Verification of networks of communicating processes

Reachability problems and decidability issues

OTHMANE REZINE
Abstract


Computer systems are used in almost all aspects of our lives and our dependency on them keeps on increasing. When computer systems are used to handle critical tasks, any software failure can cause severe human and/or material losses. Therefore, for such applications, it is important to detect software errors at an early stage of software development. Furthermore, the growing use of concurrent and distributed programs exponentially increases the complexity of computer systems, making the problem of detecting software errors even harder (if not impossible). This calls for defining systematic and efficient techniques to evaluate the safety and the correctness of programs. The aim of Model-Checking is to analyze automatically whether a given program satisfies its specification. Early applications of Model-Checking were restricted to systems whose behaviors can be captured by finite graphs, so called finite-state systems. Since many computer systems cannot be modeled as finite-state machines, there has been a growing interest in extending the applicability of Model-Checking to infinite-state systems.

The goal of this thesis is to extend the applicability of Model Checking for three instances of infinite-state systems: Ad-Hoc Networks, Dynamic Register Automata and Multi Pushdown Systems. Each one of these instances models challenging types of networks of communicating processes. In both Ad-Hoc Networks and Dynamic Register Automata, communication is carried through message passing. In each type of network, a graph topology models the communication links between processes in the network. The graph topology is static in the case of Ad-Hoc Networks while it is dynamic in the case of Dynamic Register Automata. The number of processes in both types of networks is unbounded. Finally, we consider Multi Pushdown Systems, a model used to study the behaviors of concurrent programs composed of sequential recursive sequential programs communicating through a shared memory.

Keywords: program verification, model checking, infinite-state systems, distributed programs, concurrent programs, networks of communicating processes, reachability, termination, decidability

Othmane Rezine, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Othmane Rezine 2017

ISSN 1651-6214
urn:nbn:se:uu:diva-334788 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-334788)
à Beba et à Yema.
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.


Reprints were made with permission from the publishers.
Contribution to the papers

*Paper I:*
The ideas originated and were developed in discussion with the authors. I had the main role in writing the paper.

*Paper II:*
The idea of the model came from collaborating with Ahmet Kara. The ideas in the paper and the proofs were discussed together with the authors. I had the main role in writing the paper.

*Paper III:*
This work is a continuation of Paper II. The ideas originated and were developed in discussion with the authors. I had the main role in writing the paper.

*Paper IV:*
I took part in the discussions of the theoretical part of the paper. Together with Jari Stenman, we implemented the automatic code-to-code translation into a prototype and conducted the experiments. I wrote together with Jari the code-to-code translation and the experimentation sections in the paper.
Other publications

The following papers were not included in this thesis.


Acknowledgements

First and foremost, I would like to thank my main advisor, Prof. Parosh Aziz Abdulla, who gave me the opportunity to pursue a PhD in the Algorithmic Program Verification group at Uppsala University. Parosh, it has been a privilege to undergo my PhD studies under your direction. I have learned a lot from collaborating and writing papers with you. Your ability to explain intricate ideas in a clear and thorough manner made your seminars and courses a unique opportunity for me to learn about formal verification. Besides our scientific discussions, I always took great pleasure from conversing with you on all sorts of topics. Thank you for your continuous help and support during the whole PhD.

My gratitude also goes to my second advisor Dr. Mohamed Faouzi Atig. Faouzi, thank you for having supported me all along my PhD, both scientifically and personally. I enjoyed your enthusiasm and appreciated your ideas and our discussions in every publication we wrote together. I am deeply grateful for your help and to our everyday multi-lingual discussions on the way to the coffee room.

I would also like to thank Prof. Bengt Jonsson, my third advisor, for his guidance, moral support, encouragement, and also for all the valuable graduate courses he gave and which I attended with great pleasure.

During my PhD, I particularly enjoyed collaborating with (in alphabetical order) C. Aiswarya, Giorgio Delzanno, Ahmet Kara, Subham Modi, Gaurav Saini, Arnaud Sