduction par ordre partiel dynamique dû à Cormac Flanagan et Patrice Godefroid. L'utilisation de la méthode de recherche en largeur d'abord n'est pas compatible qu'avec une réduction par ordre partiel basée sur une relation de dépendance constante. Ainsi, dans cette thèse, nous présentons également une version étendue du model-checker TLC qui prend en charge une méthode de réduc-

# Processus	Temps (secondes)	États Totaux	États Distincts
4	0.2	205	95
8	2.3	31289	7121
10	22.9	352426	63986
12	349.2	3811181	575747

Table 1: Les résultats de model-checking pour Leader election algorithm

tion par ordre partiel statique, et nous évaluons ainsi que les résultats pour nos exemples.

4 Objectifs de thèse

Les objectifs principaux de cette thèse peuvent être résumés comme suit:

1. Le premier objectif est de fournir aux concepteurs d'algorithmes un langage algorithmique qu'ils peuvent utiliser pour spécifier ou modéliser les algorithmes de vérification. Nous avons décidé d'utiliser PlusCal comme une motivation pour notre travail, car ce langage ressemble à d'autres langages algorithmiques. Les limites que nous avons mentionnées dans la section 1.2 enfreignent son utilisation pour les concepteurs d'algorithmes. L'utilisateur doit être en mesure d'écrire des algorithmes dans PlusCal-2 sans avoir à apprendre un langage mathématique, mais en utilisant la puissance de l'architecture sous-jacente qui est le langage TLA$^+$ et le model-checker tlc. Ainsi, dans cette thèse, nous présentons le langage PlusCal-2 qui enlève les limites mentionnées, tout en préservant l'expressivité et le non-déterminisme dans le langage et étant facilement accessible pour le concepteur d'algorithmes.

2. Le deuxième objectif est de lutter contre le problème d'explosion de l'espace d'états en faisant appel à la méthode de réduction par ordre partiel et la rendant accessible aux utilisateurs de PlusCal-2 et tlc. La première contribution pour atteindre cet objectif est de produire des prédicats d'indépendance conditionnelle pour les algorithmes écrits en langage PlusCal-2. Ces prédicats peuvent être utilisés dans une technique de réduction afin de résoudre le problème d'explosion d'états dans tlc. Ainsi, nous proposons une plateforme qui est tlc avec la méthode de recherche en profondeur d'abord et la technique de réduction par ordre partiel dynamique pour résoudre le problème d'explosion de l'espace d'états. La deuxième contribution est un model-checker étendu tlc qui prend en charge la méthode de réduction par ordre partiel de Gerard Holzmann et Doron Peled, adaptée pour la recherche en largeur d'abord dans tlc.

5 Structure de Thèse

Cette thèse est organisée comme suit:

- Le chapitre 2 présente le contexte de notre travail qui comprend une description des systèmes distribués et concurrents. Ensuite, nous discutons brièvement le langage PlusCal conçu par Leslie Lamport, le langage de spécification TLA$^+$ et le model-checker TLC. Ensuite, nous analysons en détail la méthode de réduction par ordre partiel qui comprend l'algorithme de réduction par ordre partiel, le concept de l'indépendance et de l'invisibilité, et les techniques pour le calcul du sous-ensemble des transitions à explorer à un état donné.

- Le chapitre 3 introduit le nouveau langage PlusCal-2 et son compilateur. Il présente le langage PlusCal-2 en détail, y compris sa syntaxe, l'organisation d'un algorithme et une discussion sur la méthode de l'écriture d'un algorithme dans le langage PlusCal-2. Ensuite, il décrit le processus de compilation du langage PlusCal-2 vers une spécification TLA$^+$. Enfin, il explique comment le model-checker TLC est utilisé pour vérifier les spécifications produit par le compilateur PlusCal-2.

- Le chapitre 4 définit les prédicats d'indépendance pour les algorithmes en PlusCal-2 qui garantissent que deux actions sont indépendantes à un état donné. Puis, il présente l'extension au compilateur PlusCal-2 qui extrait ces prédicats indépendance. Enfin, il présente la plate-forme proposée pour les prédicats de l'indépendance qui est TLC avec la méthode de recherche en profondeur d'abord et la méthode de réduction par ordre partiel dynamique fondée sur les relations d'indépendance conditionnelles.

- Le chapitre 5 décrit l'adaptation de la méthode de réduction par ordre partiel de Holzmann et Peled pour le model-checker TLC. Il examine brièvement la méthode de réduction par ordre partiel. Puis il présente les détails de l'intégration de cette méthode en TLC avec une validation expérimentale. Enfin il détaille la preuve de correction de l'algorithme adapté.

- Enfin, dans les conclusions, nous discutons des contributions de cette thèse et des travaux proposés à venir.

Chapter 1

Introduction

Contents

1.1 Background

Distributed and concurrent systems are in the mainstream of information technology [Lynch 1996] and the design and implementation of these systems is a highly challenging task. They are error prone and can have problems like deadlocks, race conditions, etc. that could have a dramatic impact on human life, the environment or significant assets. These systems must satisfy a variety of specific qualities including availability, security, reliability and safety. Thus, they should be verified extensively before deployment.

There are many formal and informal techniques that have been proposed and are being used for their verification including the algorithmic verification which is a formal technique that stresses that the verification itself is performed algorithmically, in contrast to manual or interactive verification. Algorithmic verification requires a model of the system and the specifications of the system written in a precise mathematical language. It also requires the set of properties written in the form of formulas that are required for the verification of the system. This method is also known as Model Checking, which verifies a given model of the system to satisfy some properties [Clarke 1996a].

The model of the system is usually represented using a state transition graph called a Kripke structure [Clarke 1996a, Baier 2008]. It is composed of a set of states, a set of transitions between the states, and a function that labels each state with a set of properties that hold in the given state. A Kripke structure can formally be defined as follows:

Definition 2. *A Kripke structure M over a set of propositions AP is a four-tuple M =(S, I, R, L) where*

1. *S is a finite set of states.*
2. *I is the set of initial states.*
3. *$R \subseteq S \times S$ is a transition relation that is assumed to be total.*
4. *$L{:}S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

In the literature, a kripke structure is also referred as a state transition system. It can either be deterministic where all the future states are completely predictable or it can be non-deterministic where a transition can lead to one of the possible successor states. Non-determinism is an important concept in state transition systems to model the unpredictable behavior of the system. For example, a transition that represents a statement or a block of statements in an algorithm can be composed of an if-then-else statement with more than one conditions. Thus, such a transition can lead to one of the possible successor states that cannot be determined at the time of initialization.

Once the system is modeled, then we can describe the specifications for the model of the system using a specification language like TLA$^+$. TLA$^+$ [Lamport 2002] is a complete specification language based on the temporal logic of actions TLA [Lamport 1994] and Zermelo-Fraenkel set theory. It was developed by Leslie Lamport and it is used to specify the model of the system.

TLA$^+$ is not a programming language but rather a specification formalism. It requires knowledge about the mathematical set theory to model the systems. In practice, the users who write programs or algorithms have no background of this formalism. Thus, it becomes difficult for the users to make use of this language. In order to make it easy for the algorithm designers to use TLA$^+$, Leslie Lamport proposed PlusCal [Lamport 2009] which is a high level algorithmic language to generate TLA$^+$ code for concurrent and distributed algorithms. It provides very simple statements to express non-deterministic algorithms. PlusCal combines five important features [Lamport 2006a]: simple conventional program constructs, extremely powerful expressions, nondeterminism, a convenient way to describe the grain of atomicity, and model checking.

The PlusCal language provides the platform for the algorithm designers to generate the models for their algorithms in TLA$^+$ specification language. Then they can make use of underlying architecture that is TLA$^+$ specification language along with its supported TLC model checker. The TLC model checker [Yu 1999] provides a platform to model check the specifications written in TLA$^+$ language. Model checking is an algorithmic verification technique that determines whether a model of a system meets the required set of properties that are represented as logical formulas. In practice, the model checking tools execute the given specifications and verify the specified properties for the correctness of the specifications. There are various other techniques including theorem proving techniques that provides a mathematical demonstration to show the correctness of the system.

The basic idea behind model checking is to explore all the possible executions from the initial set of states for the given system. However, the well-known state space explosion problem limits the size of instances that can be verified effectively. In the literature, various methods have been studied to reduce the effect of state space explosion including symbolic state space representation, efficient memory management strategies, symmetry reduction methods, partial-order reduction methods.

In this thesis, we focused on the performance of partial-order reduction methods for the TLC model checker. The main idea of partial-order reduction is to restrict the state-space exploration such that redundant interleavings of transitions are avoided, hence preserving soundness of the verification. This method relies on the independence and commutativity of the transitions from different processes.

1.2 Motivations for PlusCal-2

PlusCal [Lamport 2006b] is a high-level language for describing concurrent and distributed algorithms. It is targeted towards algorithm designers who do not want to master the complexity of writing full-fledged TLA$^+$ specifications, but nevertheless want to take advantage of the verification capabilities of the TLC model checker.

We modeled various algorithms in PlusCal language including Naimi-Trehel algorithm that was proposed by Naimi and Trehel in [Naimi 1996]. It is a distributed algorithm for mutual exclusion that maintains two distributed data structures: a list of processes that are waiting for access to the critical section, and a tree of process whose root is the process at the end of the waiting queue (or the process who last accessed its critical section if the queue is empty). Its model in PlusCal language is shown in appendix 1.1.1 and its corresponding TLA$^+$ specifications are shown in appendix 1.1.2.

In this section, we describe a number of problems that we noticed when using PlusCal for modeling concurrent and distributed algorithms.

Need to understand TLA$^+$ and the compilation. In practice, PlusCal cannot be used without a good understanding of the TLA$^+$ formalism and even of the translation generated by the PlusCal compiler. For example, to define the properties for Naimi-Trehel algorithm the user must understand the generated TLA$^+$ specifications, and write the properties in terms of the TLA$^+$ variables introduced by the compiler. Although the compiler tries to preserve variable names, this is impossible if variables of the same name are declared in different procedures. Local variables of processes are translated to arrays in TLA$^+$, and the user must be aware of this when annotating the TLA$^+$ model. Fairness properties, necessary for the verification of liveness properties, must be defined in terms of the TLA$^+$ actions generated by the compiler. Finally, the user has to provide a configuration file to the TLC model checker, which defines the finite instance to be verified.

Flat process hierarchy and lack of scoping. PlusCal processes must all be declared at top level and cannot be nested. However, many algorithms are more naturally expressed using hierarchies of processes. This is particularly true for distributed algorithms, where several threads (sharing local memory) may coexist within physically distributed nodes that communicate asynchronously over a network. For example, the Naimi-Trehel algorithm would have been represented easily with nested processes. We could introduce subprocesses for the process Site, one to handle the received messages and another process to send a request for using the critical section. Moreover, the user is responsible for assigning identities to processes and ensuring that they are unique. While this is a minor annoyance, it provides a potential source for errors that are easily overlooked.

A related issue is the lack of scoping rules in PlusCal. Although variables may be declared local to processes as we have in Naimi-Trehel algorithm for the process Site, they are in fact freely accessible for reading and writing in different processes or procedures. Beyond being a limitation when modeling algorithms and being an all too likely cause for errors, the lack of a proper hierarchy of processes and of scoped local variables also makes it very difficult to implement optimizations in verification, such as partial-order reduction.

Restricted specification of atomicity. When modeling concurrent and distributed algorithms, it is important to specify the "grain of atomicity", i.e. blocks of statements that can be considered as being executed without interleaving with statements of other processes. Whereas too coarse-grained atomicity may hide errors that arise in the implementation due to unexpected interleavings, too fine-grained atomicity causes state space explosion in verification and is unnecessary as long as only local computation is involved. PlusCal uses a simple but powerful convention: the user decorates statements with labels to specify where interleaving may occur. All statements between two labels are executed atomically. However, labels also serve other purposes for compilation, and PlusCal imposes a number of labeling rules that restrict the freedom of the specifier to determine which groups of statements should be considered atomic.

Technical limitations. The PlusCal compiler must be kept simple so that users can read the TLA$^+$ model produced by the PlusCal compiler, as mentioned above. The simplicity of the compiler imposes some unintuitive limitations of the PlusCal language. For example, although sets are the basic construct for representing data, PlusCal does not contain a primitive construct for iterating over all elements of a set. The programmer has to introduce an auxiliary variable for iteration and keep track of the elements that have already been handled. Without special care, these auxiliary variables can aggravate state space explosion during model checking. Another technical restriction is PlusCal's rule that there may be only one assignment per variable within an atomic step.

We first tried to extend the existing PlusCal compiler to get around these lim-

# Processes	Time(seconds)	Total States	Distinct States
4	0.2	205	95
8	2.3	31289	7121
10	22.9	352426	63986
12	349.2	3811181	575747

Table 1.1: Model checking results for Leader election algorithm.

itations. However, it quickly became clear that it was necessary to redefine the language and redesign the compiler in order to overcome the more serious deficiencies. While we maintained the basic ideas and constructs of PlusCal in our new language PlusCal-2, we favored a clean design over backward compatibility in case of conflict.

1.3 Motivation for reduction methods

We modeled various algorithms in PlusCal-2 including the Leader election algorithm that was proposed by Dolev, Klawe, and Rodeh [Dolev 1982] for electing a leader in a unidirectional ring. We described the algorithm in PlusCal-2 language as shown in appendix 1.2, to generate the TLA$^+$ specifications and to model check them using TLC model checker.

The PlusCal-2 compiler successfully generated the required TLA$^+$ specifications. Then, we passed them to TLC model checker for verifying some properties. Table 1.3 shows the results of model checking for different number of processes. This clearly shows that model checking results in a state space explosion problem with an increase in number of processes.This issue motivated us to look for a technique that could reduce this problem. In history, a lot of research has been carried out to tackle the state space explosion problem, e.g., partial-order reduction methods that used the independence relation from the specifications of the system. This idea inspired us of using our PlusCal-2 compiler to generate these independence relations for the actions in TLA$^+$ specifications.

TLC uses breadth-first search method to explore all the possible states which restricts the use of conditional independence relation. Thus, we propose a variant of TLC model checker along with an adaptation of dynamic partial-order reduction algorithm by Cormac Flanagan and Patrice Godefroid. As TLC supports breadth-first search method, it can only use constant dependency relation in a partial-order reduction algorithm. Thus, in this thesis we also present an extended TLC model checker that supports static partial-order reduction method along with the results for our examples.

1.4 Thesis objectives

The main objectives of this thesis are as follows:

1. The first objective is to provide the algorithm designers an algorithmic language that they can use to specify or model the algorithms for verification. We decided to use PlusCal as a motivation for our work because it was similar to other algorithmic languages. The limitations that we mentioned in the section 1.2 restricted its use for algorithm designers. The user must be able to write algorithms in PlusCal-2 without having to learn a mathematical language but using the power of the underlying architecture that is TLA$^+$ language and the model checker TLC. Thus, in this thesis, we present the language PlusCal-2 that removes all the limitations while preserving the expressiveness of non-determinism in the language and thus making it easily accessible for the algorithm designer.

2. The second objective is to combat state space explosion problem by making partial-order reduction method accessible to the users of PlusCal-2 and TLC. The first contribution to achieve this objective is to produce conditional independence predicates from the algorithms written in PlusCal-2 language. These conditional predicates can further be used in a partial-order reduction technique to address the state space explosion problem in TLC. Thus, we propose a platform that is TLC with depth first search method along with the dynamic partial-order reduction technique to address the state space explosion problem. The second contribution is an extended TLC model checker that supports Holzmann's partial-order reduction method adapted for breadth first search in TLC.

1.5 Thesis Structure

This thesis is organized as follows:

- Chapter 2 presents the background of our work that includes description of distributed and concurrent systems. Then, we briefly discuss the original PlusCal language by Leslie Lamport, TLA$^+$ specification language and the TLC model checker. Then, we discuss in detail the partial-order reduction method that includes general partial-order reduction algorithm, the concept of independence and invisibility, and the techniques for computing subset of transitions at a given state.

- Chapter 3 introduces the new PlusCal-2 language and its compiler. It presents the PlusCal-2 language in detail including its syntax, organization of an algorithm and a discussion about the method of writing an algorithm in PlusCal-2 language. Then it describes the process of compilation from PlusCal-2 language to TLA$^+$ specifications. Finally, it explains how the TLC model checker is used to model check the resulting specifications.

- Chapter 4 defines the independence predicates for PlusCal-2 algorithms ensuring that two actions are independent at any given state satisfying the pred-

icate. Then it discusses the extension for PlusCal-2 compiler that extracts these independence predicates. Finally it presents the proposed platform for the PlusCal-2 independence predicates that is TLC with depth first search method along with the dynamic partial-order reduction method supporting conditional independence relations.

- Chapter 5 describes the adaptation of Holzmann's partial-order reduction method for TLC model checker. It briefly discusses the original Holzmann's method for partial-order reduction. Then it presents the integration details of this method for TLC along with the examples and their results. Finally it details the correctness proof for the adapted algorithm.

- Finally, in the conclusions, we discuss the contributions of this thesis and the proposed future work.

Chapter 2

State of the art

Contents

2.1 Distributed and concurrent systems

> You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.
>
> Leslie Lamport [Anderson 2001]

Distributed and concurrent systems are used in a wide range of domains and environments. A distributed system consists of a collection of independent computers, connected through an infrastructure that connects different entities [Andrews 2000, Lynch 1996]. This system enables computers to coordinate their activities and tasks and to share the resources of the entire distributed system. It should be perceived as a single, integrated computing facility by the user. This kind of system provides lots of advantages including the remote resource connection with scalability and openness. The openness refers to the availability of each component to continually interact with other hardware. And by scalability, we mean that the system could easily be extended in order to integrate new components, users and other resources in the system. Thus, a distributed system can grow in size and can also become

more powerful using the combined capabilities of the distributed components, than combinations of independent systems. The distributed system can be built but it's not easy to guarantee that it will be available to the end user. To ensure that a distributed system will complete the required task, it must be reliable. This goal is difficult to achieve because of the complexity of the interactions between simultaneously running hardwares.

To ensure the reliability of a distributed or concurrent system, the following characteristics must be present in the system [Andrews 2000, Anderson 2001]:

- Consistency: The system must be able to provide a consistent view of the system to the user independent of his/her location.
- Deadlock free: The system must try to complete the requested task and must not halt the system processing in case of deadlock. It should be able to recover from such a state.
- Starvation freedom: It is also an important characteristics of a distributed system. Deadlock freedom ensures that (at least part of) the system is alive. This a weak progress property. Starvation freedom ensures that every request/process individually progresses, which implies deadlock freedom but is strong progress property.
- Fault-Tolerance: A fault in a system can cause an error and it can lead it to an incorrect state. For example, race conditions can cause faults that result in a system crash, unexpected shutdown of the program or notifications like illegal operation. The system must be capable of handling such situations.
- Availability: Whenever there is a failure of hardware, software or network, the system must be able to maintain the availability of other resources to the users. It should not be visible to the user that some resource was not accessible. To achieve this, the system must support some recovery or redundancy method to be available to the user.
- Scalable: The distributed system must be able to adapt to new additions of components to the system. It must correctly operate and be transparent to users whenever a new hardware or software is added. For example, we might increase the number of users or servers, or overall load on the system.
- Predictable Performance: The distributed systems must have the ability to provide required responsiveness in a timely manner.
- Resource sharing: Any user or component should be able to access hardware or software in the system. The resource manager must provide naming scheme and control concurrency. For example, in a client/server environment, the resources are provided by the servers and the clients interact with them to use those resources.

It is a challenging task to achieve these high standards in a distributed or concurrent system. Therefore, it becomes necessary to verify such systems using formal techniques like formal verification, model checking, etc.

2.2 PlusCal - Algorithmic Language

The aim of PlusCal [Lamport 2007] language is to describe the algorithm in the form of pseudo-code that is translated to TLA$^+$ specifications for formal verification using TLC model checker. As it retains a typical pseudo-code like syntax, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow. The PlusCal compiler generates a TLA$^+$ specification, which is then verified using TLC. PlusCal is a high-level and powerful modeling language for algorithms, featuring mathematical abstractions, non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms.

In this section, we will show how Peterson's algorithm [Peterson 1981] can be written in PlusCal language. Peterson's two process concurrent algorithm was designed by Gary L. Peterson in 1981, for mutual exclusion that allows two processes to share a single resource without conflict, using only shared memory for communication.

```
--algorithm Peterson
variables turn = 1, try = [id ∈ Peers ↦ FALSE], pCount = 0
process Node ∈ Peers
  begin
ncs:
      while TRUE do
        skip;
        try[self] := TRUE;
        turn := 2 + 1 - self;
try1:
        when ¬(try[2 + 1 - self] = TRUE ∧ turn ≠ self);
cs:
        pCount := pCount + 1;
        try[self] := FALSE;
exit:
        pCount := pCount -1;
        try[self] := FALSE;
      end while;
  end process
end algorithm
```

Peterson's algorithm is a multiprocess algorithm with two processes, numbered

using a constant set Peers. The constant Peers is declared in the enclosing TLA$^+$ module and is defined in the configuration file that we will discuss later in this section. The algorithm starts with the name of the algorithm followed by the variable and process declarations. In our example, we have three global variables turn, pCount and try where the variable try is an array that is defined as a function in TLA$^+$. Then the process Node is declared using the identifiers from the set Peers. The PlusCal language provides useful features to express the functionality of the algorithm as follows:

- It provides constructs like while loop to express repetitive functionality of the algorithm as shown in the Peterson's algorithm.
- The constructs that can halt the activity of the processes to synchronize by waiting for a condition to become true. For example, when construct that is used in the Peterson's algorithm. It blocks the activity of the corresponding process if the condition $\neg(try[2+1-self] = \text{TRUE} \wedge turn \neq self)$ does not hold.
- It allows the grain of atomicity to be expressed by labels. A single atomic step consists of an execution starting at a label and ending before the next label. For example, the statements $count := count + 1$; and $try[self] := \text{FALSE}$; are executed atomically under the label cs.
- It also provides means for expressing nondeterminism using constructs like either and with.

Once the algorithm is written is it enclosed as a comment in a TLA$^+$ module as shown below:

```
---------------------------- MODULE Peterson----------------------------
EXTENDS Naturals, TLC, Sequences, FiniteSets
CONSTANTS Peers

(* --algorithm Peterson
...
end algorithm *)

\* BEGIN TRANSLATION
   Translator adds TLA+ specification here
\* END TRANSLATION
```

The PlusCal translator inserts the algorithm's TLA$^+$ translation, which is a TLA+ specification, between the BEGIN and END translation comment lines, replacing any previous version. The translator also produces a configuration file

that is required by the TLC model checker. We must add to that file the commands that specify the values of the constants declared in the algorithm. The configuration file for the Peterson's algorithm will be as follows:

```
SPECIFICATION Spec
\* Add statements after this line.

CONSTANTS
    Peers = {1,2}
```

PlusCal can easily describe various forms of concurrent algorithms without adding complexity in their expression. In this section, we only discussed a multi process algorithm and showed how it is modeled in the PlusCal language. However, PlusCal language also provides useful constructs to express nondeterminism in the algorithm. Once we have an algorithm written in PlusCal, we can produce its TLA$^+$ specifications and model check them using TLC model checker.

2.3 TLA$^+$ - Specification Language

TLA$^+$ [Lamport 2002] is a formal specification language based on the combination of TLA(Temporal logic of Actions) and ZF(Zermelo-Fraenkel) set theory. It is used to specify and reason about concurrent and distributed systems. It is a rich language that has well-defined semantics for formal reasoning and is designed for writing clear and expressive specifications. It was designed for the verification of large and complicated systems such as communication networks and cache coherence protocols. Its goal is mainly to target the formal reasoning of concurrent and reactive systems. The use of set theory allows TLA$^+$ to be more expressive and easier for specifying algorithms at a high level of abstraction.

The TLA$^+$ language provides a module structure for writing specifications. A system is represented in the form of actions that specify its functionality. Each action states the operations to be carried out and updates the context if required. TLA$^+$ uses prime operator to represent the updated values in the context. Consider an action a that updates a variable i then, the updated variable will be referred as i'. Below, we have TLA$^+$ specifications for the Peterson's algorithm discussed in the previous section.

```
---------------------------- MODULE Peterson----------------------------
EXTENDS Naturals, TLC, Sequences, FiniteSets
CONSTANTS Peers

VARIABLES t, try, pCount, depth, pc, Proc_data

vars ≜ ⟨ t, try, pCount, depth, pc, Proc_data ⟩
```

```
ProcSet ≜ (Peers)

Init ≜ (* Global variables *)
        ∧ t = 1
        ∧ try = [id ∈ Peers ↦ FALSE]
        ∧ pCount = 0
        ∧ depth = 0
        (* Process Proc *)
        ∧ Proc_data = [self ∈ Peers ↦ [ count ↦ 0]]
        ∧ pc = [self ∈ ProcSet ↦ CASE self ∈ Peers → "ncs"]

ncs(self) ≜ LET _try ≜ [try EXCEPT ![self] = TRUE] IN
             LET _t ≜ 2 + 1 - self IN
             ∧ pc[self] = "ncs"
             ∧ TRUE
             ∧ pc' = [pc EXCEPT ![self] = "try1" ]
             ∧ t' = _t
             ∧ try' = _try
             ∧ UNCHANGED ⟨ pCount, depth, Proc_data ⟩

try1(self) ≜ ∧ pc[self] = "try1"
             ∧ ¬(try[2 + 1 - self] = TRUE ∧ t ≠ self)
             ∧ pc' = [pc EXCEPT ![self] = "cs" ]
             ∧ UNCHANGED ⟨ t, try, pCount, depth, Proc_data ⟩

cs(self) ≜ LET _count ≜ Proc_data[self].count + 1 IN
           LET _try ≜ [try EXCEPT ![self] = FALSE] IN
           ∧ pc[self] = "cs"
           ∧ pc' = [pc EXCEPT ![self] = "exit" ]
           ∧ try' = _try
           ∧ Proc_data' = [Proc_data EXCEPT ![self].count = _count]
           ∧ UNCHANGED ⟨ t, pCount, depth ⟩

exit(self) ≜ LET _count ≜ Proc_data[self].count -1 IN
             LET _try ≜ [try EXCEPT ![self] = FALSE] IN
             ∧ pc[self] = "exit"
             ∧ pc' = [pc EXCEPT ![self] = "ncs" ]
             ∧ try' = _try
             ∧ Proc_data' = [Proc_data EXCEPT ![self].count = _count]
             ∧ UNCHANGED ⟨ t, pCount, depth ⟩

Proc(self) ≜ ncs(self) ∨ try1(self) ∨ cs(self) ∨ exit(self)
```

Next $\triangleq$ ($\exists$ self $\in$ Peers: Proc(self))
    $\vee$ (* Disjunct to prevent deadlock on termination *)
      ($\forall$ self $\in$ ProcSet: pc[self] = "Done" $\wedge$ UNCHANGED vars)

Spec $\triangleq$ Init $\wedge$ $\Box$[Next]$_{vars}$

A TLA$^+$ formula describes behaviors, namely those for which it evaluates to TRUE. The main part of a TLA$^+$ specification consists of an initial predicate, *Init* and a next-state action, *Next*. The *Init* predicate specifies the possible initial states, and the next-state action, *Next*, specifies the possible state transitions. An action is a formula containing primed and unprimed variables, where unprimed variables refer to the old state and primed variables refer to the new state. In the Peterson's algorithm, we have four actions identified as ncs, try1, cs and exit. Finally, the TLA formula *Spec* defines the complete TLA$^+$ specifications.

2.4 Model checking

Model checking [Clarke 1981, Queille 1981] is an automated technique for the verification of finite state reactive systems [Clarke 1996a, Clarke 1986]. In this technique, most of the verification process is carried out independently by the system without any involvement by the user. The model checking process can formally be defined using transition system/Kripke structure $M = (S, I, R, L)$ that represents a finite concurrent system and a temporal logic formula ϕ that states the desired property of the system. This process works by verifying if there exists some execution σ of the transition system M such that

$$\sigma \not\models \phi$$

The basic idea of model checking is to traverse each path in a transition system to verify the correctness of the entire system. The tools used for model checking take the description/model of an algorithm in the form of a transition system that describes the possible behaviors of the system. It also requires the set of properties that should hold for the soundness of the system. Then, the tools model check the specifications for all the possible interleavings and try to prove the correctness by verifying the set of properties. If they find an execution that does not verify certain property, they report an error and produce the counter example for the user to find out the cause of the invalidation of property. Then, at this point, the user is required to modify the description of the system.

The set of properties represented as temporal logic formulas are interpreted over sequences of states and can be classified as follows:

Safety properties. Safety properties state that something bad never happens. For example, a mutual exclusion property states that no more than one process

enters the critical section to avoid a bad situation. The logical formulas to guarantee absence of race conditions is an example of such properties.

Liveness properties. If there is no progress in a system, then a safety property is fulfilled if it is initially satisfied. Liveness properties assert that something good will happen eventually. Thus liveness properties are important as they require some progress in the system.

2.4.1 TLC model checker

The TLC model checker [Yu 1999] is used to check TLA$^+$ specifications. As TLA$^+$ can be used to write specifications for very large and complex systems, TLC can only handle subclass of these specifications that includes most specifications of actual system design. This subclass of specifications constitutes the high-level specifications that characterize the correctness of the design.

The TLC model checker performs a breadth-first search to traverse the state graph. Instead of storing complete information of a state in the state graph, TLC uses fingerprints that are 64-bit, probabilistically unique checksums [Yu 1999, Rabin 1981] to represent them. This compact form of representation reduces the amount of space required during the model checking process, hence reducing the space complexity. For exploration of the state graph, it starts by generating all the initial states and verifies the invariant properties for all of them. Then, it adds them to FIFO queue, and launches threads which repeatedly execute the process described below:

- pick a state from FIFO queue and generate all its successor states,
- for each successor state, check if it satisfies all the invariant properties and add it to the end of the FIFO queue,
- if some successor does not satisfy some invariant property, report an error and print the corresponding counter example.

2.4.2 Other modeling languages and model-checkers

There are several other modeling languages that are used to specify algorithms and then to verify them with the help of a model checker. Promela [Holzmann 2003] is a modeling language that is used to describe a system, introduced by Gerard J. Holzmann. A model written in Promela is composed of asynchronous processes, buffered and unbuffered message channels, synchronizing statements, and structured data. The restrictions like lack of clock concept and floating point numbers, allow the representation of an algorithm at the abstract level that is often key to successful verification [Holzmann 2003].

The Promela language is supported by the SPIN model checker [Holzmann 1997], designed by Gerard J. Holzmann, that can exhaustively check all the possible executions to verify the correctness of the algorithm. To overcome the state-space

explosion problem, the SPIN model checker also implements a partial-order reduction technique that successfully reduces the state space.

The Uppaal modeling language [Behrmann 2004] uses timed automata to represent the specifications of the system. It is particularly appropriate for verifying systems that exhibit real-time aspects (in their behavior and their properties), and we do not consider real time in this thesis.

2.5 State space explosion problem

The infamous state explosion problem is well known to be the most serious limitation for the application of model checking techniques. It refers to the fact that the state space generated by a transition system usually grows exponentially in the number of processes and variables. The concurrent and distributed systems are composed of multiple processes cooperating with each other to solve a certain task. These systems are represented using parallelism and their processes are interleaved to represent all the possible runs of the system.

The concept of interleaving of all the transitions from different processes and consequently, parallel composition of all the processes results in an exponential growth of the state space with a small increase in the number of processes. For n processes in a system, one would have to represent $n!$ different orderings of the processes. Many different techniques have been proposed to mitigate state explosion like symbolic state-space representation, state-space hashing, equivalence relations and partial-order reduction. In the next section, we will mainly focus on partial-order reduction that we have adapted in our work.

2.6 Partial-order Reduction

It is quite common in practice that we have asynchronous systems that are composed of a set of processes that cooperatively solve a certain task, e.g., communication protocols, distributed systems, etc. The system executions are modeled as interleavings of process executions. Interleaving is based on a concept that an execution is a totally ordered sequence of actions and to model all possible runs of the system, all possible interleavings of actions need to be represented. Thus, resulting in an exponential increase in the number of executions that must be explored during a traditional model checking process [Clarke 1996a]. The methods that are used for reducing the number of executions to be explored are called reduction methods that include symmetry reduction techniques [Emerson 1996, Clarke 1996b, Ip 1993] and partial-order reduction methods [Godefroid 1991, Peled 1993, McMillan 1992, Valmari 1996].

The aim of partial-order reduction is to reduce the size of the state space to be explored by the model checkers. It exploits the commutativity of actions, which result in the same state when executed in different orders [Clarke 1999]. The reduced state space only represents those interleavings that must be preserved for the verification of a given property. Thus, the execution sequences in the reduced state

space are a subset of the execution sequences of the full state space [Clarke 1999].

2.6.1 Basic partial-order reduction algorithm with DFS

In general, partial-order reduction method is implemented using a depth-first search method that is used to explore all the states in a given system. Below, we show a basic depth-first search method.

```
1  stack contains the states in the current execution sequence
2  statespace contains all the visited states
3
4  initialize() {
5  Add initial states to stack and statespace
6  explore()
7  }
8
9  explore() {
10   s = Pop new state from stack
11    for each transition t enabled at s {
12      for each s' in successor_states(s, t) {
13        if new_state(s') {
14          add s' to the stack and statespace
15          explore()
16        }
17      }
18 } }
```

Depth-first search method can be modified to implement partial-order reduction technique by selecting a subset of all the enabled transitions at line 11 rather than exploring all of them as in the full state space exploration process. The modification of the above code is shown below that implements basic partial-order reduction for the above depth-first search method.

```
1  explore() {
2   s = Pop new state from stack
3   subset = select subset of transitions enabled at s
4    for each transition t in  subset {
5      inPath = false
6      for each s' in successor_states(s, t) {
7        if new_state(s') {
8          add s' to the stack and statespace
9          explore()
10       }
11       else if s' is in stack
12         inPath = true
```

```
13      }
14      if ~inPath
15        break
16    }
17  }
```

At line 3, we select a subset of transitions in the variable *subset* for reduced search that represents the complete set of transitions in full search. These transitions in the subset must have certain properties to ensure the correctness of the verification algorithm. These properties include independence, invisibility and commutativity of the transitions which will be discussed in detail later in this section. In the literature, the subset of transitions is computed using different concepts that include ample set, persistent sets, stubborn sets and sleep sets. The main idea behind their selection is to find out a subset that is sufficient to prove the correctness of the algorithm. As mentioned in [Clarke 1999], the calculation of this subset must satisfy the following three goals:

- When the subset is used instead of complete set of enabled transitions, sufficiently many behaviors must be present in the reduced state graph so that the model checking algorithm gives correct results.
- Using the subset instead of complete set of enabled transitions should result in a significantly smaller state graph.
- The overhead of calculating the subset must be reasonably small.

During the selection of a subset of transitions, the idea of *reduction proviso* is used in the partial-order reduction methods to make the reduction process successful. It was first proposed in [Valmari 1990] by Antti Valmari. The version of proviso used in the above code at line 11 was suggested in [Holzmann 1994], that checks if the successor state was already visited along the current path or not. The *statespace* represents the entire state space of the system while the *stack* represents the current execution path starting from the initial state. While performing traversal of the entire state graph, we remove the states from the *stack* that are not in the execution path any more, but they exist in the *statespace*, thus they are represented as *already visited* states. During search, it is possible that one finds a state already visited in the *stack*. This information is necessary to find out if there was a cycle in the execution path. As mentioned in [Peled 1993],

> A cycle is detected exactly when an edge is created pointing a state that was already visited.

Whenever we encounter a cycle, it leads to a possibility that we missed a transition that was enabled at all the states along the cycle, but that was never taken. One way is to try to find enabled transitions at the first state from where the cycle starts, but this may become very expensive in practice. Thus, we try to explore all the enabled transitions at the current state.

2.6.2 Independence

In distributed algorithms, which are the focus of PlusCal-2 and TLA$^+$, the main potential for reducing state spaces comes from the fact that many actions executed by different processes commute, and the same global configuration is obtained when performing these actions in either order. Whereas the standard interleaving model of concurrency distinguishes two executions that differ in the order in which two independent transitions are performed, a semantics based on partially ordered executions would identify them. Partial-order reduction techniques aim at identifying independent transitions and avoiding the construction of equivalent runs. We assume that every action a is characterized by the set $Cond(a)$ of states where a is enabled and, for any state $s \in Cond(a)$, the set $Act(a, s)$ of states that can be reached by executing a in s. Moreover, actions are associated with processes: we write $Proc(a)$ to denote the process executing action a. The following definitions have been adapted from Holzmann and Peled [Holzmann 1994].

Definition 3. *Two actions a and b are* independent at state s *if the following conditions hold:*

- $s \in Cond(a) \cap Cond(b)$, *i.e., actions a and b are enabled at s,*
- $Act(a, s) \subseteq Cond(b)$, *i.e., the execution of a cannot disable b,*
- $Act(b, s) \subseteq Cond(a)$, *i.e., the execution of b cannot disable a, and*
- $\bigcup_{s' \in Act(a,s)} Act(b, s') = \bigcup_{s' \in Act(b,s)} Act(a, s')$, *i.e., the same sets of states can be reached by executing a and b in either order.*

Two actions are globally independent *if they are independent at every state where they are both enabled. An action a is* safe *if it is globally independent of all actions b with $Proc(b) \neq Proc(a)$.* □

This definition states that two independent transitions can never disable or enable each other and that their execution commutes from any state where they are both enabled. It gives us a dependency relation for two transitions but in practice it is not possible to check these properties for the transitions in a concurrent system. Thus, as mentioned in [Godefroid 1996], this definition is only for semantic purpose. To define independence syntactically, the conditions for two transitions t_1 and t_2 in Γ to be independent are (adapted from [Godefroid 1996]):

- the set of objects that are accessed by t_1 is disjoint from the set of objects that are accessed by t_2.
- the set of objects read by t_1 and t_2 may be similar if none of them writes an object in that set.

These conditions ensure that the data which is accessed by the two transitions does not overlap (detailed discussion on these conditions can be found in [Godefroid 1996]). The author has also discussed the concept of independence between operations for certain kinds of computation that a transition can perform on the data. The operations can affect the enabledness and may also affect the output of the transitions. Below, we have a table from [Godefroid 1996] that shows which operations are dependent and which are independent of each other.

Dependency	Write	Read
Write	*dep*	*dep*
Read	*dep*	*indep*

Two *Write* operations on the same object will always be dependent as they can result in different values depending on the order in which they are executed. Similarly, a *Write* and a *Read* operation performed on the same object are dependent because the result of the *Read* operation will be different. If they are executed on same object then the output depends on the order in which they are performed. Whereas, two *Read* operations are always independent as they only read the values of the objects, independent of the order in which they are executed.

Apart from the *Read* and *Write* operations, Godefroid [Godefroid 1996] also discussed the dependencies that can arise between operations that are performed on bounded FIFO channel of size N. Below we show and explain the two dependency tables for these FIFO channel operations.

The FIFO channel can have *Send*, *Receive* and *Length* operations that can be performed on it to send a message on the FIFO channel, to receive a message from the FIFO channel or to query about number of messages that are currently on the FIFO channel. Some of these operations may be dependent on each other. For example, if two transitions perform a *Send* operation then the output will depend on the order in which they are executed.

Dependency	Send	Receive	Length
Send	*dep*	*dep*	*dep*
Receive	*dep*	*dep*	*dep*
Length	*dep*	*dep*	*indep*

The above table details a constant dependency relation that shows which two operations might be dependent or independent of each other. Consider that there is only one message in the FIFO channel, then the execution of one *Receive* operation will disable the execution of other *Receive* operation. Similarly, if there is only one location left in the FIFO channel, then the execution of one *Send* operation will disable the execution of other *Send* operation on that FIFO channel. Besides,

these operations on a FIFO channel are always dependent as the output of those operations will be different depending on the order in which they are executed.

In constant dependence relation, we assume that the two *Send* operations are always dependent, but it is possible that under certain conditions, they become independent of each other. This provides us an idea of defining conditional dependency relation instead of constant relation. Below we have a table from [Godefroid 1996] that states the conditions under which two operations on a FIFO channel can be considered to be dependent.

Dependency	Send	Receive	Length
Send	$n \geq N$	$n>0$ and $n<N$	$n \geq N$
Receive	$n>0$ and $n<N$	$n \leq 0$	$n \leq 0$
Length	$n \geq N$	$n \leq 0$	*indep*

where n is the number of messages in the FIFO channel and N is the size of the FIFO channel.

The interaction between atomic transitions of a concurrent or distributed algorithm is an important issue for both implementation [Allen 1987] and verification [Katz 1988, Peled 1990, Valmari 1989, Katz 1992]. The concept of defining conditional independence in the form of predicate was first introduce by Shmuel Katz and Doron Peled in [Katz 1992] as follows:

Definition 4. *The independence condition between transitions is a set of predicates* $\Gamma = \{\Delta_{\alpha,\beta} | \alpha, \beta \in T, \alpha \neq \beta\}$ *such that for all* $\alpha, \beta \in T, \alpha \neq \beta, \Delta_{\alpha,\beta} (= \Delta_{\beta,\alpha})$ *satisfies:*

1. *for all the states where the transitions* α *and* β *are enabled and predicate* $\Delta_{\alpha,\beta}$ *holds, the transitions commute, and*

2. *for all the states where the transition* α *is enabled and predicate* $\Delta_{\alpha,\beta}$ *holds, then the execution of transition* α *does not affect the enabledness of* β*.*

Now, the dependency conditions that we showed in the previous section for the operations on FIFO channel will be considered as predicates. At any given state, these predicates can easily be computed and can result in better reduction as compared to constant dependency relations.

2.6.3 Invisibility

The concept of invisibility of transitions was first introduced by Antti Valmari in [Valmari 1990]. A transition is said to be visible if it modifies the truth value of any state predicate from the set of formulas specified for the system. It is described by Antti Valmari as follows:

> Let Φ be a collection of LTL formulas and assume that we have a set $vis(\Phi)$ of visible transitions. A transition t is considered to be visible, iff there are states s and s' such that $s \xrightarrow{t} s'$ and the truth value of at least one state predicate appearing in at least one formula in Φ is different at s and s'. Thus, transition t is *visible* if $t \in vis(\Phi)$; otherwise, t is invisible.

Peled [Peled 1996b, Peled 1998] defines the concept of visibility for a transition system as follows:

Definition 5. *Given a system (Γ, P, M), where*

1. *Γ = (S, T, i) is a finite state system,*
2. *P is a finite set of propositions, and*
3. *$M : S \mapsto 2^P$ is the state labeling function.*

a transition $\tau \in T$ is visible if there are two states s, $t \in S$ such that M(s) $\neq$ M(t) and $t \in \tau(s)$. □

All the concepts of invisibility in the literature use the truth value of the invariants and the temporal properties to identify the invisibility of a transition between any two given states. In this thesis, we use a similar concept of invisibility by dividing a property into a set of predicates that are easier to evaluate for given pair of states. A transition is invisible if the truth values of the set of predicates remains unchanged in the pair of states that is the current state and the successor state. Now, we can redefine the definition by Antti Valmari as follows:

> Let Ω be a collection of predicates built from the given set of properties and assume that we have a set $vis(\Omega)$ of visible transitions. A transition t is considered to be properly visible, iff there are states s and s' such that $s \xrightarrow{t} s'$ and the truth value of at least one predicate appearing in Ω is different at s and s'. Thus, transition t is *visible* if $t \in vis(\Omega)$; otherwise, t is invisible.

2.6.4 Techniques for computing subset of actions/transitions

As introduced earlier, the basic idea of a reduction method is to select a subset of transitions *subset* and ignore the rest of the transitions to avoid unnecessary exploration of the state space. Thus, the researchers have been trying to develop advanced state space techniques where only few of the orderings are explored, ideally they focus on exploring only one ordering for each set of concurrent transitions. The basic idea behind these techniques is to investigate only few transitions at each state that are added to the reduced state graph. The subset of transitions is chosen such that the occurrences of remaining transitions is ignored, without modifying the verification results.

In this section, we will briefly discuss some of the techniques that are used to compute the subset of transitions.

Persistent set: Persistent set were first introduced by P. Godefroid and D. Pirottin in [Godefroid 1993]. A subset T of the set of transitions enabled in a state s of complete state graph is called persistent in s if whatever one does from s, while remaining outside of T, does not interact with T. It is defined as follows [Godefroid 1994]:

Definition 6. *A set of transitions, $T \subseteq \Gamma$, enabled in a state s is persistent in s if and only if, for all nonempty sequences of transitions*

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all the transitions in T. □

Ample set: The ample sets were introduce by Doron Peled in [Peled 1996a]. They are defined as follows:

Definition 7. *An ample set ample(s) for a state s is a set of actions enabled at s that ensures the following conditions:*

C0. *ample(s) $= \emptyset$ only if enabled(s) $= \emptyset$ where enabled(s) is the set of actions enabled at s.*

C1. *Along every path in the full state graph that starts at s, the following condition holds: an action that is dependent on an action in ample(s) cannot be executed before some action in ample(s) is executed.*

C2. *If ample(s) $\neq$ enabled(s), then every $\alpha \in$ ample(s) is invisible.*

C3. *No cycle in the reduced graph contains a state at which some action α is enabled, but is never included in ample(s) for any state s along the cycle.* □

An ample set must satisfy the above four conditions to guarantee the successful selection of a subset that represents the actual set of transitions at a given state.

Stubborn set

The stubborn set method was proposed by Antti Valmari in [Valmari 1990]. The basic stubborn set method or semistubborn set can be defined as follows [Valmari 1996]:

Definition 8. *A set $T_s \subseteq T$ is semistubborn or weakly stubborn at state s_0, if and only if the following holds:*

- *D1: If $t \in T_s, t_1, ..., t_n \notin T_s, s_0 \xrightarrow{t_1 t_2 \cdots t_n} s_n$, and $s_n \xrightarrow{t} s'_n$, then there is s'_0 such that $s_0 \xrightarrow{t} s'_0$ and $s'_0 \xrightarrow{t_1 t_2 \cdots t_n} s'_n$.*
- *D2: There is at least one transition $t_k \in T_s$ such that if $t_1, ..., t_n \notin T_s$ and $s_0 \xrightarrow{t_1 t_2 \cdots t_n} s_n$, then $s_n \xrightarrow{t_k}$. The transition t_k is called a key transition of T_s at s.* □

A stubborn set must satisfy the above two conditions. The first condition, D1, guarantees the commutativity relation of transitions in the stubborn set with transitions outside the stubborn set. The second condition, D2, ensures that there is at least one transition that cannot be disabled by the transitions that do not belong to the stubborn set. That transition is called *key transition*. If every transition $t \in T_s \cap enabled(s_0)$ is a key transition then the set is called strongly stubborn set.

The basic behind these selective search techniques is to consider only those successor states that are reachable through dependent transitions while ignoring the independent transitions. They differ in the way they select the representative transitions and exploit the information from the structure of the system being verified. In our work, we will be focusing mainly on ample sets that we prove to produce in TLC with partial-order reduction technique and persistent sets.

2.6.5 Variants of partial-order reduction

Static partial-order reduction [Holzmann 1994, Kurshan 1998] and dynamic partial-order reduction [Peled 1996a, Yang 2008, Flanagan 2005, Yang 2007] are variants of partial-order reduction technique. In static partial-order reduction, all the computations that include calculation of independence relations are performed before executing the model checking process. In certain cases where systems are complicated, these computations are complex and add additional overhead to the model checking process. Thus, in those cases, static partial-order reduction methods can be used to reduce the size of the state space.

In [Flanagan 2005], C. Flanagan and P. Godefroid have presented a dynamic partial-order reduction technique for model checking softwares. They dynamically compute the redundant parts of the state graph to avoid the unnecessary exploration. They also adapt to the dynamic change in the structure of the algorithm, that involves creation of new processes and threads, or new memory allocations, etc. Its implementation is less complicated as it does not require static analysis of the algorithm but all the additional computation and dynamic changes in the algorithm can cause excessive consumption of resources at runtime. In large systems, this technique might be less effective than static partial-order reduction where, most of the computation is kept separate from the actual model checking process.

Chapter 3

Expressing concurrent and distributed algorithms in PlusCal-2

Contents

Introduction

Algorithms for concurrent and distributed systems [Lynch 1996] are notoriously hard to design, due to the number of interleavings of their constituent processes that must communicate and synchronize properly in order to achieve the desired function. It is all too easy to overlook corner cases, and hard to generate or reproduce particular behaviors during testing. Formal verification of such algorithms is therefore essential, and model checking in particular has been applied with great success in this context. However, there is a conceptual gap between the languages algorithm designers use to convey their ideas and the input languages of model checking tools. While the former emphasize high levels of abstraction in order to present the algorithmic ideas, their semantics is not precisely defined. Languages for model checkers come with a precise (at least operational) semantics but tend to make compromises in terms of the available data types in order to enable compact state representations and the efficient computation of operations such as the computation of successor

(or predecessor) states. Most model checkers, in particular symbolic ones, support only low-level data types such as fixed-size integers and records. TLC [Yu 1999], the model checker for the specification language TLA$^+$ [Lamport 2002], accepts a significant fragment of TLA$^+$, which is based on set theory; it thus provides one of the most expressive and high-level input languages for model checking. However, TLA$^+$ models encode transition systems via logical formulas, losing much of the (control) structure that is present in code. As a result, TLA$^+$ representation of an algorithm becomes unnatural for the algorithm designers.

Recently, Lamport introduced the PlusCal algorithm language [Lamport 2006b] (originally called +Cal). While retaining the high level of abstraction of TLA$^+$ expressions, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow. The PlusCal compiler generates a TLA$^+$ model corresponding to the PlusCal algorithm, which is then verified using TLC. PlusCal is a high-level language that features set-based abstractions, non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms and aims at bridging the gap that we described above.

Unfortunately, as we discussed in chapter 1, use of Lamport's PlusCal requires good knowledge of TLA$^+$, and even of the translation of PlusCal to TLA$^+$. Aiming at a simple translation in order to make the resulting TLA$^+$ model human readable, Lamport imposed some limitations on the language that can make it difficult or unnatural to express distributed algorithms. After initial attempts to extend the original language and its compiler, these limitations motivated us to develop a new version of PlusCal that retains the basic ideas of Lamport's language but overcomes the shortcomings that we identified. At the same time, we aim at a translation that enables the use of reduction techniques and hence more efficient verification.

In this chapter, we will first discuss the requirements for a modeling language. Then, we will present the syntax of PlusCal-2 [Akhtar 2010] language along with the organization of the algorithm and an example explaining the details of how to write an algorithm in PlusCal-2. In section 3.3, we will describe the compilation of PlusCal-2 algorithm to TLA$^+$ specifications. Finally, in section 3.4, we will show how a PlusCal-2 algorithm can be model checked using the TLC model checker.

3.1 Requirements for a Modeling Language

The purpose of a modeling language is to describe *what* a system must perform, not *how* a system must perform. In general, programming languages focus on the efficiency of the implemented system. The use of objects in object-oriented languages, to represent the data structure introduces more complexity when it comes to handling those data structures. As a result, they add unnecessary details to the system description. The languages that focus mainly on the accurate description of the system instead of implementation, were introduced for the purpose of system

analysis or the verification of the system.

There are many languages in the literature that are proposed and used for modeling systems. Some of these languages are similar to pseudo-code like AsmL [Gurevich 2005] while others are input languages for model checkers like Promela [Holzmann 2003] and SMV [McMillan 1993]. DistAlgo [Liu 2011] is another example of pseudo-code like languages but it also generates executable implementations for the modeled distributed systems. MACE [Killian 2007] is a domain specific language, to design robust and high performance distributed systems. An extension to MACE is a tool called CrystalBall [Yabandeh 2009] that is built on the top of the MACE framework. It predicts potential future safety property violations in a deployed running distributed system instead of verifying the system from initial state.

In practice, the users of these modeling languages are algorithm designers who are responsible for describing the functionality of the system in terms of algorithms before actual implementation. Thus, these languages should be simple so that the users can learn and use the language constructs easily. In general, a modeling language must have following properties.

- **High-level abstractions of system:** High-level of abstractions are used to hide the details of a system so that the algorithm designer has to focus on less concepts at a time. Abstractions help in reducing the errors introduced at the design stage and allows better understanding of the main goals of the system instead of its implementation. Thus, a modeling language must have the capability of expressing a system at high-level of abstractions.

- **Ability to express concurrency:** A modeling language should be able to express concurrency for describing concurrent and distributed systems. In other words, it should be able to describe a system in which several computations are performed simultaneously, and potentially interacting with each other.

- **Non-determinism:** Non-determinism is an important concept in state transitions systems. To focus on the high-level abstractions, details are left open for non-deterministic choices. It provides ability to express a transition in a system that can lead to multiple states. For example, an iteration over a set by a construct can have multiple set orderings to iterate over the set. Thus, it can lead to multiple different states that cannot be determined at the time of initialization.

- **Express fairness assumptions:** A modeling language should be able to assert fairness assumptions that are required by the liveness properties. There are two common types of fairness conditions; strong and weak fairness. Weak fairness of a transition ensures that the transition must occur if it remains continuously enabled and strong fairness ensures that a transition must occur if it is repeatedly enabled from time to time.

	SMV	Alloy	Promela	AsmL	SETL	PlusCal
High-level Abstraction	++	++	++	++	++	++
Concurrency	++	+	++	-	+	++
Non-determinism	+	+	++	++	+	++
Fairness assumptions	+	-	+	-	-	-
Simplicity	-	-	-	++	++	++

Table 3.1: Comparison of various modeling languages.

- **Simplicity:** Algorithm designers are the users of modeling languages who, in general, use pseudo-code like languages to express the model of the system. They should be able to easily learn and use these languages. The language should be composed of simple and understandable constructs that do not add complexity to the algorithm. It also helps in reducing the amount of errors that are produced because of complex data structures and constructs.

In the literature, we found various modeling languages that focus on some of the above properties while compromising on others as shown in the figure 3.1. Most of these languages have the ability to conveniently express concurrent systems at higher-level of abstractions. However, some of these language do not have the capability to express fairness assumptions that is an important feature of concurrent systems.

PlusCal language by Leslie Lamport provides simple pseudo-code like interface for the user to express concurrent systems. It provides constructs to express non-determinism in the algorithm while retaining the aspect of simplicity. It allows set-theoretic expressions to model the system at high-level of abstraction but it has other limitations as mentioned in chapter 1 section 1.2. Thus, in this chapter, we propose a new version of this modeling language PlusCal-2 that removes all the previous limitations and has the ability to express concurrency, non-determinism, fairness and is simple in practice.

3.2 The PlusCal-2 Language

PlusCal-2 is a language for describing concurrent and distributed algorithms. In this section, we briefly explain the basic structure and semantics of an algorithm in PlusCal-2 with the help of a Leader election algorithm shown in appendix 1.2. It is an algorithm for electing a leader in a unidirectional ring proposed by Dolev, Klawe, and Rodeh [Dolev 1982].

3.2.1 Structure/Organization of an Algorithm

The structure of a PlusCal-2 algorithm can accommodate all the information required to write complete specifications of an algorithm. It is composed of various

sections as in figure 3.1 that shows a general outline of a PlusCal-2 algorithm. Here, we explain all the sections that make up a complete PlusCal-2 algorithm.

Header section. This section starts the algorithm and contains the information about the name of algorithm, the TLA$^+$ modules to be imported and the constants that appear in the description of the algorithm. The constants are the parameters of the algorithm whose value remains unchanged throughout the algorithm. Thus, they are declared separately in the header section. The definition of these constant symbols is done in the instance section which will be explained later.

Below, we have the header section from the Leader election algorithm:

algorithm Leader
extends Naturals, Sequences
constants N, I

The reserved word **algorithm** is followed by the name of the algorithm and then the reserved word **extends** lists any modules to be imported. These modules contain definitions of operators that are used within the algorithm. In our example, it imports the module Naturals and Sequences from the TLA$^+$ standard library. Then the global constant parameters (N and I in our example) are also declared in the header section; these will later be instantiated to obtain a concrete instance for verification.

Declaration section. This section provides the space for declaring variables, definitions and procedures. It is available for declaring global, process and procedure level entities. At global and process level, one can declare variables, definitions and procedures whereas at procedure level a user can only declare variables as the functionality required by a procedure is less complicated. In contrast to the original PlusCal, PlusCal-2 implements scoping rules for the variables, definitions and procedures. The variables declared within one process cannot be accessed in any other process. This helps in reducing some of the errors that get introduced by accidentally using a variable of some other process.

In our Leader election algorithm, we have variable declaration as shown below:

variable
net = [p ∈ 0..(N−1) ↦ ⟨⟩]

A user can declare multiple variables along with their initializations. In the above example, we have variable net that represents the network in the algorithm on which all the nodes send messages for each other. The declaration of net states that initially, it is an array indexed by the set 0..(N−1) such that each entry in net is equal to the empty sequence ⟨⟩. This variable is declared as global variable and will be accessible in the entire algorithm.

The next part of the declaration section in our example is declaration of definitions, as shown below:

```
1                          (* Header section *)
2
3  algorithm ⟨⟨algorithm name⟩⟩
4  extends ⟨⟨module names⟩⟩
5  constants ⟨⟨name of the constants⟩⟩
6
7                          (* Declaration section *)
8
9  variable ⟨⟨variable declarations⟩⟩
10 definition ⟨⟨definition name⟩⟩ ≜ ⟨⟨definition description⟩⟩
11 procedure ⟨⟨procedure name⟩⟩(⟨⟨parameters⟩⟩)
12   variable ⟨⟨local variable declarations⟩⟩
13   begin
14     ⟨⟨procedure body⟩⟩
15   end procedure
16
17                         (* Process section *)
18
19 ⟨⟨strong⟩ fair⟩ process ⟨⟨process name⟩⟩[⟨⟨number of instances⟩⟩]
20   ⟨⟨Declaration section as above⟩⟩
21   ⟨⟨Process section⟩⟩
22   begin
23     ⟨⟨main code for process⟩⟩
24   end process
25   ⟨⟨Invariant and property definition section as below⟩⟩
26
27                         (* Main block section *)
28 begin
29   ⟨⟨main code for algorithm⟩⟩
30 end algorithm
31
32                         (* Invariant and property definition section *)
33
34 invariant ⟨⟨invariant definition⟩⟩
35 temporal ⟨⟨temporal property definition⟩⟩
36
37                         (* Instance and constraint section *)
38
39 instances ⟨⟨definition of constants declared in Header section⟩⟩
40 constraints ⟨⟨constraint definition⟩⟩
```

Figure 3.1: General outline of PlusCal-2 algorithm.

definition send(ch, msg) $\triangleq$
    [net EXCEPT ![ch] = Append(@, msg)]

where send is an operator that carries two parameters, the channel number ch and the message msg. The purpose of this operator is to send the message msg on a given channel over the network net. This is written in PlusCal-2 as an EXCEPT expression that builds a function that is similar to net but overrides the function value for argument ch.

Append is an imported function from the standard module Sequences. It appends the message msg at the end of sequence represented by the symbol @. The symbol @ inside an EXCEPT represents the value of the original function at the argument that is being overridden. This is a TLA symbol and in this example it stands for net[ch].

Process section. It allows the designer to describe the algorithm in the form of processes. The user can declare multiple processes along with their definitions as it was available in original PlusCal. In PlusCal-2, we extended this section by introducing sub-processes. A user can now define sub-process in a similar way as the processes are defined. These sub-proceses can have their own variables, definitions and procedures to carry out their own local tasks. These sub-processes can read and modify the variables of their parent processes but they cannot access variables of their peer processes. The implementation of scoping rules helps us eliminates the chances of interference between processes.

In our Leader election algorithm, we have one process declaration in *process section* as follows:

process Node[N]
variables
    Active = TRUE, know_winner = FALSE,...
begin
...
end process

It starts with the keyword **process**, followed by the name of the process Node and then, in square brackets, we define the number of instances of that process in the system. In the above example, we use a constant N that tells the number of processes required. The name of the process is followed by the *declaration section* of the process that is used for the declaration of local variables which cannot be accessed outside this process. In our example, we only have variable declarations, but a process can also have procedure and definition declarations if they are required by the user. The *declaration section* is followed by the *code section*, between the keywords **begin** and **end process**, that prescribes the functionality of the process.

Main block/code section. This section describes the task of a process, procedure or an algorithm that it is required to perform during its execution. This description can be written using assignment statements, loops, etc. that we'll explain in the next section. In our example, the process declaration contains a *code section* whereas the code section for the algorithm is absent as it was not required in the algorithm.

Invariant and property definition section. Invariants are the predicates that should remain true during the execution of an algorithm. These invariants and the temporal properties are used to express the correctness of an algorithm. This section provides the space to write the invariants and the temporal properties.

In Leader election algorithm, we have a temporal property and an invariant for the entire algorithm that should remain true during the execution of the algorithm. The temporal property is as follows:

temporal ∃ p ∈ Node : ◇ Node[p].winner

where **temporal** is a reserved word that indicates the start of a temporal property. This property states that the nodes in the leader election algorithm will choose a leader at some point during execution. When a node is a leader it sets its local variable winner to true. In the property, we use the symbol ◇, that stands for 'one day', along with the variable winner and it can be read as one day the variable winner becomes true.

The processes or algorithm can also be followed by an invariant as in our example of Leader election algorithm. The algorithm has an invariant that should remain true during the entire execution of the algorithm. The invariant is as follows:

$$\textbf{invariant } \forall p \in Node : Node[p].winner \Rightarrow (\forall q \in Node \setminus \{p\} : \neg Node[q].winner)$$

invariant is a reserved word that is followed by the expression that represents the description of an invariant.

Instance and constraint section. Constraints are used to restrict the use of an entity during the execution of the algorithm e.g., one can add a constraint to bound the length of a communication channel. They can be written in this section along with the instantiation of the constants that are used in the algorithm. The constants are declared in the header section, at the start of the algorithm.

In our Leader election algorithm, we have constant initializations as shown below:

instances N = 3, I = 1

where **instances** is a reserved word that marks the start of constant initializations. Then the constants declared in the *header section* are initialized.

3.2.2 Syntax and semantics of PlusCal-2

The statements in PlusCal-2 are simple and for the most part similar to the original PlusCal. Statements are organized in the form of small blocks that represent the atomic steps of an algorithm. These atomic steps are labeled to indicate the start of an atomic block of statements.

Labeling conditions. Labels are essential at certain places for the purposes of compilation. In PlusCal-2 language, if the user doesn't insert labels, the compiler will add them and it informs the user about the locations at which the labels are inserted. However, the user should be aware that these additional labels influence the grain of atomicity, beyond the explicit labels.

The compiler follows some rules to place the labels in the PlusCal-2 algorithm which are as follows:

- The first statement of any process, procedure or the main algorithm should be labeled. As this statement marks the starting point of that entity.
- A **loop** statement must be labeled as it repeatedly executes its block of statements. The statement following the **loop** statement must also be labeled as it should be executed after the execution of a **break** statement inside the **loop**.
- A **for** statement must be labeled as it executes its block of statements for each value in the set. Similarly, the statement following a **for** statement must also be labeled.
- A **with** statement with an existential quantification over a set must be labeled.
- A **goto** or a procedure call statement is always followed by a labeled statement as both these statements change the flow control of the algorithm.
- The **atomic** statement must also be labeled as it changes the access rules for the execution of the blocks of statements.

The PlusCal-2 compiler makes use of these rules and generates the labels (if necessary) to produce the corresponding TLA$^+$ specifications.

Fairness annotations. Assumptions of fairness conditions are necessary for the verification of liveness properties and they must be defined in terms of the TLA$^+$ actions. PlusCal-2 provides a simple way to add these fairness conditions in the TLA$^+$ specifications. A user can annotate the processes or labels using the keyword **fair** in order to specify weak fairness conditions. Strong fairness conditions can be added using the keyword **strong** before the keyword **fair**.

In our Leader election algorithm, we have fairness annotations for the process Node as shown below:

fair process Node[N]

Statements. Below we explain the PlusCal-2 statements that are used to express an algorithm in PlusCal-2 language.

- **Skip**
  The **skip** statement does not have any effect. It simply passes the control to the next statement.

  skip

- **Assignment**
  The assignment statement can be an assignment to a variable or a location in an array or a record as shown in the sample examples below:

  counter := counter + 1;
  array[i] := TRUE;

  The first statement is an assignment to a variable, while the other one is an assignment to location i of variable *array*.

- **Atomic**
  The **atomic** statement allows the user to have multiple labeled blocks of statements to be executed without any intervention from the other processes. A process that first starts executing an atomic statement, acquires a global lock over all the labeled blocks. Only that process can progress while the other processes do not have any right to execute further.

  Below we have a general form of atomic statement,

  $\underline{\mu}$: **atomic**
  B_1
  $\underline{\nu}$: B_2
  end atomic

  where B_1 and B_2 are the two separate blocks of statements that are to be executed. The process which starts the execution should finish the execution of both blocks of statements atomically.

- **Branch**
  The **branch** statement is composed of multiple blocks of statements guarded by conditions. It allows execution of any one of the blocks whose corresponding condition is TRUE. If none of them is TRUE then the **branch** statement blocks. It is similar to **if-then-else** statement in the original PlusCal language. It can also be used instead of **when** and **either** statement of original PlusCal. The basic structure of a **branch** statement is shown below:

branch
  C_1 **then** B_1
or
  C_2 **then** B_2
or
  ...
or
  C_n **then** B_n
[**else** B]
end branch

The above structure of **branch** statement will try to check the truth value of the guard conditions C_1, C_2,... C_n and will select the ones that are found to be TRUE in order to non-deterministically execute some corresponding block B_i. If none of them are true then it will select the **else** block for execution, where **else** is a shorthand for

or
  $\neg(C_1 \vee C_2 \vee ...C_n)$ **then**

- **If-then-else/Either/When**
  PlusCal-2 supports partial backward compatibility by allowing usage of some statements from the original PlusCal language. They are encoded to **branch** statements during compilation. The syntax for these statements is similar to the syntax available in PlusCal language. Below we have a general structure for **if-then-else** statement:

  if C_1 **then**
    B_1
  else if C_2 **then**
    B_2
    ...
  else B
  end if

  where C_1, C_2,... are the conditions that have to be TRUE to execute their corresponding blocks of statements. If none of the conditions are TRUE then it executes the **else** part of the **if-then-else** statement. The **if-then-else** statement is encoded to **branch** statement during PlusCal-2 compilation as follows:

  branch
    C_1 **then** B_1

or
  $C_2 \wedge \neg C_1$ **then** B_2
or
  $\cdots$
or
  $\neg(C_2 \vee C_1 \vee \cdots)$ **then** B
end branch

Similarly, we have **either** statement and below we have a general structure for this statement:

either B_1
or B_2
or ...
end either

where B_1, B_2,... are the blocks of statements that will be chosen non-deterministically by the TLC model checker for execution. It is also encoded to the **branch** statement during the compilation process as follows:

branch
  TRUE **then** B_1
or
  TRUE **then** B_2
or ...
end branch

The **when** statement is also available in PlusCal-2. Its syntax is also similar to the one available in original PlusCal. Below we have a **when** statement with a condition C_1 that should be TRUE in order to proceed the execution.

when C_1;

The above statement is encoded to the following **branch** statement:

branch
  C_1 **then skip**
end branch

In the above **branch** statement if C_1 is FALSE then it will block until it is executed at a state where C_1 is TRUE. Thus, **branch** statement of PlusCal-2 subsume **when** statement of PlusCal language.

- **With**

with $i \in Set$
  B_1
end with

The above **with** statement executes the block of statements B_1 with a non-deterministically chosen value for identifier i from the set *Set*. It is similar to the **with** statement from the original language PlusCal. Another special syntax for **with** statement is to initialize the identifier i with a value that can also be an expression and then execute the block B_1 with that value.

with $i = Expr$ B_1 **end with**

- **For**
  The **for** statement is an iterative statement that executes its block of statements repeatedly. Unlike **with** statement, for each element i in the set denoted by *Expr*, in some fixed (but unspecified) order, it executes the block of statements inside the **for** statement. Below we have a general form of **for** statement,

  $\underline{\lambda}$: **for** $i \in Expr$
    B_1
  end for

  Although the block of statements B_1 can modify the variables appearing in the expression *Expr*, the initial set that is obtained by evaluation of the expression *Expr*, is used during the execution of the **for** statement.

- **Loop**
  The **loop** statement is a statement that executes its block of statements for an infinite number of times.

  loop
    B_1
  end loop

  where B_1 is a block of statements that the loop will execute infinitely. A **goto** or a **break** statement can be used to change the execution flow of the algorithm.

- **Procedure call**
  The procedure call statement is used to invoke a procedure to carry out a different task. Parameters of the procedure can be instantiated by arbitrary TLA$^+$ expressions, typically involving variables. The name of the procedure along with the parameters inside parenthesis is used in the call as shown below:

  $\langle\langle procedure\ name \rangle\rangle(\langle\langle parameters \rangle\rangle)$

- **Return/Break/Goto**
  The **return** statement is used inside the procedure to move the program control back to the point from where the procedure call was initiated. In contrast to traditional programming languages, a **return** statement in PlusCal-2 language cannot be used to return a value of an expression. However, a value can still be returned using a global variable. This constraint is present to simplify the translation to TLA$^+$ specifications.

  return

  The **break** statement is used to stop the execution of a statement like **for**, **with** or **loop**. It transfers the program control to the label following the statement.

- **Print**
  The **print** statement is similar to a **skip** statement. The only difference is that it asks TLC to print the value of a given variable or expression. It is written as follows:

  print ⟨⟨*Expr*⟩⟩

  The TLA$^+$ expression *Expr* can be built from variables and constants. In particular, it can be a text within quotes.

  print "Text"

3.3 Compilation: translation to TLA+

Once the algorithm is written in PlusCal-2 language, the PlusCal-2 compiler translates it to a TLA$^+$ specification. The compiler has multiple stages for translating a PlusCal-2 algorithm. These stages are explained in Figure 3.2. The parser analyzes the algorithm and produces an Abstract Syntax tree (AST), which is then passed on to the PlusCal-2 normalizer that simplifies the structure of the AST tree. In the next step, the PlusCal-2 translator converts the AST tree into blocks of statements identified by their labels, in a format called intermediate language. This intermediate language helps organize all the information required to produce TLA$^+$ specifications. Finally, the PlusCal-2 TLAgenerator assembles the TLA$^+$ specifications and writes them to the files for TLC model checker. In the sections below, we explain the compilation process of the PlusCal-2 compiler in detail.

3.3.1 PlusCal-2 Parser

The PlusCal-2 parser contains the complete information about the syntax and structure of a valid PlusCal-2 algorithm. The algorithm written in PlusCal-2 language is parsed and validated by the parser. In result, if the algorithm contains

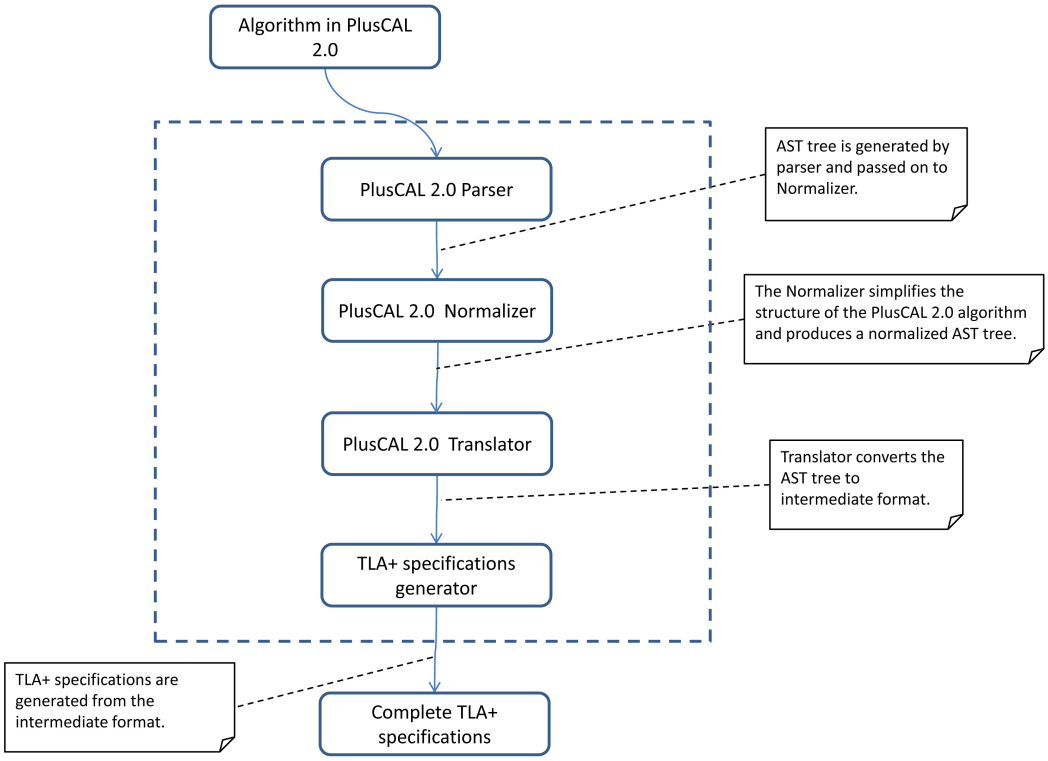

Figure 3.2: The compilation phases for PlusCal-2.

an invalid syntax for statements, then a syntactic error is produced to inform the user. The error message specifies the location for the incorrect statement in the file and states the error details.

PlusCal-2 also implements the scoping rules to define the structure of a PlusCal-2 algorithm. These rules are maintained in the PlusCal-2 parser and it confirms that the algorithm obeys these rules. The scoping rules followed by the PlusCal-2 parser are defined below:

- The variables at the top level are treated as global variables.
- The variables defined inside a process can not be accessed outside their scope.
- Sub-processes have an access to the local variables of their parent processes.

To implement these hierarchical rules, the PlusCal-2 parser also manages a symbol table to record the details of the algorithm. It records the declaration section that contains variables, procedures, and definitions at the global level and for all the processes in a data structure. This data structure is represented as a stack of frames (that represents the declaration sections). As the parser parses the PlusCal-2 algorithm, it updates the symbol table with the information that it finds related to declaration sections. The parser not just adds information to the symbol table, it also consults it when necessary. It refers to the symbol table in the following cases:

- when it finds a variable/procedure/definition name in a statement it refers to the symbol table to confirm its declaration.
- when it finds a variable/procedure/definition in declaration section it refers to the symbol table to confirm that it is not declared yet.

PlusCal-2 allows same names for a procedure with different number of variables, to implement this flexibility in the language, the symbol table also manages all the required information.

3.3.2 PlusCal-2 Normalizer

After the parser parses the algorithm for syntactic errors, it produces an Abstract syntax tree (AST) for that algorithm and passes it to the PlusCal-2 Normalizer. The normalizer traverses the whole AST tree and reorganizes/simplifies the statements. The normalization process is necessary to simplify the complex PlusCal-2 constructs that cannot be directly represented in TLA$^+$ language. Below, we explain the normalization process for the different statements:

Branch/if-then-else statement. For **branch** statement, the normalization step helps reducing irrelevant nesting of **branch** statement with other **branch** or **if-then-else** statements. A TLA$^+$ action cannot be produced directly from the code

written in PlusCal-2 language. Thus, the reorganization of **branch** statement helps PlusCal-2 translator to generate its corresponding TLA$^+$ action. In this section, we will discuss two cases for which normalization is performed to reorganize the **branch** statement. Below we have the first case where a **branch** statement is followed by an unlabeled block B_3 of statements:

branch
  C_1 **then** B_1
or
  C_2 **then** B_2
end branch
B_3

The block of statements B_3 that follows the **branch** statement will always be executed after the execution of one of the **branch** arm/entry. This means it can be added to each of the **branch** arm and will avoid an extra label that would have been generated by the PlusCal-2 translator for the block of statements B_3. Below is the new structure of the above **branch** statement after normalization:

branch
  C_1 **then** B_1 B_3
or
  C_2 **then** B_2 B_3
end branch

The second case where normalization for **branch** statement is performed, is to reduce the nesting of **branch** statements. If the nested **branch** statements are not preceded by statements then the structure of **branch** statement can be simplified. Below we have the case that demonstrates the simplification of nested **branch** statements.

branch
  C_1 **then**
    branch
      C_3 **then** B_1
    else
      B_2
    end branch
or
  C_2 **then** B_3
end branch

The normalized form of the above case is as follows:

branch

$C_1 \wedge C_3$ **then**
B_1
or
$C_1 \wedge \neg C_3$ **then**
B_2
or
C_2 **then** B_3
end branch

The only restriction to remove this nesting is that there should not be any statement before the inner branch statement, that should be executed before the execution of the inner branch statement.

With statement. The normalization of **with** statement involves reorganization of statements that might be present after the **with** statement. The statements directly after the **with** statement are added to a new block of statements by introducing a new label in the algorithm, thus increasing the number of actions in the TLA$^+$ specifications. The normalizer tries to adjust them inside the **with** statement if the expression is an assignment to an auxiliary variable and the block B_1 does not contain a **break** statement.

with i = 100
B_1
end with
B_2

In the above case, **i** is an auxiliary variable that is initialized with the execution of **with** statement and is inaccessible outside **with** statement. In this case, the block of statements B_2 hanging after the statement can be added inside it if the block B_1 does not contain a **break**, **return**.

3.3.3 PlusCal-2 Translator

The main idea behind PlusCal-2 Translator is to simplify the structure and reorganize the data in PlusCal-2 algorithms. This eases the generation of TLA$^+$ specifications and configuration data for TLC model checker. The PlusCal-2 algorithm is simplified to an intermediate format that is composed of blocks of statements identified by their labels from PlusCal-2 algorithm. The syntax for the intermediate format is described below:

Intermediate format syntax The intermediate format uses 4 types of PlusCal-2 statements that are **skip**, **branch**, **assignment** and **with** statement where (without loss of generality) the **branch** is assumed not to contain an **else** part. It also introduces some additional variables and some assignment statements updating those

variables to define the control flow of the blocks of statements. One of the additional variable is an array **pc** that refers to the program control or control flow of the algorithm. It contains a location for each process and main algorithm (if present) identified by its identity number. In the PlusCal-2 algorithm, the user simply writes the number of processes it requires for each type of process. Then, the compiler assigns the identities by producing a range of identity numbers for each type of process and one for the main algorithm if present. This program control array is used to store the labels of the next block of statements to be executed for each process or main algorithm.

Another addition to the algorithm in intermediate format are the guards that are added to each labeled block in this format. For the blocks belonging to the main algorithm and the processes, this guard has the form:

cp = any

where cp holds the identity of the current process that is allowed to execute the blocks of statements, and any is some number other than the identities of the processes. This guard allows access to all the processes. This is introduced in PlusCal-2 to support the **atomic** statement. We will explain the other guards in the discussion about translation of **atomic** statements.

Another addition to the algorithm is the declaration of all the auxiliary variables that are introduced during the translation to intermediate format of **for** and **with** statement. This is necessary to avoid any conflict with the user defined variables. If the conflict exists, then the auxiliary variable is renamed by adding an extra '_' before the auxiliary variable.

Below we explain the translation of various PlusCal-2 constructs to intermediate format.

Branch. The **branch** statement is one of the constructs of intermediate format, thus the only modification in the structure of the **branch** statement is the introduction of the program control statement. Now, if we take a general form of the **branch** statement as shown below:

$\underline{\lambda}$: **branch**
  C_1 **then** B_1
or
  C_2 **then** B_2
or ...
end branch
$\underline{\mu}$: ...

In this piece of code, the **branch** statement is identified by the label λ and the next label that follows it in the algorithm is μ, so the PlusCal-2 translator will add a program control statement inside the **branch** statement. The new structure of the above code after the translation will be as follows:

$\underline{\lambda}$: **branch**
  C_1 **then**
    B_1
    pc := μ
or
  C_2 **then**
    B_2
    pc := μ
or ... pc := μ
end branch
$\underline{\mu}$: ...

The **when**/**either**/**if-then-else** statements were introduced in PlusCal-2 for backward compatibility and they are eliminated during PlusCal-2 normalization phase. Thus, we do not need to handle them any more.

Loop. A loop statement is an infinite execution of a block of statements. The translation of this statement is performed by simply removing the construct and adding a program control statement to move the control back to the same execution block. If we take the following case:

$\underline{\lambda}$: **loop**
  B_1
end loop
$\underline{\nu}$:...

In the above general form of a loop statement, we have two labels λ and ν. The **loop** statement in λ defines an infinite execution of block of statements B_1. The PlusCal-2 translator will translate this code as follows:

$\underline{\lambda}$: B_1
  pc := λ
$\underline{\nu}$:...

The label λ will be executed indefinitely and only the statements within B_1 can cause it to break and move to any other label. For example, an **if-then-else** statement can have a condition to break this loop by changing the program control variable.

With. As the syntax for intermediate language contains a **with** statement, thus a **with** statement in PlusCal-2 algorithm remains unchanged during translation to intermediate language. However, the body of the **with** statement is translated to intermediate language.

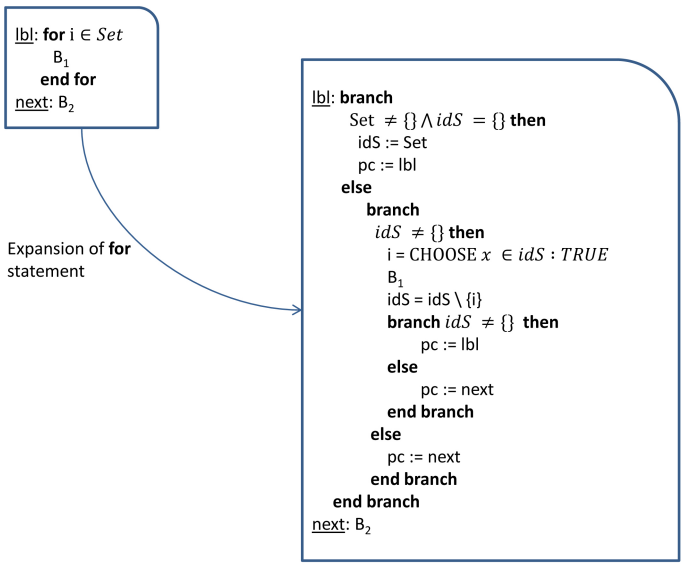

Figure 3.3: The expansion of *for* statement in PlusCal-2 translation phase.

For statement. The **for** statement is an iteration statement that allows execution of a block of statements with each element in the set. This statement is further expanded during translation to intermediate format. The expansion of **for** statement is shown in the Figure 3.3.

The figure shows the exact translation of the **for** statement to the intermediate format. The main idea is to pick an element in the Set and execute the block of statements represented by B_1 using that element. The translation of the **for** statement introduces an auxiliary variable idS to store the values of the original set Set. Once it picks an element from the set, it should be removed after the execution of B_1 using that element. To avoid corruption of data in Set, the auxiliary variable idS is used to keep track of the unused elements. If the block of statements B_1 contains a **break** or **goto** statement then the auxiliary variable idS is reinitialized to an empty set before updating the program counter value.

In the translation, we use branch statement to initialize idS if the set of values provided by the user are not empty. After the initialization of idS, it sets the program control variable to loop back to the same label so that the block of statements in the **for** statement can be executed.

Once the auxiliary variable idS is initialized, each element in the set is chosen using the following statement:

```
i := CHOOSE x ∈ idS : TRUE
```

This statement picks an element and then executes the block of statements B_1. At the end of the statements, it removes the element from the set that is represented by the following statement

idS := idS \ {i}

After the removal of the element, we use branch statement that checks if the set idS is empty then it should finish the repetitive execution of the label by setting the program control variable to the next label. If idS is not empty then it should repeat the execution of the same label for the rest of the elements in the set.

Atomic statement. The **atomic** statement allows the execution of multiple blocks of statements atomically without the intervention of other processes that might want to execute the same or other set of blocks. The only restriction that this construct has is that a user cannot use **goto** statement to jump inside or outside an atomic block. The translation of **atomic** statement to intermediate language introduces an assignment statement for the variable cp in the first block identified by the label of the **atomic** statement. It sets the variable cp to the identity of the process or main algorithm executing it. Another assignment statement for the variable cp is added in the last block inside the **atomic** statement that sets it back to the constant any. If the **atomic** statement only contains single block of statements, then the atomic statement is removed.

Below we have the general form of an **atomic** statement.

$\underline{lbl_1}$:
  atomic
    B_1
$\underline{lbl_2}$: B_2
...
$\underline{lbl_n}$: B_n
  end atomic

Then, the translation of this statement will be as follows:

$\underline{lbl_1}$:
  branch
    cp = any [$\vee$ cp = self] **then**
      cp := self
      B_1
  end branch
$\underline{lbl_2}$:
  branch
    cp = self **then**
      B_2
  end branch
  ...
$\underline{lbl_n}$:
  branch
    cp = self **then**

B_n
cp := any
end branch

In translated code above, we have labeled blocks, lbl_1, lbl_2,.. lbl_n, that represent n blocks of statements and each of them are guarded by a condition. The blocks for the main algorithm and the processes outside the **atomic** statement as well as the first block of the **atomic** statement in the intermediate language are guarded by a condition as follows:

cp = any

where **cp** represents the identity of the current process or main algorithm that is allowed to execute the blocks of statements. However, the blocks for the procedures are guarded by the condition as follows:

cp = any ∨ cp = self

The block labeled by lbl_1, is the first block of **atomic** statement that has the above guard and allows any process to enter it. The first process who is granted the access changes the value of the variable cp to its own identity self using the following statement:

cp := self

After the execution of the block lbl_1, no other process can execute this **atomic** statement or any other block of statements unless this process finishes. All the blocks following the first block of statements inside the **atomic** statement are guarded by the following condition:

cp = self

This guard only allows the process or the main algorithm who first executed the **atomic** statement to enter the corresponding block. At the end of the execution of the last block lbl_n, the process or the main algorithm will set the value of variable cp back to the constant any. That will enable the other processes to proceed with their execution.

Now, if the **atomic** statement contains a procedure call and that procedure also contains an atomic statement then the additional variable depth is used to determine the level of the nested **atomic** statements. The value of the variable depth is incremented by 1 at the beginning of an **atomic** block by adding the following statement

depth := depth + 1

and decremented at the end by 1 by adding the following statement in the last block

depth := depth − 1

However, for procedures, an additional check is added to determine the next value of cp, which is computed as follows:

cp' = (IF depth = 1 THEN any ELSE cp)

where the variable cp is assigned the value of the constant any, if process is not running an atomic statement and otherwise, it is left unassigned.

Break, Goto and Return statements. These statements change the flow control of a process to other locations in an algorithm by changing the value of the program control variable pc. The **break** statement is used to stop the repetitive execution of a **for** or **loop** statement.

break

A **break** statement is translated to intermediate format by replacing it with an assignment statement updating the array pc at a location identified by the variable self. The variable self represents the identity of the process executing that block of statements. The array pc at the location self is updated by the label that follows the construct in which it is used.

pc[self] := ⟨⟨*next label*⟩⟩

The **goto** statement is similar to **break** statement but it can be used anywhere in the algorithm and moves the program control to the label mentioned in the statement. The general form of **goto** statement is as follows:

goto ν;

The above **goto** statement will be translated to intermediate format as follows:

pc[self] := ν

The **return** statement is used within procedures to transfer the program control back to the point where it is called from. Its translation to intermediate format is similar to **break** and **goto** statements that update the variable pc. In **return** statement, the label at which it should return is unknown. To keep track of that label in PlusCal-2, a record is built for that process or main algorithm in the variable **stack** whenever a process or main algorithm calls a procedure. All the local variables of that procedure and the location where it should return after execution is stored in that record. The returning location can be accessed by the variable **stack[self].pc**. Thus the **return** statement in intermediate format will be replaced by an assignment statement for the program control variable, that will be as follows:

pc[self] := stack[self].pc

Apart from the translation of the algorithm to intermediate format, the PlusCal-2 translator also collects other necessary information from the algorithm that includes the variables, translation of definitions, properties, invariants, constraints and fairness conditions. ThePlusCal-2 translator prepares a list of variables along with their initialization data from the PlusCal-2 algorithm. The variables that are declared at the top level in PlusCal-2 algorithm are treated as global variables whereas the variables declared as local variables of a process or procedure are treated differently. They are stored in a new data structure that is a record identified by the name of the process to store the local variables.

The local variables for a procedure are not added to the list of variables in this phase, instead the PlusCal-2 translator saves them inside the symbol table and whenever it encounters a procedure call, it prepares the statements that load the information about that procedure including the variables and their initialization information into the stack. So, the local variables of a procedure are initialized at run time whenever they are called.

3.3.4 PlusCal-2 TLA generation

The main functionality of this phase is to generate two files for the TLC model checker. The first one is the TLA file that contains the TLA^+ specifications for the algorithm and the other one is configuration file that contains the configuration information. All the information that is accumulated during the previous phase is passed on to this phase. It generates a basic structure of a TLA^+ module then it starts adding initialization information for the TLA^+ specifications.

In the next step, it prepares the TLA^+ actions for the final specifications from the algorithm in intermediate format. These TLA^+ actions represent the labeled blocks of statements in the PlusCal-2 algorithm. The blocks of statements in intermediate language are mapped directly to TLA^+ actions. The labels represent the name of the TLA^+ actions whereas the statements are translated to TLA^+ language. A general structure of a TLA^+ action for PlusCal-2 blocks is as follows:

$$\begin{aligned} \langle\langle action\ name\rangle\rangle(self) \triangleq\ & \wedge\ pc[self] = \text{“}\langle\langle action\ name\rangle\rangle'' \\ & \wedge\ cp = \langle\langle any/self\rangle\rangle \\ & \wedge\ \langle\langle action\ description\rangle\rangle \\ & \wedge\ \langle\langle updates\ for\ \text{TLA}^+\ variables\rangle\rangle \end{aligned}$$

A TLA^+ action is identified by its name that represents a label in the PlusCal-2 algorithm. It is followed by a parameter $self$ in parenthesis that carries the identity of the process or main algorithm executing it. Its definition starts with two guards where the first one ensures that it is the next action to be performed by the process or main algorithm and the second one ensures that the process has an access right

to proceed. Then the statements corresponding to a label in PlusCal-2 algorithm are translated to TLA$^+$ language and added to the action definition. Finally, the TLA$^+$ variables that are modified in the action definition are updated with the new values.

Here, we will show how the statements in intermediate language are translated to TLA$^+$ language.

Assignment statement. An assignment statement in intermediate language can be an assignment to a variable or to an array/record. An assignment to a variable is translated using a TLA$^+$ LET IN where as an assignment to an array/record is translated using a LET IN and an EXCEPT forms. The following assignment statement

counter := counter + 1;

will be translated to TLA$^+$ LET IN as follows:

LET _counter $\triangleq$ counter + 1 IN

Now, if we have an array/record update

array[i] := TRUE;

It will be translated as

LET _array $\triangleq$ [array EXCEPT ![i] = TRUE] IN

In PlusCal-2, we allow multiple assignments to the same variable/array/record within one block. Now, if we have multiple assignments as

assign := assign + 1;
assign := assign + 2;

it will be translated to TLA$^+$ as follows:

LET _assign $\triangleq$ assign + 1 IN
LET __assign $\triangleq$ _assign + 2 IN

With statement. In TLA$^+$, the **with** statement is represented as an existential quantification over a set if the expression in the **with** statement is not an assignment to an auxiliary variable.

$\underline{\lambda}$: **with** i $\in$ Set
  B_1
end with

This type of **with** statement that involves execution of B_1 by non-deterministically choosing a value from set **Set** is translated as follows:

$$
\begin{array}{lll}
\lambda(self) & \triangleq & \land \cdots \\
 & & \land \exists\, i \in Set : \\
 & & \quad \land\ {B_1}^* \\
 & & \quad \land\ \cdots
\end{array}
$$

The special type of **with** statement that is assignment to an auxiliary variable is translated to TLA$^+$ language by converting the expression of **with** statement to a TLA$^+$ LETIN statement. Now, if we have following case:

$\underline{\nu}$: **with** i = 500
  B_1
end with

The translation of the above block to TLA$^+$ action will be as follows:

$$
\begin{array}{lll}
\nu(self) & \triangleq & \land \_pc[self] = \text{“}\nu'' \\
 & & \land\ cp = any \\
 & & \land\ \text{LET}\ i \triangleq\ 500\ \text{IN} \\
 & & \quad \land\ {B_1}^* \\
 & & \quad \land\ \cdots
\end{array}
$$

Branch statement. A **branch** statement from intermediate language is represented using a list of disjunctions in TLA$^+$ language. A general **branch** statement as shown below:

branch
  C_1 **then** B_1
or
  C_2 **then** B_2
or
  $\cdots$
or
  C_n **then** B_n
end branch

is translated to TLA$^+$ as follows:

$$
\begin{array}{l}
\lor \land\ C_1 \\
\quad \land\ {B_1}^* \\
\lor \land\ C_2 \\
\quad \land\ {B_2}^* \\
\lor \cdots \\
\lor \land\ C_n \\
\quad \land\ {B_n}^*
\end{array}
$$

where each disjunction is guarded by the conditions C_1, C_2,.. C_n followed by their corresponding blocks of statements. These blocks of statements are also translated to TLA$^+$ language represented as $B_1{}^*$, $B_2{}^*$,.. $B_n{}^*$ in the above translation.

The actions for a process are added as a separate TLA$^+$ definition that defines the transition relation of a process as the disjunction of the actions it may execute. If we take the Leader election algorithm, shown in appendix 1.2, the definition representing the process $Node$, will be as follows:

$$\begin{aligned} \_Node(self) \triangleq\ & \lor\ start(self) \\ & \lor\ forever(self) \end{aligned}$$

where *start* and *forever* are the actions that represents the functionality of the process $Node$.

After generating all the TLA$^+$ actions corresponding to the individual blocks in the intermediate format, the TLA generator defines the overall next-state relation as the disjunction of the transition relations for all process instances, and for the main code section if present. For the Leader election algorithm, shown in appendix 1.2, the next-state relation will be as follows:

$$Next \triangleq\ \lor\ \exists self \in Node : \_Node(self)$$

where $Node$ is a set containing the process identifiers of processes of type Node and $\_Node$ is a definition representing the actions of the process as explained earlier. The PlusCal-2 language does not allow dynamic creation of processes. Thus, the PlusCal-2 compiler can compute the process identifiers statically by adding a definition for each process specifying the range of identities allocated to the process in the TLA$^+$ specifications. It also keeps track of the identifiers already used by other processes in the algorithm. The following definition specifying the range of identities allocated to the process $Node$ will be added to the TLA$^+$ specifications of Leader election algorithm.

$$\begin{aligned} Node \triangleq\ & \text{LET } Node\_start == 1 \\ & \qquad\quad Node\_end == N \\ & \text{IN } Node\_start..Node\_end \end{aligned}$$

Then, it checks the fairness conditions information passed on by the previous phase. It prepares the fairness conditions and adds them to the TLA$^+$ specifications. Fairness condition for the process $\_Node$ in Leader election algorithm is as follows:

$$Fairness \triangleq\ \land\ \forall self \in Node : \mathrm{WF}_{vars}(\_Node(self))$$

where $vars$ is a tuple containing all the variables defined in the TLA$^+$ module by the PlusCal-2 compiler. Fairness conditions for individual actions can also be assumed and are added to the TLA$^+$ specifications. For example, if we had **fair**

with the labels *start* and *forever* in Leader election algorithm then its corresponding fairness condition in TLA$^+$ specification would have been

$$\begin{aligned} Fairness \triangleq\ & \wedge \forall self \in Node : \mathrm{WF}_{vars}(start(self)) \\ & \wedge \forall self \in Node : \mathrm{WF}_{vars}(forever(self)) \end{aligned}$$

After the generation of next-state relation and fairness conditions, the PlusCal-2 TLA generator prepares the overall specifications. For the Leader election algorithm, shown in appendix 1.2, the overall specifications will be:

$$Leader \triangleq Init \wedge \Box[Next]_{vars} \wedge Fairness$$

At the final stage, it reads the information about properties, invariants and constraints. It prepares their definitions and adds them to the TLA$^+$ specifications. For the Leader election algorithm, shown in appendix 1.2, the definition of a temporal property and an invariant will be as follows:

$$Temp0 \triangleq \exists p \in Node : \Diamond Node[p].winner$$

$$\begin{aligned} Inv0 \triangleq\ & \forall p \in Node : Node[p].winner \Rightarrow \\ & (\forall q \in Node \setminus \{p\} :\ \neg Node[q].winner) \end{aligned}$$

It also writes their references in the configuration file to tell TLC model checker to take them into account during model checking. The statements that it adds are as follows:

For the invariant discussed above, it adds,

$$INVARIANT\ \ Inv0$$

For a temporal property discussed above, it adds,

$$PROPERTY\ \ Temp0$$

3.4 Model checking using TLC

Once the PlusCal-2 compiler generates the two files, TLA$^+$ specification file with an extension of *tla* and a configuration file with an extension of *cfg*, then a user can use TLC model checker to model check the specifications for the algorithm. The command used to run TLC is as follows:

java tlc.TLC ⟨⟨file name⟩⟩

For the Leader election algorithm, the command will be as follows:

java tlc.TLC Leader

For the Leader election algorithm without the temporal property, TLC will generate the following results:

```
TLC Version 2.0 of January 16, 2006
Model-checking
Parsing file Leader.tla
Parsing file D:\PlusCal\TLC\src\tlasany\StandardModules\Naturals.tla
Parsing file D:\PlusCal\TLC\src\tlasany\StandardModules\Sequences.tla
Semantic processing of module Naturals
Semantic processing of module Sequences
Semantic processing of module Leader
Finished computing initial states: 1 distinct state generated.
Model checking completed. No error has been found.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
    calculated (optimistic):  1.3471211993132393E-16
    based on the actual fingerprints:  1.4408040857745612E-16
106 states generated, 35 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 16.
```

This result shows that for 3 processes in Leader election algorithm, the TLC generated 106 states out of which 35 were the distinct states. As we increase the number of processes, the number of states increase exponentially, consequently it makes the model checking process slow.

3.5 Summary

System analysis or verification of the system requires an expressive language for modeling a concurrent or distributed system. In this chapter, we discussed that a modeling language must have certain properties e.g., it must be able to express higher-level abstractions of the system. It should also have the ability to express concurrency and provide constructs to add non-determinism in the model of the system. For the verification of liveness properties, it should have the capability to add fairness assumptions and along with these properties the aspect of simplicity should not be compromised.

Then we presented a new language PlusCal-2, inspired by the PlusCal language. PlusCal is a pseudo-code like language for specifying concurrent systems that is simple in use, provides non-deterministic constructs and allows description of the system at higher-level of abstractions. We found various limitations in this language as discussed in chapter 1. One of the major limitations is the knowledge of the TLA$^+$ language for adding fairness assumptions and properties. Thus, we have strived at making our PlusCal-2 models entirely self-contained. We added features such as nested processes, scoped declarations, user-defined grain of atomicity, and fairness assumptions. We believe that the new version PlusCal-2 ensures all the

properties of a modeling language without compromising on any one of them. We also consider that it will be more accessible to users that are not experts in formal methods.

Chapter 4

Partial-order reduction for PlusCal-2 algorithms

Contents

Introduction

Model checking [Clarke 1999] is a popular verification technique for concurrent and distributed algorithms. It provides tools for deciding automatically whether properties (typically expressed in temporal logic) are verified for finite instances of systems or algorithms, described in a formal modeling language. Its main limitation is the well-known state explosion problem, which can be mitigated by verifying algorithms at a high level of abstraction. Various methods have been proposed in the literature to combat the state space explosion problem including symbolic state space representation, efficient memory management strategies, symmetry reduction methods and partial-order reduction methods. Our focus will be on the application of partial-order reduction methods for PlusCal-2 algorithms.

Partial-order reduction methods exploit the commutativity of concurrent transitions, which result in same state when they are executed in different orders. This idea of avoiding redundant transition sequences helps in reducing the size of state space to be explored. Thus, the transition sequences in the reduced state graph are a subset of the transition sequences in the full state graph. A representative transition sequence is selected on the basis of independence relation between the transitions in the transition sequence. This independence relation can either be constant or conditionally defined. In constant independence, the relation is precomputed based

on Definition 3 whereas the conditional independence relation is precomputed in the form of predicates that are evaluated in a given context to determine the relation between the transitions at the current state.

Partial-order methods based on constant independence relations have been widely explored in the literature. However, conditional independence promises to allow for better reductions, and so we investigate their use for the verification of PlusCal-2 models. Computing independencies between the transitions or statements of PlusCal-2 algorithms is substantially easier than inferring them from the resulting "flat" TLA$^+$ models. We therefore decided to extend the compiler by a static analyzer that would produce the necessary information and feed it as an additional input to the model checker, helping it to reduce the state explosion problem.

In this chapter, we will first describe the independence predicates for PlusCal-2 algorithms, then we will explain how we extended the PlusCal-2 compiler to produce the independence predicates. We will also discuss the proposed dynamic partial-order reduction method by Cormac Flanagan and Patrice Godefroid presented in [Flanagan 2005] for conditional independence relation along with TLC model checker with depth-first search.

4.1 Independence Predicates for PlusCal-2

One goal in the design of PlusCal-2 was to enforce variable scoping: although Lamport's PlusCal allows a user to declare variables local to a process, these can still be accessed from other processes. Two actions are independent if they access different variables, and clear scoping rules help us to determine that two PlusCal-2 statements are independent. We now explain more precisely how we compute predicates that ensure that two (blocks of) PlusCal-2 statements are independent in a given state.

4.1.1 Intermediate representation of PlusCal-2 algorithms

The PlusCal-2 compiler first produces an intermediate representation of the given algorithm, which consists of labeled blocks of loop-free guarded commands that will be executed atomically. Each block is then translated to a TLA$^+$ action, and sequencing between blocks is ensured by adding explicit control variables. More precisely, blocks of the intermediate language are given by the following grammar,

where brackets denote optional parts:

block ::= **skip**
| assignment [; block]
| **with** *id* ∈ *expr*
block
end with
| **branch**
C_1 **then** block
or C_2 **then** block
⋮
or C_n **then** block
end branch

Left-hand sides of assignments can be simple variables, array components as in

net[out] := Append(net[out], [type ↦ "one", number ↦ mynumber])

or record components. The **with** statement executes its corresponding block for some value of *id* from the set that is produced by resolving the expression *expr*. The **branch** statement is executable only if some guard (state predicate) C_i is true at the current state.

This intermediate representation allows us to compute the updates that will be performed by the TLA$^+$ actions corresponding to each block. From these, we can derive independence predicates that ensure that all possible updates commute.

4.1.2 Inductive definition of independence predicates

In general, two blocks A and B are independent if they modify different parts of the state space, and if neither reads a variable that may be modified by the other. We will make this intuition more precise by inductively defining a predicate $P_{indep}(A, B)$ that guarantees that blocks A and B are independent at any state satisfying the predicate and where both blocks are enabled. In the definition of $P_{indep}(A, B)$, we make use of an auxiliary predicate $P_{unch}(A, E)$ for a block A and an expression E, which ensures that the value of E is unaffected by the execution of A.

Before we present the formal definition, consider the example where both blocks A and B correspond to the assignment at line 23 of the DKR algorithm in appendix 1.2, executed by two different processes $\_p$ and $\_q$. The representation of this assignment in the intermediate format is

```
net[_Node_data[self].out] :=
    Append(net[_Node_data[self].out],
        [type ↦ "one", number ↦ _Node_data[self].mynumber])
```

where _Node_data is an array of records containing the local variables of the processes including the variable out. It is indexed by a variable self that carries

the identity of a process executing this statement e.g., if it is executed by process $\_p$ then self $= \_p$. The variable self is replaced by the identifiers $\_p$ and $\_q$ for the two blocks A and B before producing the independence predicate to clarify the presentation. We might consider the two assignments to be dependent since both update the same global variable net, and compute the independence predicate FALSE. However, they will actually be independent provided the values of the local variables out of processes $\_p$ and $\_q$ are different, and we may therefore generate the independence predicate

$$\_\text{Node\_data}[\_p].\text{out} \neq \_\text{Node\_data}[\_q].\text{out}.$$

As described earlier, the predicate $P_{unch}(A, E)$ ensures the independence of a block A executed by process $\_p$ that yields a transition and an expression E executed by process $\_q$, now assume A is the block corresponding to the assignment above and that E is the expression

$$\text{net}[\_\text{Node\_data}[\text{self}].\text{out}].\text{type} = \text{“one”},$$

that compares the value at a location in an array with a string value. Now, we can define the predicate $P_{unch}(A, E)$ such that it ensures the locations updated by the transition corresponding to block A are not read by the expression E, thus ensuring the execution of the block A leaves the value of expression E unchanged. The block A updates the global variable net and the expression E reads the same global variable net. Then we can produce the same independence predicate

$$\_\text{Node\_data}[\_p].\text{out} \neq \_\text{Node\_data}[\_q].\text{out}$$

that ensures that the block A does not change any location read by the expression E and can therefore be chosen as the predicate $P_{unch}(A, E)$.

Predicate for skip statement.

Computing independence predicate for a skip statement with any other block B is trivial as the skip statement does not affect the state space of the algorithm. Thus, the independence predicate, $P_{indep}(\text{skip}, B)$, will be TRUE. Similarly, the unchanged predicate, $P_{unch}(\text{skip}, E)$, for a skip statement and an expression E will also be TRUE.

Predicate for two assignments.

To compute the independence predicate, $P_{indep}(A, B)$, for two blocks A and B that are both single assignments, we have to start by ensuring that the two statements do not interfere in each other's execution by introducing conditions on reads and writes of the two assignment statements. If one statement performs a read operation on a variable updated by the other statement then, the two statements will in general not commute as they will result in a different state and they will be marked as dependent (predicate FALSE).

An assignment statement writes a new value to a location identified by the variable that can either be a scalar variable or a location in a record or an array. We can inductively define a location as follows:

$$\text{loc} ::= \langle \text{base}, \langle \text{path}, \text{loc}^* \rangle \rangle$$

where *base* is the name of a scalar variable, a record or an array, *path* is a sequence of expressions e_1, e_2, ... e_n that resolves to an index in the record or array and loc^* is the set of locations, loc_1, loc_2,... loc_m, accessed in the *path* expression.

Now, to ensure that the two statements do not interfere, we must guarantee that one assignment statement does not update a location that is accessed by the other assignment statement, described as follows:

- the location written by assignment A should not be read by assignment B,
- similarly, the location written by assignment B should not be read by assignment A,
- and, they should not write to the same locations.

To formally define the above description, we must first compute the set of locations, rd_A and rd_B, read by each assignment statement A and B respectively, along with the locations, loc_A and loc_B, updated by these assignment statements. For example, if we have following assignment statement

$\mathsf{a}[i] := expr$

The above example writes in the array a that is the base variable, at the index identified by the path expression i that only reads single location i and might also read other locations in the expression *expr* at the right hand side. Thus, an assignment statement writes to the location at left hand side and might read the locations on both sides. If we assume that we have the assignment statement A as shown below:

$\mathsf{a}[i] := q[i + 4]$

Then, loc_A and rd_A will have the following values:

$loc_A = \langle\ \mathsf{a},\ \langle [i],\ i \rangle\ \rangle$
$rd_A = \{\langle\ q,\ \langle [i + 4],\ i \rangle\ \rangle,\ \langle i \rangle\}$

Two locations that represent a scalar variable can easily be distinguished by the name of the base variable as they will always point to different memory locations. Whereas, the difference between two locations that represent an array or a record, e.g., $a[i]$ and $a[j]$, depends on the evaluation of their paths. Once the locations being read and written are identified, then we can use the following conditions that the independence predicate must ensure:

1. $loc_A \notin rd_B$: location loc_A updated by A should not belong to the set of locations, rd_B, read by B.
2. $loc_B \notin rd_A$: location loc_B updated by B should not belong to the set of locations, rd_A, read by A.
3. $loc_A \neq loc_B$: both the statements should update different locations.

We therefore compare pairs of locations $\langle l_1, l_2 \rangle$ and compute predicates P^{l_1,l_2} that ensure that l_1 and l_2 do not denote the same location:

- If the base variables for l_1 and l_2 are different then the locations cannot be the same and $P^{l_1,l_2} = \text{TRUE}$.
- Otherwise, we compute P^{l_1,l_2} by comparing the paths for the two locations.
  - If both the paths have the same number of expressions $e_i^1, \ldots, e_i^n$ (for $i = 1, 2$) then
    $$P^{l_1,l_2} \triangleq \bigvee_{j=1}^{n} e_1^j \neq e_2^j.$$
    In particular, $P^{l_1,l_2} = \text{FALSE}$ if $n = 0$.
  - Otherwise, we define $P^{l_1,l_2} \triangleq \text{FALSE}$.

Finally, the independence predicate $P_{indep}(A, B)$ is the conjunction of all predicates P^{l_1,l_2} for all locations that must be checked to be different:

$$P_{indep}(A, B) \triangleq \Big(\bigwedge_{l \in rd_B} P^{loc_A,l} \Big) \wedge \Big(\bigwedge_{l \in rd_A} P^{l,loc_B} \Big) \wedge P^{loc_A,loc_B}$$

We now define the predicate $P_{unch}(A, E)$ that ensures that the assignment A leaves the expression E unchanged. Clearly, this is the case if the location modified by the assignment A is not read by the expression E, and given the location loc_A updated by A and the set rd_E of locations read by expression E, we set

$$P_{unch}(A, E) \triangleq \bigwedge_{l_E \in rd_E} P^{loc_A,l_E}.$$

We compute a sound approximation of independence predicates and our computation could be refined. For example, the independence predicate for two array updates

$$v[a] := e \qquad \text{and} \qquad v[b] := e'$$

includes the conjunct $a \neq b$. In fact, the two assignments can be independent if $a = b$ but also $e = e'$ and each assignment leaves the right-hand side of the other assignment unchanged. Since the right-hand sides of assignments are often complex expressions whereas the left-hand sides are usually simple, we chose not to implement this improvement in order to keep independence predicates small.

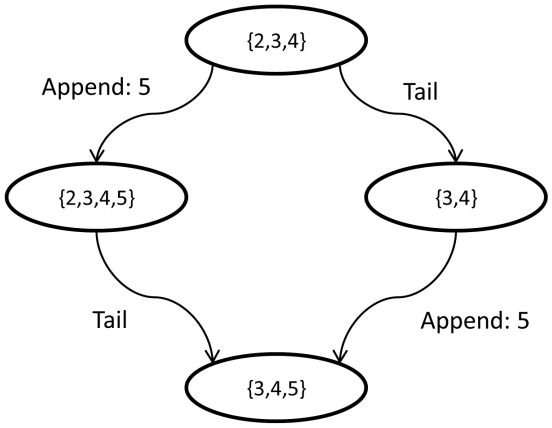

Figure 4.1: Append and Tail commute over non-empty sequences.

Other refinements depend on the concrete operations that appear in the right-hand sides. For example, FIFO channels are represented in TLA$^+$ (and PlusCal-2) by sequences, where message sending corresponds to appending at the end of the sequence, and message reception to removing the first element of the sequence by the Tail operation. If both actions are actually enabled, the sequence must be non-empty, and the two actions indeed commute, as illustrated in Fig. 4.1. Since these operations occur frequently in the algorithms we consider, we implement this optimization.

Independence predicates for sequential composition.
Independence predicates for complex blocks are computed recursively. In particular, assume that block A is of the form

$$lhs := e;\ A'$$

and that we have already computed the independence predicates P_1 for the leading assignment and B and P_2 for the blocks A' and B. The overall independence predicate, $P_{indep}(A, B)$, is the conjunction $P_1 \wedge P_2'$ where P_2' is obtained by replacing the base variable of *lhs* by the value of that variable after assignment. Considering again the assignment of the running example, we obtain

$$\begin{array}{lll} P_2' \triangleq & \text{LET} & \_\mathsf{net} \triangleq [\mathsf{net}\ \text{EXCEPT}\ ![\_\mathsf{Node\_data}[\_p].\mathsf{out}] \\ & & \qquad = \mathsf{Append}(@, [\mathsf{type} \mapsto \text{“one”}, \mathsf{number} \mapsto \mathsf{mynumber}])] \\ & \text{IN} & P_2[\_\mathsf{net}/\mathsf{net}]. \end{array}$$

To compute the predicate $P_{unch}(A, E)$ for block A as defined above and an expression E, assume that we have already computed the predicates P_1 and P_2. P_1 for the leading assignment and the expression E, $P_{unch}(lhs := e, E)$ and P_2 for the block A' and the expression E, $P_{unch}(A', E)$. Then, $P_{unch}(A, E)$ will be the

conjunction of these two predicates, $P_1 \wedge P_2'$, where P_2'

$$P_2' \triangleq \begin{array}{ll} \text{LET} & \_\mathsf{lhs} \triangleq \mathsf{e} \\ \text{IN} & P_2[\_\mathsf{lhs}/\mathsf{lhs}]. \end{array}$$

Independence predicates for branches.

Now assume that block A is of the form

$$\begin{array}{lll} \textbf{branch} & & \\ & C_1 \ \textbf{then} & A_1 \\ \textbf{or} & C_2 \ \textbf{then} & A_2 \\ \textbf{end} & & \end{array}$$

(the generalization to a branch block with n arms will be obvious). The overall independence predicate for a branch must ensure the following conditions:

1. whenever C_1 holds (and therefore A_1 may be executed), A_1 and B are independent,
2. the symmetric condition for A_2 and B, and
3. executing B cannot disable any execution of A that would have been possible in the original source state, or enable an execution that would have been impossible.

Assume that we have already computed independence predicates $P_{indep}(A_1, B)$ and $P_{indep}(A_2, B)$ that ensure independence of A_1 and A_2 with B. These predicates will be used for ensuring conditions (1) and (2). For condition (3) to hold, we require that the conditions C_1 and C_2 are unaffected by any execution of B. We therefore obtain the overall independence predicate

$$\begin{array}{l} \wedge\ C_1 \Rightarrow P_{indep}(A_1, B) \\ \wedge\ C_2 \Rightarrow P_{indep}(A_2, B) \\ \wedge\ P_{unch}(B, C_1) \wedge P_{unch}(B, C_2) \end{array}$$

The unchanged predicate $P_{unch}(A, E)$ for the block A and any expression E is given by the conjunction

$$\begin{array}{l} \wedge\ C_1 \Rightarrow P_{unch}(A_1, E) \\ \wedge\ C_2 \Rightarrow P_{unch}(A_2, E) \end{array}$$

Independence predicates for with statement.

To compute the independence predicate $P_{indep}(A, B)$ for a block of with statement

$$A \ \equiv\ \mathsf{with}\ id \in expr\ A_1\ \mathsf{end\ with}$$

where A_1 is the sub-block that is to be executed for some value of identifier id in the set obtained by evaluating $expr$, we again assume that we already have computed the independence predicate $P_{indep}(A_1, B)$. The predicate $P_{indep}(A, B)$ must ensure the following conditions:

- A_1 (for any value of id) should be independent of block B and
- execution of B should leave unchanged the value of expression $expr$.

These two conditions suggest the definition of $P_{indep}(A, B)$ as

$$\begin{array}{l} \land\ P_{unch}(B, expr) \\ \land\ \forall id \in expr : P_{indep}(A_1, B) \end{array}$$

Similarly, the unchanged predicate $P_{unch}(A, E)$ can be defined as

$$\forall id \in expr : P_{unch}(A_1, E).$$

Generating a matrix of independence predicates.

When computing the independence predicates P_{ij} for pairs of atomic blocks A_i and B_j, we perform some elementary simplification, and in particular propagation of constants TRUE and FALSE. We then define a TLA$^+$ operator that represents the matrix of independence predicates, and that will be passed to the model checker. The operator takes four parameters, which correspond to the names of the blocks and the process identifiers $\_p$ and $\_q$, and is defined as

$$\begin{array}{l} IndepMatrix(\_p, \_q, A, B) \triangleq \\ \quad \text{CASE} \\ \qquad\quad A = name_1 \land B = name_1 \ \to\ P_{11} \\ \qquad \square \quad A = name_1 \land B = name_2 \ \to\ P_{12} \\ \qquad\quad \vdots \\ \qquad \square \quad A = name_n \land B = name_n \ \to\ P_{nn} \end{array}$$

where $name_1, \ldots, name_n$ are the names of the TLA$^+$ actions that are generated for the atomic blocks of the PlusCal-2 algorithm in intermediate representation. (In the actual implementation, we only give those entries of the matrix for which P_{ij} is different from FALSE, and add a catch-all clause that returns FALSE for all other inputs.)

The correctness of the independence predicates for every combination of blocks can be established by the following theorem:

Theorem 1. *At any given state where any two blocks of statements are enabled and the independence predicate holds, then they are independent at that state.*

Before presenting the proof of the above theorem we will present a lemma that we will use in the proof of the theorem.

Lemma 1. *Two blocks of statements enabled at any state belong to two different processes.*

Proof. We prove the above lemma by contradiction that two blocks of statements enabled at any state belong to the same process. Now, assume that two blocks of statements B_1 and B_2 enabled at state s belong to same process such that $proc(B_1) = proc(B_2)$. In the specifications generated by PlusCal-2 for TLC, we have a global array _pc that contains one location per process in the entire algorithm. It stores the name of the next block of statements to be executed for each process and is indexed by the identity of the process. Thus, every block $\lambda : B$ in TLA$^+$ representation has a guard $\_pc[self] = \text{“}\lambda''$, and since there is a one-to-one correspondence between blocks and their labels, there can never be two enabled blocks at any state. This contradicts the assumption that the two blocks B_1 and B_2 are from same process. Hence, $proc(B_1) \neq proc(B_2)$. □

Now, we present the proof of Theorem 1.

Proof. We present the proof of Theorem 1 by induction on the definition of blocks according to the grammar of the intermediate format.

Base case. Assume that we have two assignment statements corresponding to two blocks B_1 and B_2 enabled at state s and they belong to two different processes using Lemma 1. To ensure that they commute, the memory location updated by one assignment statement must not be read by the other assignment statement. Similarly, the memory locations updated by both the assignment statements must be different. Now, using the definition of conditional independence, two blocks that are enabled at a state and they commute, are independent at that state.

Inductive case. For the inductive case, we will reconsider the different cases that we used to compute independence predicates.

- skip statement: Assume that we have two blocks, B_1 that contains a skip statement, and B_2 is any other block. Using lemma 1, we can assume that they belong to two different processes. As the skip statement does not modify the state space, thus its independence predicate with any other block will be TRUE. Now, using the definition of conditional independence, the two blocks enabled at a state commute and they are independent at that state.

- Sequential composition of statements: Assume that we have two blocks B_1 and B_2 enabled at state s and they belong to two different processes using lemma 1. B_1 is a complex block containing sequential statements and B_2 is any block of statements. Now, to ensure that they commute, the memory locations updated and read by block B_1 should not be updated and read by the other block B_2. Using the definition of conditional independence, these two blocks that are enabled at a state and they commute, are independent at that state.

- branch statement: Assume that we have two blocks, B_1 that represents a branch statement and B_2 is any other block, enabled at a state and belong

to two different processes using lemma 1. To ensure that they commute, the memory locations read by B_1 in the conditions of the branch statements and in the blocks corresponding to each condition should not be updated by B_2. Similarly, the locations updated by B_2 should not be updated by B_1 and the locations updated by both the blocks should not overlap. Now, using the definition of conditional independence, the two blocks that are enabled at a state and they commute, are independent at that state.

- with statement: Assume that we have two blocks, B_1 that represents a with statement

  with $id \in expr$ B end with

  and B_2 is any other block, enabled at a state and belong to different processes using lemma 1. Now, to ensure that they commute, assume that B is independent with B_2 and the locations read in the expression *expr* are not updated by B_2. Now, using the definition of conditional independence, the two blocks enabled at a state commute and are independent at that state.

□

4.2 Extension to PlusCal-2 Compiler

The extended PlusCal-2 compiler is shown in figure 4.2. As the independence data collection is performed on the PlusCal-2 algorithm in intermediate format, thus the compiler performs the extraction process at the intermediate stage of the compilation. When the algorithm is in the intermediate format, the blocks of statements that will represent actions in the TLA$^+$ specifications are clearly identified. Once the blocks are identified, we can compare each pair of block to find out their corresponding independence predicate.

The extended PlusCal-2 compiler takes the algorithm in the intermediate format that is composed of blocks of statements. These blocks of statements are then paired to compute their corresponding independence predicate. In the figure, it shows that it generates a pair of blocks and then computes its independence predicate. This computation is shown in detail in the figure 4.3 that further explains the computation process. For the algorithm in intermediate format, it repeatedly generates the pair of blocks for all the available blocks of statements and once all the corresponding independence predicates are generated, it finally computes the independence matrix. The independence matrix is a 2-dimensional array that is labeled by the blocks of statements and contains their corresponding independence predicate. The independence predicates are normalized as well to simplify their presentation in TLA$^+$ specifications.

To explain the implementation process in detail, we will take the blocks of statements from the PlusCal-2 sorting algorithm shown in appendix 1.3.1 and its corresponding intermediate format for the labeled blocks is shown in appendix 1.3.2. The sorting algorithm contains three processes, left, middle and right, and it has

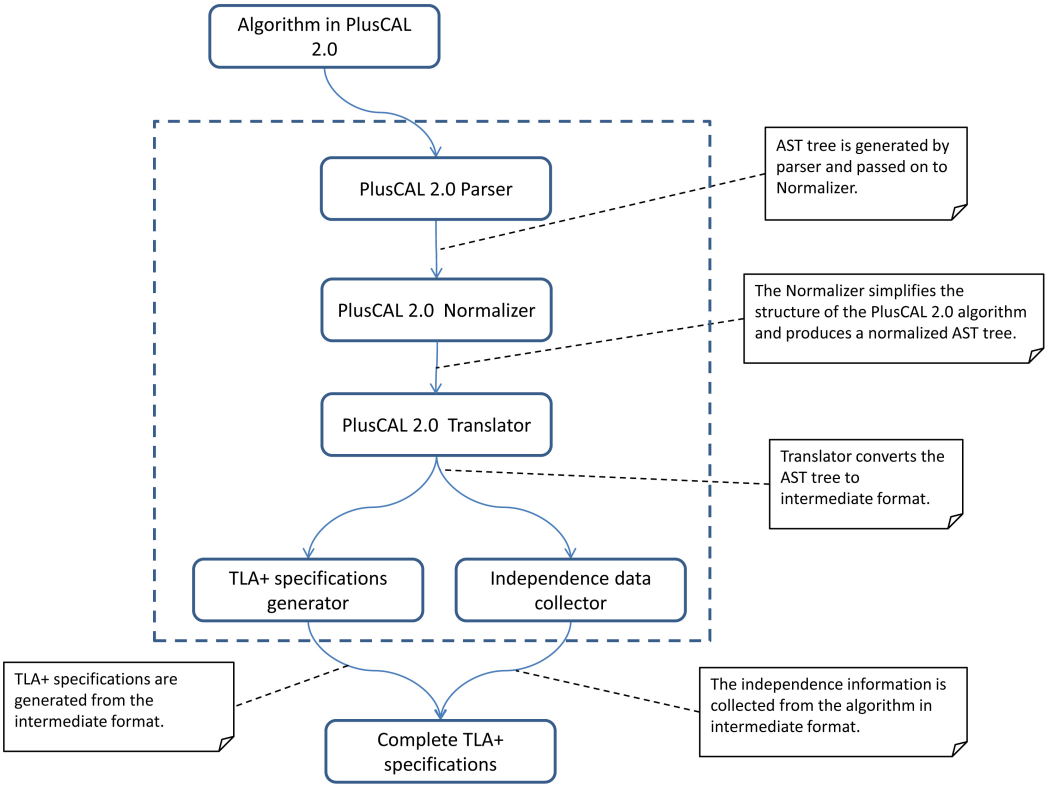

Figure 4.2: The compilation phases for PlusCal-2.

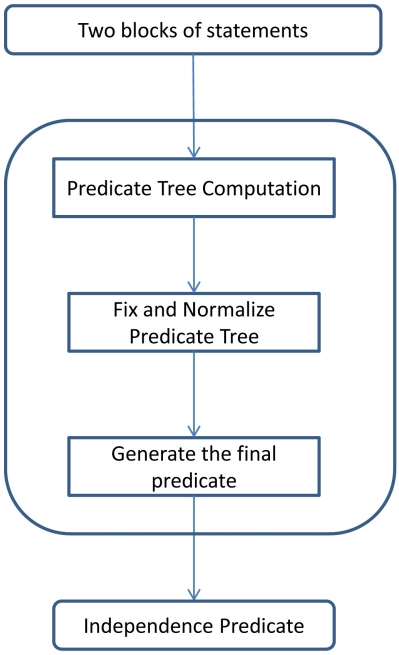

Figure 4.3: Computation of independence predicate for two blocks.

four blocks of statements that are left-lbl, start, mid-lbl and right-lbl. At this step, we produce a list of pairs of blocks of statements and in the next step an independence predicate is produced for each pair of blocks.

For each of the pairs of blocks, we compute its corresponding independence predicate by comparing them and finding out the conditions under which they can be considered as independent. Here, we will show how we compute those predicates for the pair of blocks left-lbl and left-lbl. It is necessary to compare a block with itself as two instances of a same process can execute the same block at the same time. We assume that the blocks are always executed by two different processes whose identifies are referred as _p and _q . To compare the block left-lbl with itself, referred in our text as A and B, we will start comparing each statement for block A, process identity _p, with each statement in the block B, process identity _q. For both the blocks A and B, the first statement is

```
network[_left_data[self].out] :=
    Append(network[_left_data[self].out], [value ↦ _left_data[self].seed])
```

network is a global array in the sorting algorithm that is used for sending messages to other processes. Both the statements will be updating this globally defined array at the location identified by a process local variable out that is represented as _left_data[self].out in the intermediate format. We must also check if this global array is read by any of these statements and in our example both of the statements read the array updated by the other one. To ensure the independence of these two

statements, we must have a predicate that ensures that these locations are different. This can be written as

$$\_left\_data[\_p].out \neq \_left\_data[\_q].out.$$

where _p and _q replace the variable self that was carrying the identity of the process executing the block of statements. Here, the above predicate says, that the value of the variable out for process _p must be different from the value of the variable out for process _q.

Then, we take the next pair of statements by moving to the second statement in block B executed by process q, that is:

```
_left_data[_q].counter := (_left_data[_q].counter + 1)
```

As discussed earlier, the first statement in the pair modifies a global array called network. The second statement updates a process local variable counter. As they both modify different variables and a statement does not read the variable modified by the other statement. Thus, they both can be considered independent of each other and we can generate TRUE independence predicate. Similarly, we take the next pair of statements by moving to the third statement in block B and computing its independence predicate with first statement in block A. The third statement in block B executed by process q is:

```
_left_data[_q].seed := RANDOM(_left_data[_q].seed)
```

This statement by block B updates a process local variable seed that is different from the variable seed read by the other statement in block A. Indeed, our definitions yield the independence predicate $\_p \neq \_q$, for this pair of statements. The fourth statement in the block B is a branch statement as follows:

```
branch
(_left_data[_q].counter = N) then
     _pc[_q] := "Done"
or
(_left_data[_q].counter # N) then
     _pc[_q] := "left-lbl"
end branch
```

A branch statement is composed of multiple conditions and each condition is followed by its corresponding block of statements that is executed if that condition holds. Now, to compare an assignment statement with a branch statement we must make a comparison of the assignment statement with all the blocks of statements to produce independence predicates and a comparison of the assignment statement with all the conditions to produce unchanged predicate.

We will start by the computation of unchanged predicate by comparing the assignment statement with all the conditions in the branch statement. The assignment

statement updates a global variable network and this variable is not read by any of the conditions in the branch statement. Therefore, for both the conditions we will produce the TRUE independence predicate.

The blocks of statements associated with each condition in the branch statement only contain the assignment statement for the _pc variable. Therefore, we produce TRUE independence predicate for both pairs of assignment statement and blocks. As we have completed the comparison of first statement in block A with block B, now we will pick the second statement in block A that is

```
_left_data[_p].counter := (_left_data[_p].counter + 1)
```

It's comparison with the first and the third statement in block B will result in TRUE predicate whereas its comparison with second statement will result in the predicate

$$\_p \neq \_q.$$

The fourth statement in the block B is a branch statement and we will start the comparison by computing the unchanged predicate by comparing the assignment statement with both the conditions in the branch statement. The resulting unchanged predicate for both the conditions will be $\_p \neq \_q$. Then, as both the block of statements corresponding to the conditions only contain the assignment statement for the variable _pc, thus we produce the TRUE independence predicate for both the blocks.

Now, as we have finished the comparison of second statement in block A, we will pick the third statement that is

```
_left_data[_p].seed := RANDOM(_left_data[_p].seed)
```

The comparison of this statement with first and third statement in block B will result in an independence predicate $\_p \neq \_q$ whereas its comparison with second statement will result in TRUE independence predicate. Finally, its comparison with the last statement in block B will also result in TRUE independence predicate as none of the locations updated and read by the assignment of block A are updated or read by the branch statement in block B.

The final statement in block A is a branch statement as follows

```
branch
(_left_data[_p].counter = N) then
    _pc[_p] := "Done"
or
(_left_data[_p].counter # N) then
    _pc[_p] := "left-lbl"
end branch
```

It's comparison with the first and third statement in block B will result in an independence predicate, TRUE. The second statement in block B updates the

variable counter and it is read in both the conditions of the branch statements. Therefore, the unchanged predicate for conditions and the assignment statement will be _p ≠ _q. Both the blocks of statements in the branch statement only update _pc, thus we generate TRUE independence predicate for both the blocks.

Finally, we compare the branch statements for both the blocks A and B. Both these statements only update the global variable _pc at locations _p and _q thus we produce the independence predicate _p ≠ _q for their blocks of statements.

After normalization of the overall independence predicate for the blocks left-lbl and left-lbl, we will obtain the independence predicate as follows:

```
LET  _left_data ≜ [_left_data EXCEPT
                        ![_q].counter = (_left_data[_q].counter + 1),
                        ![_p].counter = (_left_data[_p].counter + 1)]
IN   ∧ (_left_data[_p].out) ≠ (_left_data[_q].out)
```

In the above overall independence predicate, the predicates $\_p \neq \_q$ are removed (i.e., replaced by TRUE) because the overall operator will only be called when the values for $\_p$ and $\_q$ are different.

4.3 Dynamic partial-order Reduction with Conditional Independence

A PlusCal-2 algorithm is translated to TLA$^+$ language that is supported by the model checker TLC. The main difficulty for adapting TLC model checker to support partial-order reduction with conditional independence predicates is the breadth-first search method that is used to explore all the possible states. It complicates the computation of the persistent set for any state s that requires construction of the set using the depth-first search from that state s. Secondly, in breadth-first search, the stack that contains the current transition sequence is not easily accessible. TLC provides a notion to retrieve the current transition sequence using a simply linked list that is implemented in such a way that each state knows about its parent state in the state space. In this way, one can easily trace back to the initial state. However, this information is not enough for partial-order reduction methods, that support conditional independence. They require complete knowledge about the states and transitions in the current transition sequence.

As breadth-first search does not have the notion of stack, this means that one would have to store complete information about all the states. This will in turn increase the space complexity required to store the entire state space and the time complexity to retrieve each state from the disk. Thus, breadth-first search only supports partial-order reduction with constant dependency as we will show in detail in the chapter 5.

In this section we propose an adaptation of dynamic partial-order reduction method by Cormac Flanagan and Patrice Godefroid in [Flanagan 2005], which could

be implemented in a variant of TLC model checker that supports depth-first search strategy for exploration.

4.3.1 TLC with depth-first search

The TLC model checker performs a breadth-first search to traverse the state graph. Its starts by generating all the initial states and adds them to FIFO queue, then launches threads which repeatedly execute the process described in the chapter 2. In this section, we propose a pseudo-code of the depth-first search for TLC model checker. The algorithm that we propose is single-threaded and below, we have the pseudo-code for the initialization of the algorithm and the thread as follows:

Initialization:

1. for each initial state s not in the list of visited states,
   (a) if state s satisfies the invariant properties then add it to the *Stack* and the list of visited states,
   (b) if it does not, then print the corresponding counter example and stop further execution.
   (c) call the thread.

Thread:

1. pick the topmost state s from the *Stack*,
2. generate all the successors for the state s,
3. for each successor state s' not in the list of visited states,
   (a) if state s' satisfies the invariant properties then add it to the *Stack* and the list of visited states,
   (b) if it does not, then print the corresponding counter example and stop further execution of the model checker.
   (c) call the thread.
4. return.

4.3.2 Dynamic partial-order reduction

The reduction method that we propose for TLC model checker with depth-first search is dynamic partial-order reduction method by Cormac Flanagan and Patrice Godefroid in [Flanagan 2005]. This technique starts by exploring an arbitrary interleaving of some of the concurrent processes. During this exploration, it dynamically tracks interactions between them to identify backtracking points. These backtracking points refer to the alternative paths in the state space that need to be explored.

In practice, the interaction between the transitions can be dependent in one context and independent in other context. Conditional dependency relation was introduced to relax the restriction on a pair of transitions to be dependent all the time even if they become independent in some context during the execution. This partial-order reduction method can easily adapt to the dynamic changes and it does not rely on the static analysis of the algorithm. Thus, it can accommodate new changes at runtime and it can make use of conditional independence predicates that can serve as an additional check to find out more accurate information about (in)dependence between transitions. These conditional independence predicates are evaluated during execution for given states.

To reason about the equivalence class represented by a transition sequence, the authors in the algorithm maintain a *happens-before* relation on the transitions in the transition sequence. If a transition sequence contains independent adjacent transitions then that transition sequence represents an equivalence class of sequences that can be obtained by swapping independent adjacent transitions. For a transition sequence $S = t_1 \dots t_n$, a *happens-before* relation is the smallest relation on $\{1, \dots, n\}$ such that

1. if $k \leq j$ and S_k is dependent with S_j then $k \rightarrow_S j$,
2. $\rightarrow_S$ is transitively closed.

Another important relation used in the algorithm is a variant of *happens-before* to identify the backtracking points during the search. We adapted this relation for our variant of algorithm and it is written as $i \rightarrow_S a$, and holds for $i \in dom(S)$ and action a if either

1. $proc(S_i) = proc(a)$ or
2. there exists $k \in \{i + 1, \dots, n\}$ such that $i \rightarrow_S k$ and $proc(S_k) = proc(a)$.

In TLA$^+$, the notion of processes is hidden and at any state, they are represented by their corresponding actions enabled or disabled at that state. However, for TLA$^+$ specifications compiled from PlusCal-2 algorithms, process identities can be retrieved and used in the partial-order reduction algorithm. Below we explain the adapted dynamic partial-order reduction algorithm in detail.

The Algorithm

The main functionality of the dynamic partial-order reduction algorithm remains the same as in [Flanagan 2005]. The algorithm maintains a transition sequence S starting from the initial state s_0 as in a traditional depth-first search algorithm. The algorithm starts with an empty transition sequence and at each state starting from initial state s_0, it picks any one transition or action a at line 12 to further explore the state space. Before exploring any transition from the current state s, the algorithm computes any new backtracking points for the previous states visited in the current transitions sequence S. This is carried out by a for loop at line 3.

```
1  Explore(S) {
2  let s = last(S);
3    for all the transitions/actions a enabled at s {
4      if ∃ i = max({i, dom(S) | S_i at state pre(S, i) is conditionally dependent and
5                                    may be co-enabled with a and i ↛_S a}) {
6        let E = {b ∈ enabled(pre(S, i)) | b = a or ∃ j ∈ dom(S) : j > i
7                                                  and proc(b) = proc(S_j) and j →_S a};
8        if (E ≠ ∅) then add any b ∈ E to backtrack(pre(S, i));
9        else add all b ∈ enabled(pre(S, i)) to backtrack(pre(S, i));
10     }
11   }
12   if (∃ a ∈ enabled(s)) {
13     backtrack(s) := {a};
14     let done = ∅;
15     while (∃ b ∈ (backtrack(s) \ done)) {
16       add b to done;
17       for all the variants v of b that lead to different successor states
18         Explore(S.v);
19     }
20   }
21 }
```

The *for* loop picks a transition a and tries to find the last transition S_i in the transition sequence S that is conditionally dependent and may be co-enabled with the transition a, and such that $i \not\rightarrow_S a$. Conditional dependence is computed by evaluating the independence predicate for the pair of transitions S_i and a at state $pre(S, i)$ that is the state from which S_i was executed. If the pair of transitions are found to be dependent at the state $pre(S, i)$ then a backtracking set E is computed at line 6. If backtrack set E is nonempty then one of processes in E is added to the backtrack set of the state from which S_i was executed otherwise all of the enabled processes are added.

Then, at line 12, the algorithm picks the next transition to be explored from the current state. It adds it to the backtrack set of the current state and starts a loop at line 15 over all the transitions that will be added to the backtrack set in the future. As an action or a transition in TLA$^+$ can lead to multiple successor states, thus all those successor states must be explored. The *for* loop at line 17 ensures that all those paths are explored by the algorithm.

The algorithm computes the backtrack sets for the states visited in the current sequence thus reducing the actual size of the state space to be explored. The set of transitions that are explored at each state form a persistent set at that state and the detection of deadlock and safety property violation is also guaranteed. This dynamic partial-order reduction algorithm is presented for constant dependency relation between the transitions in the paper by Cormac Flanagan and Patrice Godefroid in [Flanagan 2005]. However, as mentioned in the paper, it is possible to add con-

ditions in the form of predicates that can be easily checked for any given state. The conditional independence predicates does not add any fundamental change in the functionality of the algorithm and the time required for the evaluation of these predicates would be negligible. Thus the correctness of this dynamic partial-order reduction algorithm can still be ensured by the following theorem:

Theorem 2. *Whenever a state s is backtracked during the search performed by the algorithm in an acyclic state space, the set T of transitions that have been explored from s is a persistent set in s.*

The original proof of the above theorem does not require any fundamental change for the conditional dependency relation. The difference is the evaluation of the independence predicate at the state for which the backtrack set is being computed. Here, we present the proof from [Flanagan 2005] adapted for our variant of dynamic partial-order reduction algorithm.

Let A_G denote the state space of the system and let s_0 denote its unique initial state. The postcondition for a transition sequence S and a transition a refers to the lines 4 to 9 in the algorithm,

$PC(S, j, a)$ is defined as
if
    S is a transition sequence from s_0 in A_G and $i = max(\{i \in dom(S)|$
    S_i at state $pre(S, i)$ is conditionally dependent and co-enabled with
    a and $i \not\rightarrow_S a\})$ and $i \leq j$
then
    if $E(S, i, a) \neq \emptyset$
    then $backtrack(pre(S, i)) \cap E(S, i, a) \neq \emptyset$
    else $backtrack(pre(S, i)) = enabled(pre(S, i))$

where $E(S, i, a)$ is a function to compute the backtrack set that refers to the condition at line 6 in the algorithm. If a transition a has a dependency with transitions S_i in transition sequence S, then the function to compute the backtrack set $E(S, i, a)$ for the state $pre(S, i)$ is defined as

$E(S, i, a)$:
$\{\ q \in enabled(pre(S, i))\ |\ q = a$ or
    $\exists j \in dom(S) : j > i$ and $proc(q) = proc(S_j)$ and $j \rightarrow_S a\}$

Now, the postcondition PC for all the transitions/actions a and their corresponding transition sequences w in the algorithm $Explore(S)$ can be defined as

$\forall\ a\ \forall w : PC(S.w, |S|, a)$

Now, we will start the proof of the theorem by a lemma

Lemma 2. *Whenever a state s reached after a transition sequence S is backtracked during the search performed by the dynamic partial-order reduction algorithm, the set T of transitions that have been explored from s is a persistent set in s, provided the postcondition PC holds for every recursive call Explore(S.t) for all* $t \in T$.

Proof. Let s be the last state reached in the transition sequence S denoted as $last(S)$ and T be the set of transitions in the backtrack set of state s. We will prove the above lemma by contradiction assuming that a transition t_n is dependent with some transition $t \in T$. Assume that there exists $t_1, \cdots, t_n \notin T$ such that:

1. $S.t_1 \cdots t_n$ is a transition sequence from s_0 in A_G and
2. $t_1, \cdots, t_{n-1}$ are all independent with T.

where $S.t_1 \cdots t_n$ refers to extending the transition sequence S using the transitions $t_1 \cdots t_n$. Using the property of conditional independence, we can infer that t is enabled in the state $last(S.t_1 \cdots t_{n-1})$ and hence co-enabled with t_n. Without loss of generality, assume that $t_1 \cdots t_n$ is the shortest sequence. Thus, we have

$$\forall 1 \leq i < n : i \rightarrow_{t_1 \cdots t_{n-1}} n$$

(If this was not true for some i, the same transition sequence without i would also satisfy the assumptions and be shorter.) Let ω denote the resulting (possibly empty) transition sequence produced by removing from $t_1 \cdots t_{n-1}$ all the transitions t_i (if any) such that

$$i \not\rightarrow_{t_1 \cdots t_{n-1}} t_n$$

As $S.t_1 \cdots t_n$ is a transition sequence, $S.\omega$ is itself a transition sequence from s_0 in A_G. Although t_n is enabled in $last(S.t_1 \cdots t_{n-1})$, t_n may no longer be enabled in $last(S.\omega)$, but this does not matter for the proof.

Now, if $proc(t) = proc(t_n)$ then this implies that $t = t_n$ (by Lemma 1), conflicting with the assumptions that $t \in T$ but $t_n \notin T$. Hence $proc(t) \neq proc(t_n)$.

Since t is executed by a different process than t_n and since t_n is independent with all the transitions in ω, then $S.\omega$, $S.\omega.t$ and $S.t.\omega$ belong to same equivalence class.

Let $i = |S| + 1$, and consider the postcondition $PC(S.t.\omega, i, t_n)$ for the recursive call Explore($S.t$). Clearly,

$$i \not\rightarrow_{S.t.\omega} t_n$$

(since t is in a different process than t_n and since t is independent with $t_1, \cdots, t_{n-1}$). In addition, we have (by definition of E):

$$E(S.t.\omega, i, t_n) \subseteq \{t_1, \cdots, t_{n-1}, t_n\} \cap enabled(s)$$

Moreover, we have

$$\forall j \in dom(S.\omega) : j > i \Rightarrow j \rightarrow_{S.t.\omega} t_n$$

Hence, by PC for the recursive call Explore($S.t$), either $E(S.t.\omega, i, t_n)$ is nonempty and at least one process from $E(S.t.\omega, i, t_n)$ is in $backtrack(s)$, or $E(S.t.\omega, i, t_n)$ is empty and all the processes enabled in s are in $backtrack(s)$. In either cases, at least one transitions among $\{t_1, \cdots, t_n\}$ is in T. This contradicts the assumption that $t_1, \cdots, t_n \notin T$. □

Now, using the above lemma we present the proof of Theorem 2 adapted for conditional independence relation from [Flanagan 2005].

Proof. Let s be the last state reached in the transition sequence S denoted as $last(S)$ and T be the set of transitions in the backtrack set of state s. The proof is by induction on the order in which states are backtracked.

Base case. Since the state space A_G is acyclic and since the search is performed in depth-first order, the first backtracked state must be a deadlock where no transition is enabled. Therefore, the postcondition for that state becomes $\forall\ a\ : PC(S, |S|, a)$.

Inductive case. Assume that each recursive call to Explore($S.t$) satisfies its postcondition and T is a persistent set in s using Lemma 2. Now, we show that Explore(S) ensures its postcondition for any a and ω such that $S.\omega$ is a transition sequence from s_0 in A_G.

1. Suppose that some transition in ω is dependent with some transition in T. In this case, we split ω into $X.t.Y$, where all the transitions in X are independent with all the transitions in T and t is the first transition in ω that is dependent with some transition in T. Since T is a persistent set in s, t must be in T (otherwise, T would not be persistent in s). Therefore, t is independent with all the transitions in X. Using the property of conditional independence, it follows that the transition sequence $t.X.Y$ is executable from s. By applying the inductive hypothesis to the recursive call Explore($S.t$), we know

   $$\forall\ a\ : PC(S.t.X.Y, |S| + 1, a)$$

   which implies by the definition of PC

   $$\forall\ a\ : PC(S.t.X.Y, |S|, a)$$

   Since t is conditionally independent with all the transitions in X, we also have

   $$\forall\ i \in dom(S.t.X.Y)\ : i \rightarrow_{S.t.X.Y} a \text{ iff } i \rightarrow_{S.X.t.Y} a$$

   Therefore, by definition

$PC(S.t.X.Y, |S|, a)$ iff $PC(S.X.t.Y, |S|, a)$

We can thus conclude that

$\forall\ a\ : PC(S.X.t.Y, |S|, a)$

2. Suppose that all the transitions in ω are independent with all the transitions in T and $a \in backtrack(s)$. Then,

   (a) $a \in T$;

   (b) a is independent with all the transitions in ω;

   (c) $proc(a)$ is a different process from any transition in ω;

   (d) a is enabled at $last(S.\omega)$ and $last(S)$;

   (e) $\forall i \in dom(S) : i \rightarrow_{S.\omega} a$ iff $i \rightarrow_S a$.

   Thus, we have $PC(S.\omega, |S|, a)$ iff $PC(S, |S|, a)$.

3. Suppose that all the transitions in ω are independent with all the transitions in T and $a \notin backtrack(s)$. Pick any $t \in T$. We then have that

   (a) $proc(t) \neq proc(a)$;

   (b) t independent with all the transitions in ω;

   (c) a is enabled at $last(S.\omega)$ and $last(S.t.\omega)$;

   (d) $\forall i \in dom(S) : i \rightarrow_{S.\omega} a$ iff $i \rightarrow_{S.t.\omega} a$.

   Thus, we have $PC(S.\omega, |S|, a)$ iff $PC(S.t.\omega, |S|, a)$. By applying the inductive hypothesis to the recursive call Explore($S.t$), we know

   $\forall\ a\ : PC(S.t.\omega, |S| + 1, a)$

   which implies by definition of PC that

   $\forall\ a\ : PC(S.t.\omega, |S|, a)$

   which in turn implies that

   $\forall\ a\ : PC(S.\omega, |S| + 1, a)$, as required.

□

4.4 Summary

Conditional independence predicates can be used to achieve successful partial-order reduction results for algorithms that perform operations on arrays, records, etc. The independence of these operations can only be detected at the time of execution and statically computed independence relation would be too conservative to be useful as the relation is fixed before performing model checking process. Thus, in this chapter, we presented how the independence predicates can be computed for PlusCal-2 algorithms. Then, we described how we extended the PlusCal-2 compiler to produce these independence predicates.

We proposed an adaptation of the dynamic partial-order reduction algorithm due to Cormac Flanagan and Patrice Godefroid, based on depth-first search, that could make use of the independence predicates generated by our PlusCal-2 compiler. We did not implement this algorithm, however, because it would require major changes to TLC, which is based on a breadth-first state exploration.

Chapter 5

Static partial-order reduction for TLC

Contents

Introduction

In practice, not all concurrent and distributed algorithms are composed of entities that use the shared memory locations for their local computations. Such systems do not require fine-grained analysis of their atomic transitions to determine independence relation as the relation remains constant and it can be determined statically that the processes access different locations or not. In the previous chapter, we focused on computing useful independence predicates for the concurrent and distributed algorithms using shared memory that leads a pair of transitions to be dependent in one context and independent in the other. Now, we will show how constant dependency relation can be used for PlusCal-2 algorithms and how the partial-order reduction method in TLC model checker can reduce the state space explosion problem.

In this chapter, we will first present the extended TLC model checker with an implementation of a static partial-order reduction technique [Akhtar 2011] adapted from the method by Holzmann and Peled presented in [Holzmann 1994]. This will be followed by a description of the constant independence information. Then, we will discuss the results of our implementation for Leader election and concurrent sorting algorithm from SPIN distribution and finally, we will show the proof of correctness for the subset of actions that are selected at any point in the reduced search space using the conditions for ample sets.

5.1 Adapted partial-order reduction for TLC

TLC[Yu 1999] is a powerful model checker used to verify specifications written in TLA$^+$. It was designed for the verification of large and complicated systems such as communication networks and cache coherence protocols. The TLC model checker performs a breadth-first search to traverse the state graph. It starts by generating all the initial states and checks if they satisfy the invariant properties. If they satisfy the invariant properties, then it adds them to a FIFO queue, if not then it reports an error and stops any further execution. Then it launches threads which execute the process described below:

- pick a state at the front of the FIFO queue and generate all its successor states,
- for each successor state, check if it satisfies all the invariant properties and add it to the end of the FIFO queue,
- if some successor does not satisfy some invariant property, report an error and print the corresponding counter example.

Once the model checking process is complete, the temporal properties are verified over the entire state space. We extend TLC by an implementation of the partial-order reduction technique first proposed by Holzmann and Peled in [Holzmann 1994]. The original algorithm is meant for depth-first search algorithm whereas in our implementation we made some minor changes for adapting it to a breadth-first search algorithm. The other modification concerns the concept of processes, which is fundamental in [Holzmann 1994]. Processes are not present in TLA$^+$ where transitions are described by *actions* over a flat system model. However, the TLA$^+$ actions generated by the PlusCal-2 compiler all have a parameter *self* that represents the PlusCal-2 process executing the action. The *self* parameter is also necessary for defining the constant independence relation in the form of a matrix to perform partial-order reduction.

The pseudo-code of the model checking algorithm with partial-order reduction that we implemented is as follows:

```
1  main() {
2    curstate = Pick a state from FIFO queue
3    order enabled actions for curstate       /* reorder actions, as explained below */
4    for each action a in enabled actions of curstate {
5      InPath = true
6      for each succstate in successors of action a {
7        if succstate in not already seen {
8          if succstate violates any invariant {
9            print counter example and stop model checking process
10         }
11         add succstate to the list of already seen states
```

```
12          add succstate to the FIFO queue
13        }
14        else if succstate seen in current path {
15          InPath = false
16        }
17      }
18      if InPath and IsActionSafe(a) {
19        break from loop over enabled actions
20 } } }
```

This algorithm picks a state from the FIFO queue of TLC model checker and performs a critical step at line 3, to reorganize the list of actions. In the original technique, the processes were reordered on the basis of safety principles. The notion of processes slightly changes for TLA$^+$ by an action corresponding to each process and these actions are reordered on the basis of safety principles. The safety property for an action can be defined as follows: an action is safe if it is independent of all other currently enabled actions and if it is non-observable by the formulas under verification.

```
1 orderActions(curstate) {
2   for each action a in enabled actions of curstate {
3     isIndependent = action a independent of all other enabled actions
4     isNotObservable = check observability for each successor state of a at curstate
5     if isIndependent and isNotObservable {
6       mark action a as safe
7     }
8     else {
9       mark action a as unsafe
10  } }
11  reorganize the list of actions as safe:unsafe:disabled
12 }
```

In the above algorithm, the independence relation computed at line 3 between the actions is represented using the independence matrix described in the previous chapter. However, it is constructed using constant predicates as required by the Holzmann and Peled's method of partial-order reduction. The method to specify these relations is explained in section 5.2. The second condition of non-observability for an action to be safe is computed at line 4. For all the pairs of current and successor states, we check if some state component evaluated by the invariant is changed by the action. If it changes, then we mark that action to be observable as it affects the invariant properties.

Then, the actions are reordered in such a way that the safe actions are placed on the top of the list, then the actions that are unsafe and finally, the actions that are disabled. Enabledness of actions is determined by computing the successors of all actions at the current state; the results of this computation is reused when actually

producing the successor states, although not all of them have to be stored for further verification if the reduction is successful.

Then, in the model checking algorithm with partial-order reduction, each successor of an enabled action at the current state is examined at lines 7 and 14. If the successor state was not already visited during the search then it is added to the list of visited states and to the FIFO queue for further exploration. At line 14, if the successor state was already seen during the search then we check if it was seen in the current execution path to avoid the *ignoring* problem. By *ignoring* problem we mean that whenever we encounter a cycle in an execution path, then there is a possibility that we ignored a safe action along the path.

As TLC implements breadth-first search, it does not have a notion of stack that contains the current execution path. However, TLC provides a mechanism to trace back to the initial state. We use this mechanism to find out the existence of the successor state in the current path. Finally, at line 18 from the original partial-order reduction algorithm, we have the reduction proviso that must hold for a successful reduction process. The condition of reduction proviso guarantees that there is no cycle in the current sequence and the action selected for further exploration is independent of rest of actions enabled at that state, and non-observable in the state graph. As the original static partial-order reduction algorithm guarantees to preserve safety and liveness properties, our implementation in TLC also preserves those properties.

5.2 Defining constant independence relation

Holzmann and Peled's partial-order reduction method requires the constant independence relation between actions to guarantee the global independence of an action. This independence relation can be computed for each pair of actions in a triangular matrix and added to the TLA$^+$ specifications in the form of the following TLA$^+$ definition:

$$
\begin{array}{l}
IndepMatrix(A, B) \triangleq \\
\quad \text{CASE} \\
\qquad\quad A = name_1 \wedge B = name_1 \;\; \rightarrow \;\; P_{11} \\
\qquad \Box \quad A = name_1 \wedge B = name_2 \;\; \rightarrow \;\; P_{12} \\
\qquad\quad \vdots \\
\qquad \Box \quad A = name_n \wedge B = name_n \;\; \rightarrow \;\; P_{nn}
\end{array}
$$

where A and B are the action names that TLC uses to retrieve the independence value for the pair of actions. $name_1, \ldots, name_n$ are the names of the TLA$^+$ actions that are generated for the atomic blocks of the PlusCal-2 algorithm in intermediate representation. $P_{11}, P_{12}, ..P_{nn}$ define the independence value that can either be TRUE or FALSE in the case of constant independence relation.

This relation for Leader election algorithm shown in appendix 1.2 is defined as follows:

$$
\begin{array}{l}
IndepMatrix(A, B) \triangleq \\
\quad \text{CASE} \\
\qquad\quad A = \text{"}start\text{"} \wedge B = \text{"}start\text{"} \quad \rightarrow \quad \text{TRUE} \\
\qquad \Box \quad A = \text{"}start\text{"} \wedge B = \text{"}forever\text{"} \quad \rightarrow \quad \text{TRUE} \\
\qquad \Box \quad A = \text{"}forever\text{"} \wedge B = \text{"}forever\text{"} \quad \rightarrow \quad \text{TRUE} \\
\qquad \Box \quad \text{OTHER} \quad \rightarrow \quad \text{FALSE}
\end{array}
$$

It can be written in the compact form as below:

$$IndepMatrix(A, B) \triangleq \text{TRUE}$$

5.3 Examples and Results

We implemented two algorithms that are contained in the distribution of the Spin model checker [Holzmann 2003] and are used to check the performance of partial-order reduction within Spin. The two algorithms are the leader election algorithm and a concurrent sorting algorithm that we used previously in this thesis as well.

5.3.1 Leader election algorithm

The Leader election algorithm that we implemented in PlusCal-2 for generating TLA$^+$ specifications is the algorithm for electing a leader in a unidirectional ring due to Dolev, Klawe, and Rodeh [Dolev 1982], shown in appendix 1.2.

The PlusCal-2 compiler generates two actions for this algorithm, which correspond to the blocks of actions labeled start (initialization) and forever (one pass through the loop). It should be noted that the second action is of much coarser granularity than that of a typical transition in Spin, but if we take a closer look at the modifications performed by this action, they remain local to the process. Figure 5.1 shows the numbers of states generated for different numbers of processes with and without partial-order reduction.

As the number of processes increases, the number of states generated by TLC increases exponentially, resulting in state space explosion. This makes it impractical to verify larger instances of this algorithm. In contrast, the number of states increases only linearly when partial-order reduction is used. The running time of TLC, shown in figure 5.1 is reduced accordingly, varying between 0.09 seconds for 4 processes to 0.17 seconds for 12 processes (on a standard laptop running Windows) compared to 0.2 and 349 seconds, respectively, without partial-order reduction. These results clearly show that our method is effective for this algorithm.

As the partial-order reduction method by Holzmann preserves both the safety and the liveness properties of concurrent systems, we also verified a safety and liveness property for the Leader election algorithm. One of the safety properties for Leader election algorithm is that only one process should be selected as a leader, this can be written formally in PlusCal-2 as follows:

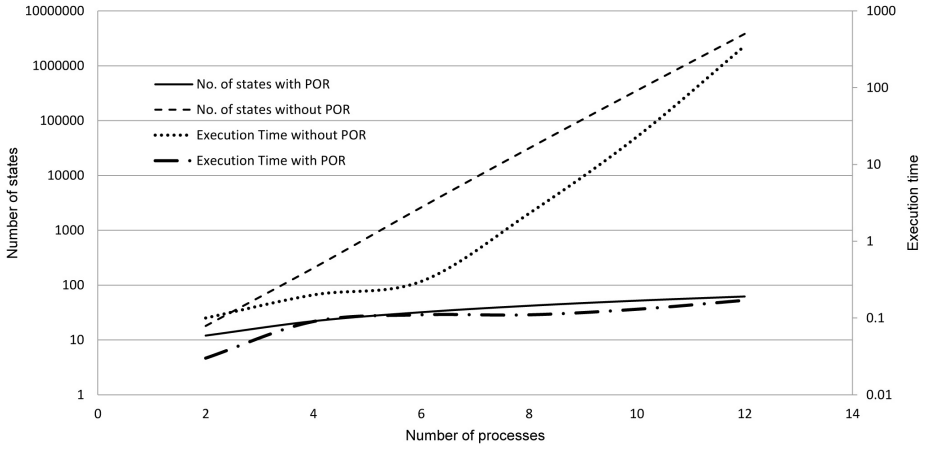

Figure 5.1: Results for the DKR algorithm with and without partial-order reduction in TLC.

$$\textbf{invariant } \forall p \in Node : Node[p].winner \Rightarrow \\ (\forall q \in Node \setminus \{p\} : \neg Node[q].winner)$$

winner is a process local flag variable in Leader election algorithm that identifies the leader process. To guarantee that only one process is selected as a leader, then the other processes should have their flags set to FALSE. For verification of a liveness property for Leader election algorithm, we state that one of the processes that are competing in the selection procedure, will become the leader at the end of the procedure. This property can be written formally in PlusCal-2 algorithm as follows:

$$\textbf{temporal } \exists p \in Node : \Diamond Node[p].winner$$

winner is a local variable of process *Node* and it is set to TRUE if a process is the leader. For the verification of the above property, we must also assume fairness conditions for the process *Node* that can easily be done by simply adding the reserved word **fair** before the name of the process as shown below:

$$\textbf{fair process } Node[N]$$

5.3.2 Sort algorithm

The second algorithm, again taken from the Spin distribution, concurrently sorts N random numbers. It is also known to be an example in which partial-order reduction works well in Spin.

The main objective of the sort algorithm is to concurrently sort the N randomly generated numbers. The algorithm contains definitions for three different types of processes: left, middle and right. The process left generates the random numbers

using the definition RANDOM and passes them on to the network. The process middle has N instances and each of the instances compare the new number with the one they already have. If its found to be larger, then it passes it on to the next process through the network otherwise it keeps the new number and passes the one it already had. The process right will always receive the number that is larger than all the other numbers held by the instances of process middle.

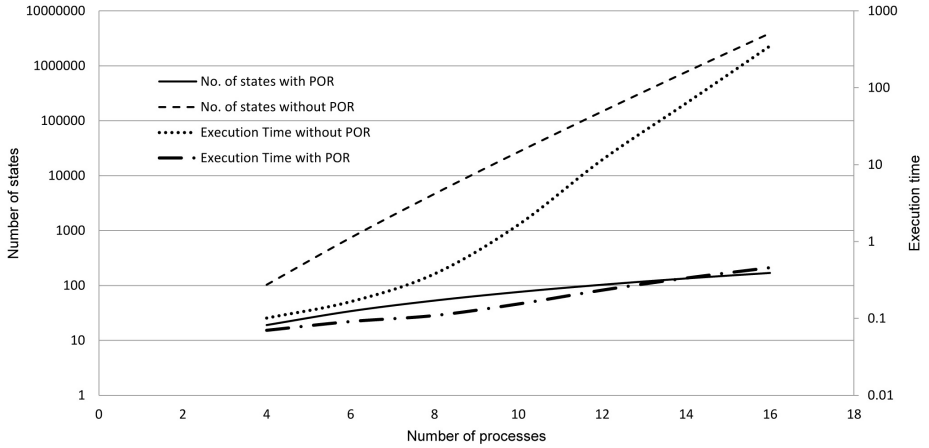

Figure 5.2: Model checking results for the sorting algorithm with and without partial-order reduction in TLC.

The full search by the model checker for this algorithm becomes exponential and it can be reduced if we manage to define the independence between the blocks of statements that yield transitions. We passed the algorithm to the PlusCal-2 compiler that successfully generated the TLA$^+$ specification. Then, we added the constant independence relation for the actions in the specifications that contains no dependence between the actions as they never interfere in each other's execution. The variables updated by all the processes are the local variables for those processes. The constant independence relation is represented using the independence matrix in a compact form as follows:

$$IndepMatrix(A, B) \triangleq \text{TRUE}$$

Figure 5.2 shows the model checking results for the TLA$^+$ specification of the sorting algorithm. We model checked the specifications for 8 to 16 processes on a standard laptop running windows and found that partial-order reduction works efficiently as shown in the figure 5.2. On the other hand, without independence information the model checker results in an exponential increase in the number of states it generates.

5.4 Proof of correctness

In this section, we prove that the subset of transitions selected at any point in the reduced search space using our adapted reduction algorithm is an ample set. The exploration of only the transitions in ample set at each state during reduced search is sufficient for the verification of stuttering invariant LTL properties [Godefroid 1994, Clarke 1999]. Since TLA is a stuttering invariant subset of LTL, it is therefore enough to show that the set of actions explored at every state by the reduced model checking algorithm is an ample set. The conditions for the ample sets are discussed in detail in chapter 2 and below we present the proof of correctness for our adaptation.

C0: ample(s) $= \emptyset$ if and only if enabled(s) $= \emptyset$

In our implementation, we reorder the set of enabled transitions before their expansion in such a way that the safe transitions are executed first which form the subset of transitions selected at that point. If we are unable to find one such transition, then no reduction is performed and all of the enabled transitions are selected for exploration. Thus, the subset of transitions is only empty if there are no enabled transitions.

C1: Along every path in the full state graph that starts at state s, the following condition holds: a transition that is dependent on a transition in ample(s) cannot be executed without a transition in ample(s) occurring first.

To prove that the condition C1 holds, let s be some state reached by the reduced model checking algorithm and denote by A the set of actions explored by the reduced model checking algorithm at state s. Assume that we have an execution sequence $\sigma = s \xrightarrow{t_0} s_0 \xrightarrow{t_1} s_1 \ldots$ in the full state graph and that action t_i in the execution sequence σ is dependent on an action $t \in A$. We have to show that $t_k \in A$ for some $k \leq i$. There are two cases to consider.

Case 1: $i = 0$. In this case, t_i is the first action in the execution sequence σ. Since t_0 and t are dependent, t is not safe. The reduced algorithm explores some unsafe action at state s only if $A = \mathit{enabled}(s)$, and since t_0 appears as the first action in σ, it is enabled at s. Therefore we have $t_0 \in A$, and the assertion is proved.

Case 2: $i > 0$. Assume that $t_k \notin A$ for all $k < i$. In particular, $t_0 \notin A$, and therefore A must be a set of safe actions. Since the actions in A and t_0 are enabled and $t_0 \notin A$, it follows that $\mathit{Proc}(a) \neq \mathit{Proc}(t_0)$ for all $a \in A$: PlusCal-2 processes are sequential and only one statement of any process can be enabled at any state.

Now the definition of safe actions implies that all actions in A are independent of t_0 and are therefore enabled at state s_0. Continuing inductively, using the assumptions that all actions in A are safe and that $t_k \in \mathit{enabled}(s_{k-1}) \setminus A$ for all $k < i$,

we find that all actions in A (and in particular t) are enabled at state s_{i-1}, as is action t_i.

If $Proc(t_i) = Proc(t)$ then by the same argument as before we must have $t_i = t$, hence $t_i \in A$ and the assertion is proved. Otherwise, since all actions in A are safe, t and t_i must be independent, and a contradiction is reached.

C2: If state s is not fully expanded, then every $\alpha \in$ ample(s) is invisible.

The condition C2 refers to the non-observability or invisibility of the transitions selected in the subset. In our adaptation, whenever we succeed to find a subset T_{sub} of safe transitions then all the transitions $\alpha \in T_{sub}$ are invisible. Thus, the condition C2 holds for our adaptation.

C3: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in ample(s) for any state s on the cycle.

In our implementation of partial-order reduction for TLC, the condition at line 18 guarantees that if a transition leads to a state that was already visited in the path then it forces the exploration of all the transitions at that point. Therefore, the condition C3 holds.

5.5 Summary

In this chapter, we presented the extended TLC model checker that includes an implementation of the partial-order reduction technique proposed by Holzmann and Peled in [Holzmann 1994]. TLC model checker supports the model checking of algorithms described in TLA$^+$ specification language. It uses breadth first search strategy to explore all the possible states that does not have a notion of stack to store the current transition sequence. Whereas, the reduction methods with conditional independence essentially require depth-first methods for state space exploration. To use conditional independence relation in breadth first search, one would have to store additional information about the states that will increase the space complexity of the model checker and time complexity to retrieve the states from the disk. Thus, breadth first search only supports partial-order reduction with constant dependency.

We also described the presentation of constant independence information. This information is added to the TLA$^+$ specifications for TLC. Then, we presented the results of our implementation for Leader election and concurrent sorting algorithm from SPIN distribution and the proof of correctness for the subset of actions that are selected at any point in the reduced search space using the conditions for ample sets.

Chapter 6

Conclusions and future work

Contents

6.1 Conclusions

Designing sound algorithms for concurrent and distributed systems [Lynch 1996] is subtle and challenging. It is particularly complicated to foresee and reproduce the high number of potential interleavings of individual component actions, which may cause deadlocks and race conditions. Moreover, it is often quite difficult to precisely state the assumptions and guarantees that determine whether an algorithm is correct. Indeed, several algorithms proposed in the literature have been found to be erroneous having different interpretations regarding the precise objectives and hypotheses of the algorithms. Formal verification is therefore crucial in concurrent and distributed computing.

Model checking [Clarke 1999] is a popular verification technique for concurrent and distributed algorithms. It provides tools for deciding automatically whether properties (typically expressed in temporal logic) are verified for finite instances of systems or algorithms, described in a formal modeling language. Its main limitation is the well-known state explosion problem, which can be mitigated by verifying algorithms at a high level of abstraction. For example, Lamport's specification language TLA$^+$ [Lamport 2002], which is supported by the model checker TLC [Yu 1999], encourages designers to express algorithms in terms of abstract mathematical concepts such as sets and functions. Although TLA$^+$ is very expressive and can describe complicated algorithms quite concisely, algorithm designers often find it difficult to write formulas, and would rather prefer a notation closer to (pseudo-)code, traditionally used to describe algorithms.

Recently, Lamport introduced the PlusCal algorithm language [Lamport 2006b]. While retaining the high level of abstraction of TLA$^+$ expressions, it provides familiar constructs of imperative programming languages for describing algorithms, such as processes, assignments, and control flow. The PlusCal compiler generates a TLA$^+$ specification, which is then verified using TLC. PlusCal is a high-level and powerful modeling language for algorithms, featuring mathematical abstractions,

non-determinism, and user-specified grain of atomicity; it emphasizes the analysis, not the efficient execution of algorithms.

Unfortunately, Lamport's PlusCal language and compiler have some significant limitations. In this thesis we presented a new version of PlusCal called PlusCal-2, which we have developed with the aim of overcoming these limitations and of providing a modeling language that is natural to use, while retaining a precise semantics, and paving the way to more efficient verification.

We briefly discuss how our implementation of PlusCal-2 overcomes the limitations of the original PlusCal language that we have identified in section 1.2.

Self-contained models. All information about an algorithm is expressed in the PlusCal-2 model, relieving the user of modifying the generated TLA$^+$ model. The user of our language does not need to read or understand the resulting TLA$^+$ model. Of course, an understanding of TLA$^+$ expressions is necessary, as these represent the data manipulated by PlusCal-2 algorithms.

In particular, correctness properties are stated within the PlusCal-2 model, in terms of the entities it contains, rather than in terms of the generated TLA$^+$ code. Similarly, fairness annotations are attached to PlusCal-2 processes or statements. The PlusCal-2 model also identifies the finite instance of the algorithm that is being model checked.

Scoped declarations and nested processes. Variable, procedure, and operator declarations are properly scoped, avoiding potential errors by inadvertently accessing the variables of a different process. Combined with the possibility to nest processes, this makes the communication structure of algorithms much more transparent.

Just as the original PlusCal language, PlusCal-2 does not contain primitives for message passing between processes. While we considered adding such primitives, we found that distributed algorithms use many different forms of message passing (synchronous or not, lossy, duplicating, preserving FIFO order, ...), and that these are better defined in a standard library of procedures.

User-defined atomicity. PlusCal-2 retains the basic idea of specifying atomicity for labels. We managed to lift some of the restrictions on label placement that were present in the original PlusCal language, and the compiler will add labels when they are required. The user can now enforce atomicity of code blocks containing labels using the new **atomic** statement of PlusCal-2.

Added flexibility. The **for** statement of PlusCal-2 greatly simplifies iteration over sets. For example, multicast communication is easily simulated by sending a message to each intended recipient in a **for** loop, within an atomic step.

We also managed to overcome the restriction of assigning each variable at most once within an atomic block by silently introducing LET-bound constants for intermediate values of the variable.

While our PlusCal-2 variant retains most of the "look and feel" of the original PlusCal language, it does not guarantee backward compatibility. For example, programs that modify variables that are not currently in scope will be rejected by the new PlusCal-2 compiler.

As mentioned earlier, the main limitation of Model checking [Clarke 1999] process is the well-known state space explosion problem, which can be mitigated by verifying algorithms at a high level of abstraction. We developed several examples from the literature in PlusCal-2. The compiler successfully produced corresponding TLA$^+$ models, and we could use the TLC model checker to verify the algorithms without any further modification. However, the well-known state space explosion problem limits the sizes of instances that can be verified effectively. Since TLA$^+$ and PlusCal-2 are mainly intended for verifying asynchronous distributed algorithms, thus we turned to partial-order reduction methods, which are known to be the most effective reduction techniques in this context. The main idea of partial-order reduction is to restrict the state-space exploration such that redundant interleavings of transitions are avoided, hence preserving soundness of the verification.

Partial-order reduction methods strongly rely on the dependency relation between the transitions in the system. This relation can either be constant or conditional dependency relation. Constant dependency relation is computed statically for global independence or dependence of transitions whereas the conditional dependency relation is composed of predicates corresponding to each pair of transitions. These predicates are evaluated during model checking process for any given state. In this thesis, we studied the constant dependency relation for partial-order reduction method by Holzmann and Peled [Holzmann 1994] and the dynamic partial-order reduction method by Cormac Flanagan and Patrice Godefroid presented in [Flanagan 2005] for conditional dependency relation.

To explore the conditional dependency relation, in this thesis, we presented an extended PlusCal-2 compiler for extracting the conditional independence predicates from PlusCal-2 algorithms. These conditional independence predicates can be further used in partial-order reduction methods that support conditional independence relations like the dynamic partial-order reduction method by Cormac Flanagan and Patrice Godefroid presented in [Flanagan 2005]. We also proposed an adaptation of this dynamic partial-order reduction algorithm for a variant of TLC model checker that could use the independence predicates produced by PlusCal-2 compiler.

Conditional independence is useful when one cannot determine the dependency relation statically for the pairs of transitions. In the concurrent and distributed systems, where the processes work independently, the computation of conditional independence predicates becomes unnecessary to have such a fine-grained analysis. For such systems, the constant dependency relation becomes useful and more efficient. Thus, in this thesis, we also presented adaptation of partial-order reduction technique using constant dependency relation proposed by Holzmann and Peled

in [Holzmann 1994] with an implementation in TLC model checker.

TLC model checker supports the model checking of algorithms described in TLA$^+$ specification language. It uses breadth first search strategy to explore all the possible states but it also provides a notion to retrieve the current transition sequence that is required to check reduction proviso condition or to address the *ignoring* problem. We presented the adapted partial-order algorithm for TLC followed by the presentation of constant independence information. This information is added to the TLA$^+$ specifications for TLC. We also presented the results of our implementation and the proof of correctness for the subset of actions that are selected at any point in the reduced search space using the conditions for ample sets.

6.2 Future work

The aim of our thesis was to evolve the existing approaches for modeling and verification of distributed and concurrent systems and provide the algorithm designers with a platform where they can easily model an algorithm and verify it using some formal verification tool. Distributed and concurrent systems are difficult to model and verify. The new PlusCal-2 language will provide an interface for the algorithm designers to verify their algorithms before implementation. For research purposes, it will be interesting to study and model distributed and concurrent algorithms in PlusCal-2 and then model check them using the TLC model checker. PlusCal-2 will provide all the required constructs to represent the specifications of any system. With the new advancements in distributed and concurrent systems that include more complexity, it becomes necessary to introduce languages that can represent them easily.

The implementation of the variant of TLC model checker using depth first search along with the adapted dynamic partial-order reduction algorithm would open various lines for future research. This implementation would require detailed research of the existing TLC model checker. An interesting line of future research is to study the results for various complex distributed algorithms using dynamic partial-order reduction method in TLC with the help of conditional independence predicates produced by PlusCal-2 compiler. It will be a useful contribution in research to develop the proposed platform that is TLC with depth first search strategy that supports partial-order reduction with conditional independence. Another line of future research would be to study the possibilities for identifying "domain-specific" independence predicates, beyond what we have so far implemented for operations on FIFO channels.

Bibliography

[Akhtar 2010] Sabina Akhtar, Stephan Merz and Martin Quinson. *A High-Level Language for Modeling Algorithms and Their Properties*. In Jim Davies, Leila Silva and Adenilso da Silva Simão, editeurs, 13th Brazilian Symp. on Formal Methods (SBMF 2010), volume 6527 of *Lecture Notes in Computer Science*, pages 49–63, Natal, Brazil, 2010. Springer. (Cited on page 36.)

[Akhtar 2011] Sabina Akhtar and Stephan Merz. *Partial-Order Reduction for Verifying* PlusCal-2 *Algorithms*. In 11th International Workshop on Automated Verification of Critical Systems, Newcastle, England, 2011. (Cited on page 91.)

[Allen 1987] Randy Allen and Ken Kennedy. *Automatic Translation of Fortran Programs to Vector Form*. ACM Trans. Program. Lang. Syst., vol. 9, no. 4, pages 491–542, 1987. (Cited on page 30.)

[Anderson 2001] Ross J. Anderson. Security engineering: A guide to building dependable distributed systems. John Wiley & Sons, Inc., New York, NY, USA, 1st édition, 2001. (Cited on pages 17 and 18.)

[Andrews 2000] Gregory R. Andrews. Foundations of multithreaded, parallel, and distributed programming. Addison-Wesley, 2000. (Cited on pages 17 and 18.)

[Baier 2008] Christel Baier and Joost-Pieter Katoen. Principles of model checking. MIT Press, 2008. (Cited on pages 1 and 9.)

[Behrmann 2004] Gerd Behrmann, Alexandre David and Kim Guldstrand Larsen. *A Tutorial on Uppaal*. In SFM, pages 200–236, 2004. (Cited on page 25.)

[Clarke 1981] Edmund M. Clarke and E. Allen Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*. In Logic of Programs, pages 52–71, 1981. Reprint in 2008. (Cited on page 23.)

[Clarke 1986] Edmund M. Clarke, E. Allen Emerson and A. Prasad Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Trans. Program. Lang. Syst., vol. 8, no. 2, pages 244–263, 1986. (Cited on page 23.)

[Clarke 1996a] Edmund M. Clarke, Orna Grumberg and David E. Long. *Model checking*. In NATO ASI DPD, pages 305–349, 1996. (Cited on pages 1, 9, 23 and 25.)

[Clarke 1996b] Edmund M. Clarke, Somesh Jha, Reinhard Enders and Thomas Filkorn. *Exploiting Symmetry in Temporal Logic Model Checking*. Formal Methods in System Design, vol. 9, no. 1/2, pages 77–104, 1996. (Cited on page 25.)

[Clarke 1999] Edmund M. Clarke, Orna Grumberg and Doron Peled. Model checking. MIT Press, Cambridge, Mass., 1999. (Cited on pages 25, 26, 27, 67, 98, 101 and 103.)

[Dolev 1982] Danny Dolev, Maria M. Klawe and Michael Rodeh. *An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle.* J. Algorithms, vol. 3, no. 3, pages 245–260, 1982. (Cited on pages 5, 13, 38, 95 and 118.)

[Emerson 1996] E. Allen Emerson and A. Prasad Sistla. *Symmetry and Model Checking.* Formal Methods in System Design, vol. 9, no. 1/2, pages 105–131, 1996. (Cited on page 25.)

[Flanagan 2005] Cormac Flanagan and Patrice Godefroid. *Dynamic partial-order reduction for model checking software.* In Jens Palsberg and Martín Abadi, editeurs, 32nd ACM Symp. Principles of Programming Languages (POPL 2005), pages 110–121, Long Beach, CA, U.S.A., 2005. ACM. (Cited on pages 33, 68, 82, 83, 84, 85, 86, 88 and 103.)

[Godefroid 1991] Patrice Godefroid. *Using partial orders to improve automatic verification methods.* In Edmund Clarke and Robert Kurshan, editeurs, Computer-Aided Verification, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer Berlin / Heidelberg, 1991. (Cited on page 25.)

[Godefroid 1993] Patrice Godefroid and Didier Pirottin. *Refining Dependencies Improves Partial-Order Verification Methods (Extended Abstract).* In CAV, pages 438–449, 1993. (Cited on page 32.)

[Godefroid 1994] P. Godefroid and P. Wolper. *A Partial Approach to Model Checking.* Information and Computation, vol. 110, no. 2, pages 305–326, 1994. (Cited on pages 32 and 98.)

[Godefroid 1996] Patrice Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem, volume 1032 of *Lecture Notes in Computer Science.* Springer, 1996. (Cited on pages 28, 29 and 30.)

[Gurevich 2005] Yuri Gurevich, Benjamin Rossman and Wolfram Schulte. *Semantic essence of AsmL.* Theor. Comput. Sci., vol. 343, no. 3, pages 370–412, 2005. (Cited on page 37.)

[Holzmann 1994] Gerard Holzmann and Doron Peled. *An Improvement in Formal Verification.* In IFIP WG 6.1 Conference on Formal Description Techniques, pages 197–214, Bern, Switzerland, 1994. Chapman & Hall. (Cited on pages 27, 28, 33, 91, 92, 99, 103 and 104.)

[Holzmann 1997] Gerard J. Holzmann. *The Model Checker SPIN.* IEEE Trans. Softw. Eng., vol. 23, pages 279–295, May 1997. (Cited on page 24.)

[Holzmann 2003] Gerard Holzmann. Spin model checker, the: primer and reference manual. Addison-Wesley Professional, first édition, 2003. (Cited on pages 24, 37 and 95.)

[Ip 1993] C. Norris Ip and David L. Dill. *Better Verification Through Symmetry*. In David Agnew, Luc J. M. Claesen and Raul Camposano, editeurs, CHDL, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993. (Cited on page 25.)

[Katz 1988] Shmuel Katz and Doron Peled. *An efficient verification method for parallel and distributed programs*. In REX Workshop, pages 489–507, 1988. (Cited on page 30.)

[Katz 1992] Shmuel Katz and Doron Peled. *Defining Conditional Independence Using Collapses*. Theor. Comput. Sci., vol. 101, no. 2, pages 337–359, 1992. (Cited on page 30.)

[Killian 2007] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala and Amin Vahdat. *Mace: Language Support for Building Distributed Systems*. In PLDI, pages 179–188, 2007. (Cited on page 37.)

[Kurshan 1998] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled and Hüsnü Yenigün. *Static Partial Order Reduction*. In TACAS, pages 345–357, 1998. (Cited on page 33.)

[Lamport 1994] Leslie Lamport. *The temporal logic of actions*. ACM Trans. Program. Lang. Syst., vol. 16, pages 872–923, May 1994. (Cited on pages 2 and 10.)

[Lamport 2002] Leslie Lamport. Specifying systems, the TLA$^+$ language and tools for hardware and software engineers. Addison-Wesley, 2002. (Cited on pages 2, 10, 21, 36 and 101.)

[Lamport 2006a] Leslie Lamport. *The +CAL Algorithm Language*. In FORTE, page 23, 2006. (Cited on pages 2 and 10.)

[Lamport 2006b] Leslie Lamport. *Checking a Multithreaded Algorithm with +CAL*. In Shlomi Dolev, editeur, 20th Intl. Symp. Distributed Computing (DISC 2006), volume 4167 of *LNCS*, pages 151–163, Stockholm, Sweden, 2006. Springer. (Cited on pages 3, 11, 36 and 101.)

[Lamport 2007] Leslie Lamport. *A +CAL User's Manual*. `http://research.microsoft.com/en-us/um/people/lamport/tla/pluscal.html`, 2007. (Cited on page 19.)

[Lamport 2009] Leslie Lamport. *The PlusCal Algorithm Language*. In Martin Leucker and Carroll Morgan, editeurs, ICTAC, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009. (Cited on pages 2 and 10.)

[Liu 2011] Yanhong A. Liu, Bo Lin and Scott D. Stoller. *Programming and Optimizing Distributed Algorithms: An Overview.* In Proc. 8th International Conference & Expo on Emerging Technologies for a Smarter World (CEWIT 2011). IEEE Press, November 2011. (Cited on page 37.)

[Lynch 1996] Nancy A. Lynch. Distributed algorithms. Morgan Kaufmann, 1996. (Cited on pages 1, 9, 17, 35 and 101.)

[McMillan 1992] Kenneth L. McMillan. *Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits.* In CAV, pages 164–177, 1992. (Cited on page 25.)

[McMillan 1993] Kenneth L. McMillan. Symbolic model checking. Kluwer, 1993. (Cited on page 37.)

[Naimi 1996] Mohamed Naimi, Michel Trehel and André Arnold. *A Log(N) Distributed Mutual Exclusion Algorithm Based on Path Reversal.* J. Parallel Distrib. Comput., vol. 34, no. 1, pages 1–13, 1996. (Cited on pages 3, 11 and 111.)

[Peled 1990] Doron Peled and Amir Pnueli. *Proving Partial Order Liveness Properties.* In ICALP, pages 553–571, 1990. (Cited on page 30.)

[Peled 1993] Doron Peled. *All from One, One for All: on Model Checking Using Representatives.* In CAV, pages 409–423, 1993. (Cited on pages 25 and 27.)

[Peled 1996a] Doron Peled. *Combining Partial Order Reductions with On-the-Fly Model-Checking.* Formal Methods in System Design, vol. 8, no. 1, pages 39–64, 1996. (Cited on pages 32 and 33.)

[Peled 1996b] Doron Peled. *Partial Order Reduction: Model-Checking Using Representatives.* In MFCS, pages 93–112, 1996. (Cited on page 31.)

[Peled 1998] Doron Peled. *Ten Years of Partial Order Reduction.* In CAV, pages 17–28, 1998. (Cited on page 31.)

[Peterson 1981] Gary L. Peterson. *Myths About the Mutual Exclusion Problem.* Inf. Process. Lett., vol. 12, no. 3, pages 115–116, 1981. (Cited on page 19.)

[Queille 1981] Jean-Pierre Queille and Joseph Sifakis. *Iterative Methods for the Analysis of Petri Nets.* In Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets, pages 161–167, 1981. Reprint in 2008. (Cited on page 23.)

[Rabin 1981] M O Rabin. *Fingerprinting by random polynomials.* Technical Report TR1581 Center for Research in, no. TR-15-81, pages 15–18, 1981. (Cited on page 24.)

[Valmari 1989] Antti Valmari. *Stubborn sets for reduced state space generation.* In Applications and Theory of Petri Nets, pages 491–515, 1989. (Cited on page 30.)

[Valmari 1990] A. Valmari. *A stubborn attack on state explosion.* In 2nd International Workshop on Computer Aided Verification, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer Verlag. (Cited on pages 27, 30 and 32.)

[Valmari 1996] Antti Valmari. *The State Explosion Problem.* In Petri Nets, pages 429–528, 1996. (Cited on pages 25 and 32.)

[Yabandeh 2009] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic and Viktor Kuncak. *CrystalBall: predicting and preventing inconsistencies in deployed distributed systems.* In Proceedings of the 6th USENIX symposium on Networked systems design and implementation, NSDI'09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association. (Cited on page 37.)

[Yang 2007] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan and Robert M. Kirby. *Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software.* In SPIN, pages 58–75, 2007. (Cited on page 33.)

[Yang 2008] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan and Robert M. Kirby. *Efficient Stateful Dynamic Partial Order Reduction.* In SPIN, pages 288–305, 2008. (Cited on page 33.)

[Yu 1999] Yuan Yu, Panagiotis Manolios and Leslie Lamport. *Model checking TLA+ Specifications.* In L. Pierre and T. Kropf, editeurs, Correct Hardware Design and Verification Methods (CHARME'99), volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer. (Cited on pages 2, 10, 24, 36, 92 and 101.)

Appendix A

Examples

1.1 Naimi-Trehel algorithm

Naimi-Trehel algorithm was proposed by Naimi and Trehel in [Naimi 1996]. It is a distributed algorithm for mutual exclusion that maintains two distributed data structures: a list of processes that are waiting for access to the critical section, and a tree of process whose root is the process at the end of the waiting queue (or the process who last accessed its critical section if the queue is empty). Below we have a model of this algorithm in the PlusCal language along with it's TLA$^+$ specifications.

1.1.1 PlusCal model

```
1  --algorithm NaimiTrehel
2  variables procQueue = [to ∈ Peers ↦ ⟨⟩] ;
3
4
5  macro send(to, msg)
6  begin
7      procQueue[to]:= Append(procQueue[to], msg);
8  end macro ;
9
10
11
12 macro recv(got)
13 begin
14     when procQueue[self] ≠ ⟨⟩ ;
15     got := Head(procQueue[self]);
16     procQueue[self] := Tail(procQueue[self]);
17 end macro ;
18
19 (* Invoke the critical section *)
20 procedure request_cs()
21 begin
22 rqcs:
23     reqCS := TRUE ;
24     if father ≠ nil then
25         send(father, [snd ↦ self, kind ↦ "request"]);
26         father := nil ;
27     end if ;
28 end_rqcs:
29     return ;
30 end procedure ;
```

```
31 procedure handle_messages()
32 variables rcvd = [snd ↦ 0 , kind ↦ ""]
33 begin
34 recv_msg:
35       if procQueue[self] ≠ ⟨⟩ then
36            recv(rcvd) ;
37            if rcvd.kind = "request" then
38                 if father = nil then
39                      if reqCS = TRUE then
40                           next := rcvd.snd ;
41                      else
42                           tokPresent := FALSE ;
43 lb_call1:                send(rcvd.snd, [snd ↦ self, kind ↦ "give_token"])
44                      end if ;
45                 else
46 lb_call2:           send (father, rcvd);
47                 end if ;
48 lb_ret:       father := rcvd.snd ;
49            elsif rcvd.kind = "give_token" then
50                 tokPresent := TRUE ;
51            end if ;
52       end if ;
53 end_recv_msg:
54       return ;
55 end procedure ;
56
57
58
59 (* Release the CS and send the token to next process *)
60 procedure release_cs()
61 begin
62 rlcs:
63        reqCS := FALSE ;
64        if next ≠ nil then
65          send (next, [snd ↦ self, kind ↦ "give_token"]) ;
66          tokPresent := FALSE ;
67          next := nil ;
68        end if ;
69 end_rlcs:
70          return ;
71 end procedure ;
```

```
72  process Site ∈ Peers
73  variables   father = 1,
74              next = nil,
75              reqCS = FALSE,
76              tokPresent = FALSE ;
77  begin
78      (* The process behavior *)
79  init:
80      tokPresent := father = self ;
81      if father = self then
82          father := nil ;
83      end if ;
84  ncs :
85      while TRUE do
86          either call request_cs() ;
87  continue1:
88                  if tokPresent = TRUE then
89                      goto cs ;
90                  end if ;
91          or
92                  call handle_messages()  ;
93  continue2:
94                  goto ncs ;
95          end either ;
96  try:
97          while tokPresent = FALSE do
98              call handle_messages() ;
99          end while ;
100 cs:       skip ;
101 exit:     call release_cs() ;
102     end while ;
103 end process ;
104 end algorithm
```

1.1.2 TLA$^+$ specifications for Naimi-Trehel algorithm

MODULE NaimiTrehel

```
EXTENDS Naturals, TLC, Sequences, FiniteSets


CONSTANTS Peers, any, nil
VARIABLES procQueue, pc, stack, rcvd, father, next, reqCS, tokPresent
vars ≜ ⟨ procQueue, pc, stack, rcvd, father, next, reqCS, tokPresent ⟩



ProcSet ≜ (Peers)

Init ≜ (* Global variables *)
      ∧ procQueue = [to ∈ Peers ↦ ⟨⟩]
      (* Procedure handle_messages *)
      ∧ rcvd = [ self ∈ ProcSet ↦ [snd ↦ 0 , kind ↦ ""]]
      (* Process Site *)
      ∧ father = [self ∈ Peers ↦ 1]
      ∧ next = [self ∈ Peers ↦ nil]
      ∧ reqCS = [self ∈ Peers ↦ FALSE]
      ∧ tokPresent = [self ∈ Peers ↦ FALSE]
      ∧ stack = [self ∈ ProcSet ↦ ⟨⟩]
      ∧ pc = [self ∈ ProcSet ↦ CASE self ∈ Peers → "init"]

rqcs(self) ≜
          ∧ pc[self] = "rqcs"
          ∧ reqCS' = [reqCS EXCEPT ![self] = TRUE]
          ∧ IF father[self] ≠ nil
              THEN ∧ procQueue' = [procQueue EXCEPT ![father[self]] = Append(
                                      procQueue[father[self]], ([snd ↦ self, kind ↦ "request"]))]
                   ∧ father' = [father EXCEPT ![self] = nil]
              ELSE ∧ TRUE
                   ∧ UNCHANGED ⟨ procQueue, father ⟩
          ∧ pc' = [pc EXCEPT ![self] = "end_rqcs"]
          ∧ UNCHANGED ⟨ stack, rcvd, next, tokPresent ⟩

end_rqcs(self) ≜
             ∧ pc[self] = "end_rqcs"
             ∧ pc' = [pc EXCEPT ![self] = Head(stack[self]).pc]
             ∧ stack' = [stack EXCEPT ![self] = Tail(stack[self])]
             ∧ UNCHANGED ⟨ procQueue, rcvd, father, next, reqCS, tokPresent ⟩

request_cs(self) ≜ rqcs(self) ∨ end_rqcs(self)

lb_ret(self) ≜
            ∧ pc[self] = "lb_ret"
            ∧ father' = [father EXCEPT ![self] = rcvd[self].snd]
            ∧ pc' = [pc EXCEPT ![self] = "end_recv_msg"]
            ∧ UNCHANGED ⟨ procQueue, stack, rcvd, next, reqCS, tokPresent ⟩

lb_call2(self) ≜
             ∧ pc[self] = "lb_call2"
             ∧ procQueue' = [procQueue EXCEPT ![father[self]] =
                                 Append(procQueue[father[self]], rcvd[self])]
             ∧ pc' = [pc EXCEPT ![self] = "lb_ret"]
             ∧ UNCHANGED ⟨ stack, rcvd, father, next, reqCS, tokPresent ⟩

```

```
recv_msg(self) ≜
        ∧ pc[self] = "recv_msg"
        ∧ IF procQueue[self] # ⟨⟩
            THEN ∧ procQueue[self] # ⟨⟩
                 ∧ rcvd' = [rcvd EXCEPT ![self] = Head(procQueue[self])]
                 ∧ procQueue' = [procQueue EXCEPT ![self] = Tail(procQueue[self])]
                 ∧ IF rcvd'[self].kind = "request"
                     THEN ∧ IF father[self] = nil
                             THEN ∧ IF reqCS[self] = TRUE
                                     THEN ∧ next' = [next EXCEPT ![self] = rcvd'[self].snd]
                                          ∧ pc' = [pc EXCEPT ![self] = "lb_ret"]
                                          ∧ UNCHANGED tokPresent
                                     ELSE ∧ tokPresent' = [tokPresent EXCEPT ![self] = FALSE]
                                          ∧ pc' = [pc EXCEPT ![self] = "lb_call1"]
                                          ∧ UNCHANGED next
                             ELSE ∧ pc' = [pc EXCEPT ![self] = "lb_call2"]
                                  ∧ UNCHANGED ⟨ next, tokPresent ⟩
                     ELSE ∧ IF rcvd'[self].kind = "give_token"
                             THEN ∧ tokPresent' = [tokPresent EXCEPT ![self] = TRUE]
                             ELSE ∧ TRUE
                                  ∧ UNCHANGED tokPresent
                          ∧ pc' = [pc EXCEPT ![self] = "end_recv_msg"]
                          ∧ UNCHANGED next
            ELSE ∧ pc' = [pc EXCEPT ![self] = "end_recv_msg"]
                 ∧ UNCHANGED ⟨ procQueue, rcvd, next, tokPresent ⟩
        ∧ UNCHANGED ⟨ stack, father, reqCS ⟩

lb_call1(self) ≜
        ∧ pc[self] = "lb_call1"
        ∧ procQueue' = [procQueue EXCEPT ![(rcvd[self].snd)] =
              Append(procQueue[(rcvd[self].snd)], ([snd ↦ self, kind ↦ "give_token"]))]
        ∧ pc' = [pc EXCEPT ![self] = "lb_ret"]
        ∧ UNCHANGED ⟨ stack, rcvd, father, next, reqCS, tokPresent ⟩

end_recv_msg(self) ≜
          ∧ pc[self] = "end_recv_msg"
          ∧ pc' = [pc EXCEPT ![self] = Head(stack[self]).pc]
          ∧ rcvd' = [rcvd EXCEPT ![self] = Head(stack[self]).rcvd]
          ∧ stack' = [stack EXCEPT ![self] = Tail(stack[self])]
          ∧ UNCHANGED ⟨ procQueue, father, next, reqCS, tokPresent ⟩

handle_messages(self) ≜ recv_msg(self) ∨ lb_ret(self) ∨ lb_call2(self)
                                       ∨ lb_call1(self) ∨ end_recv_msg(self)

rlcs(self) ≜
       ∧ pc[self] = "rlcs"
       ∧ reqCS' = [reqCS EXCEPT ![self] = FALSE]
       ∧ IF next[self] # nil
           THEN ∧ procQueue' = [procQueue EXCEPT ![next[self]] =
                       Append(procQueue[next[self]], ([snd ↦ self, kind ↦ "give_token"]))]
                ∧ tokPresent' = [tokPresent EXCEPT ![self] = FALSE]
                ∧ next' = [next EXCEPT ![self] = nil]
           ELSE ∧ TRUE
                ∧ UNCHANGED ⟨ procQueue, next, tokPresent ⟩
       ∧ pc' = [pc EXCEPT ![self] = "end_rlcs"]
       ∧ UNCHANGED ⟨ stack, rcvd, father ⟩
```

```
113 end_rlcs(self) ≜
114             ∧ pc[self] = "end_rlcs"
115             ∧ pc' = [pc EXCEPT ![self] = Head(stack[self]).pc]
116             ∧ stack' = [stack EXCEPT ![self] = Tail(stack[self])]
117             ∧ UNCHANGED ⟨ procQueue, rcvd, father, next, reqCS, tokPresent ⟩
118
119 release_cs(self) ≜ rlcs(self) ∨ end_rlcs(self)
120
121 init(self) ≜
122          ∧ pc[self] = "init"
123          ∧ tokPresent' = [tokPresent EXCEPT ![self] = father[self] = self]
124          ∧ IF father[self] = self
125              THEN ∧ father' = [father EXCEPT ![self] = nil]
126              ELSE ∧ TRUE
127                   ∧ UNCHANGED father
128          ∧ pc' = [pc EXCEPT ![self] = "ncs"]
129          ∧ UNCHANGED ⟨ procQueue, stack, rcvd, next, reqCS ⟩
130
131 ncs(self) ≜
132          ∧ pc[self] = "ncs"
133          ∧ ∨ ∧ stack' = [stack EXCEPT ![self] = ⟨ [ procedure ↦ "request_cs", pc ↦ "continue1" ] ⟩
134                                       ∘ stack[self]]
135             ∧ pc' = [pc EXCEPT ![self] = "rqcs"]
136             ∧ UNCHANGED rcvd
137            ∨ ∧ stack' = [stack EXCEPT ![self] = ⟨ [ procedure ↦ "handle_messages",
138                                                    pc ↦ "continue2",
139                                                    rcvd ↦ rcvd[self] ] ⟩
140                                                    ∘ stack[self]]
141             ∧ rcvd' = [rcvd EXCEPT ![self] = [snd ↦ 0 , kind ↦ ""]]
142             ∧ pc' = [pc EXCEPT ![self] = "recv_msg"]
143          ∧ UNCHANGED ⟨ procQueue, father, next, reqCS, tokPresent ⟩
144
145 try(self) ≜
146          ∧ pc[self] = "try"
147          ∧ IF tokPresent[self] = FALSE
148              THEN ∧ stack' = [stack EXCEPT ![self] = ⟨ [ procedure ↦ "handle_messages",
149                                                          pc ↦ "try",
150                                                          rcvd ↦ rcvd[self] ] ⟩
151                                                          ∘ stack[self]]
152                   ∧ rcvd' = [rcvd EXCEPT ![self] = [snd ↦ 0 , kind ↦ ""]]
153                   ∧ pc' = [pc EXCEPT ![self] = "recv_msg"]
154              ELSE ∧ pc' = [pc EXCEPT ![self] = "cs"]
155                   ∧ UNCHANGED ⟨ stack, rcvd ⟩
156          ∧ UNCHANGED ⟨ procQueue, father, next, reqCS, tokPresent ⟩
157
158 cs(self) ≜
159          ∧ pc[self] = "cs"
160          ∧ TRUE
161          ∧ pc' = [pc EXCEPT ![self] = "exit"]
162          ∧ UNCHANGED ⟨ procQueue, stack, rcvd, father, next, reqCS, tokPresent ⟩
163
164 exit(self) ≜
165          ∧ pc[self] = "exit"
166          ∧ stack' = [stack EXCEPT ![self] = ⟨ [ procedure ↦ "release_cs",
167                                                 pc ↦ "ncs" ] ⟩
168                                                 ∘ stack[self]]
169          ∧ pc' = [pc EXCEPT ![self] = "rlcs"]
170          ∧ UNCHANGED ⟨ procQueue, rcvd, father, next, reqCS, tokPresent ⟩
```

```
continue1(self) ≜
            ∧ pc[self] = "continue1"
            ∧ IF tokPresent[self] = TRUE
                THEN ∧ pc' = [pc EXCEPT ![self] = "cs"]
                ELSE ∧ pc' = [pc EXCEPT ![self] = "try"]
            ∧ UNCHANGED ⟨ procQueue, stack, rcvd, father, next, reqCS, tokPresent ⟩


continue2(self) ≜
            ∧ pc[self] = "continue2"
            ∧ pc' = [pc EXCEPT ![self] = "ncs"]
            ∧ UNCHANGED ⟨ procQueue, stack, rcvd, father, next, reqCS, tokPresent ⟩



Site(self) ≜ init(self) ∨ ncs(self) ∨ try(self) ∨ cs(self)
          ∨ exit(self) ∨ continue1(self) ∨ continue2(self)



Next ≜ (∃ self ∈ ProcSet:  ∨ request_cs(self) ∨ handle_messages(self)
                           ∨ release_cs(self))
      ∨ (∃ self ∈ Peers: Site(self))
      ∨ (* Disjunct to prevent deadlock on termination *)
        ((∀ self ∈ ProcSet: pc[self] = "Done") ∧ UNCHANGED vars)



Spec ≜ Init ∧ □[Next]_vars

Termination ≜ ◇(∀ self ∈ ProcSet: pc[self] = "Done")
```

1.2 Leader election algorithm

Leader election algorithm was proposed by Dolev, Klawe, and Rodeh [Dolev 1982] for electing a leader in a unidirectional ring. The model of this algorithm that we use here is the direct translation of it's implementation in Promela from SPIN library to PlusCal-2 language.

```
1  algorithm Leader
2  extends Naturals,Sequences          (* standard modules *)
3
4  constants
5    N,                                (* Number of processes *)
6    I                                 (* node given the smallest number *)
7
8  variable
9      net = [p ∈ 0..(N-1) ↦ ⟨⟩]    (* the network represented as a queue *)
10
11 definition send(ch, msg) ≜
12                 [net EXCEPT ![ch] = Append(@, msg)]
13
14 fair process Node[N]
15   variables
16     active = TRUE, know_winner = FALSE,
17     mynumber = (N+I-self)%(N+1), neighbourR = 0,
18     maximum = (N+I-self)%(N+1), in = self-1, out = self%N,
19     msg = ⟨⟩, winner = FALSE
20
21   begin
22   start:
23     net[out] := Append(net[out], [type ↦ "one", number ↦ mynumber]);
24   forever:
25     loop
26       if Len(net[in]) > 0 then
27         msg := Head(net[in]);
28         if msg.type = "one" then
29           if active then
30             if msg.number # maximum then
31               net[out] := send(out, [type ↦ "two", number ↦ msg.number]);
32               neighbourR := msg.number;
33             else
34               know_winner := TRUE;
35               net[out] := send(out, [type ↦ "winner", number ↦ msg.number]);
36             end if
37           else
38             net[out] := send(out, [type ↦ "one", number ↦ msg.number]);
39           end if;
40       else if msg.type = "two" then
41           if active then
42             if (neighbourR > msg.number) ∧ (neighbourR > maximum) then
43               maximum := neighbourR;
44               net[out] := send(out, [type ↦ "one", number ↦ neighbourR]);
45             else
46               active := FALSE;
47             end if
48           else
49             net[out] := send(out, [type ↦ "two", number ↦ msg.number]);
50           end if;
```

```
51          else if msg.type = “winner” then
52            if msg.number = mynumber then
53              winner := TRUE;
54            end if;
55            if ~know_winner then
56              net[out] := send(out, [type ↦ “winner”, number ↦ msg.number]);
57            end if;
58          end if;
59          net[in] := Tail(net[in]);
60        end if;
61      end loop;
62    end process
63    (* Temporal property for model checking *)
64    temporal ∃ p ∈ Node : ◇ Node[p].winner
65  end algorithm
66
67  (* Invariant for model checking *)
68  invariant ∀ p ∈ Node : Node[p].winner => (∀ q ∈ Node \ {p} : ∼ Node[q].winner)
69  (* Finite instance for model checking *)
70  instances N = 3, I = 1
```

1.3 Concurrent sorting algorithm

The main idea of sorting algorithm is to concurrently sort N random numbers. It is taken from the Spin distribution and also known to be an example in which partial-order reduction works well in Spin. The algorithm contains definitions for three different types of processes: left, middle and right. The process left generates the random numbers using the definition RANDOM and passes them on to the network. The process middle has seven instances and each of the instances compare the new number with the one they already have. If its found to be larger, then it passes it on to the next process through the network otherwise it keeps the new number and passes the one it already had. The process right will always receive the number that is larger than all the other numbers held by the instances of process middle.

Below we have its model in PlusCal-2 language along with it's translation in intermediate format.

1.3.1 PlusCal-2 model

```
1  algorithm sort
2  extends Naturals, Sequences                          (* standard modules *)
3
4  constants
5    N                                                  (* Number of processes *)
6  definition firstId ≜ lowerbound(middle)
7
8  definition lastId ≜ upperbound(middle)
9
10 definition RANDOM(seed) ≜ ((seed * 3 + 14) % 100)    (* Calculate random number *)
11
12 variable
13       network = [p ∈ firstId..lastId+1 ↦ ⟨⟩]        (* Network for communication *)
14
15 process left[1]                                      (* Generates random numbers *)
16   variables
17     counter = 0, seed = 15, out = firstId
18   begin
19   left-lbl:
20
21  loop
22   network[out] := Append(network[out], [value ↦ seed]);
23   counter := counter + 1;
24   seed := RANDOM(seed);
25   branch
26    counter = N then
27      break;
28   or
29    counter ≠ N then
30      skip;
31   end branch;
32  end loop;
33 end process
```

```
34 process middle[N-1]                    (* Process in middle, sort the numbers *)
35 variables in = self, out = self + 1
36 variables counter = 0, myval = 0, nextval = 0, msg = ⟨⟩
37
38 begin
39 start:
40   when Len(network[in])  0;
41   msg := Head(network[in]);
42   myval := msg.value;
43   network[in] := Tail(network[in]);
44 mid-lbl:
45  loop
46        branch
47         Len(network[in]) > 0 ∧ counter < N then
48           msg := Head(network[in]);
49           nextval := msg.value;
50           if nextval ≥ myval then
51             network[out] := Append(network[out], [value ↦ nextval]);
52           else
53             network[out] := Append(network[out], [value ↦ myval]);
54             myval := nextval;
55           end if;
56           network[in] := Tail(network[in]);
57           counter := counter + 1;
58        or
59         counter ≥ (N - self + 1) then
60           break;
61        end branch;
62  end loop;
63 end process
64
65
66 process right[1]                    (* right process accepts the biggest number *)
67 variables in = self, biggest = 0, msg = ⟨⟩
68 begin
69 right-lbl:
70   when Len(network[in]) > 0;
71   msg := Head(network[in]);
72   biggest := msg.value;
73   network[in] := Tail(network[in]);
74 end process
75
76 end algorithm
77
78 instance N = 10
```

1.3.2 Concurrent sorting algorithm in intermediate format

```
left-lbl(self):
        network[_left_data[self].out] := Append(network[_left_data[self].out],
                                                [value ↦ _left_data[self].seed])
        _left_data[self].counter := (_left_data[self].counter + 1)
        _left_data[self].seed := RANDOM(_left_data[self].seed)
        branch
          (_left_data[self].counter = N) then
            _pc[self] := "Done"
        or
          (_left_data[self].counter # N) then
            _pc[self] := "left-lbl"
        end branch

start(self):
           branch
             (Len(network[_middle_data[self].in]) > 0) then
               _middle_data[self].msg := Head(network[_middle_data[self].in])
               _middle_data[self].myval := _middle_data[self].msg.value
               network[_middle_data[self].in] = Tail(network[_middle_data[self].in])
               _pc[self] = "mid-lbl"
           end branch

mid-lbl(self):
          branch
            ((Len(network[_middle_data[self].in]) > 0) ∧ (_middle_data[self].counter < N))  then
              _middle_data[self].msg := Head(network[_middle_data[self].in])
              _middle_data[self].nextval := _middle_data[self].msg.value
              branch
                (_middle_data[self].nextval >= _middle_data[self].myval)  then
                  network[_middle_data[self].out] := Append(network[_middle_data[self].out],
                                                            [value ↦ _middle_data[self].nextval])
                  network[_middle_data[self].in] := Tail(network[_middle_data[self].in])
                  _middle_data[self].counter := (_middle_data[self].counter + 1)
                  _pc[self] := "mid-lbl"
              or
                ~((_middle_data[self].nextval >= _middle_data[self].myval))  then
                  network[_middle_data[self].out] := Append(network[_middle_data[self].out],
                                                            [value ↦ _middle_data[self].myval])
                  _middle_data[self].myval := _middle_data[self].nextval]
                  network[_middle_data[self].in] := Tail(network[_middle_data[self].in])
                  _middle_data[self].counter := (_middle_data[self].counter + 1)
                  _pc[self] := "mid-lbl"
              end branch
          or
            (_middle_data[self].counter >= ((N - self) + 1))  then
              _pc[self] := "Done"
          end branch

right-lbl(self):
        branch
          (Len(network[_right_data[self].in]) > 0) then
            _right_data[self].msg := Head(network[_right_data[self].in])
            _right_data[self].biggest := _right_data[self].msg.value
            network[_right_data[self].in] := Tail(network[_right_data[self].in])
            _pc[self] := "Done"
        end branch
```

Abstract

Designing sound algorithms for concurrent and distributed systems is subtle and challenging. These systems are prone to deadlocks and race conditions, which occur in particular interleavings of process actions and are therefore hard to reproduce. It is often nontrivial to precisely state the properties that are expected of an algorithm and the assumptions on the environment under which these properties should hold. Formal verification is a key technique to model the system and its properties and then perform verification by means of model checking.

Formal languages like TLA$^+$ have the ability to describe complicated algorithms quite concisely, but algorithm designers often find it difficult to model an algorithm in the form of formulas. In this thesis, we present PlusCal-2 that aims at being similar to pseudo-code while being formally verifiable. PlusCal-2 improves upon Lamport's PlusCal algorithm language by lifting some of its restrictions and adding new constructs. Our language is intended for describing algorithms at a high level of abstraction. It resembles familiar pseudo-code but is quite expressive and has a formal semantics. Finite instances of algorithms described in PlusCal-2 can be verified through the TLC model checker. The second contribution presented in this thesis is a study of partial-order reduction methods using conditional and constant dependency relation.

To compute conditional dependency for PlusCal-2 algorithms, we exploit their locality information and present them in the form of independence predicates. We also propose an adaptation of a dynamic partial-order reduction algorithm for a variant of the TLC model checker. As an alternative to partial-order reduction based on conditional dependency, we also describe a variant of a static partial-order reduction algorithm for the TLC model checker that relies on constant dependency relation. We also present our results for the experiments along with the proof of correctness.

Keywords: Distributed algorithms, algorithm language, model checking, PlusCal-2, partial-order reduction

Résumé

La conception d'algorithmes pour les systèmes concurrents et répartis est subtile et difficile. Ces systèmes sont enclins à des blocages et à des conditions de course qui peuvent se produire dans des entrelacements particuliers d'actions de processus et sont par conséquent difficiles à reproduire. Il est souvent non-trivial d'énoncer précisément les propriétés attendues d'un algorithme et les hypothèses que l'environnement est supposé de satisfaire pour que l'algorithme se comporte correctement. La vérification formelle est une technique essentielle pour modéliser le système et ses propriétés et s'assurer de sa correction au moyen du model checking.

Des langages formels tels TLA$^+$ permettent de décrire des algorithmes compliqués de manière assez concise, mais les concepteurs d'algorithmes trouvent souvent difficile de modéliser un algorithme par un ensemble de formules. Dans ce mémoire nous présentons le langage PlusCal-2 qui vise à allier la simplicité de pseudo-code à la capacité d'être vérifié formellement. PlusCal-2 améliore le langage algorithmique PlusCal conçu par Lamport en levant certaines restrictions de ce langage et en y ajoutant de nouvelles constructions. Notre langage est destiné à la description d'algorithmes à un niveau élevé d'abstraction. Sa syntaxe ressemble à du pseudo-code mais il est tout à fait expressif et doté d'une sémantique formelle. Des instances finies d'algorithmes écrits en PlusCal-2 peuvent être vérifiées à l'aide du model checker TLC. La deuxième contribution de cette thèse porte sur l'étude de méthodes de réduction par ordre partiel à l'aide de relations de dépendance conditionnelle et constante.

Pour calculer la dépendance conditionnelle pour les algorithmes en PlusCal-2 nous exploitons des informations sur la localité des actions et nous générons des prédicats d'indépendance. Nous proposons également une adaptation d'un algorithme de réduction par ordre partiel dynamique pour une variante du model checker TLC. Enfin, nous proposons une variante d'un algorithme de réduction par ordre partiel statique (comme alternative à l'algorithme dynamique), s'appuyant sur une relation de dépendance constante, et son implantation au sein de TLC. Nous présentons nos résultats expérimentaux et une preuve de correction.

Mots-clés: Algorithmes distribués, langage algorithmique, model-checking, PlusCal-2, réduction par ordre partiel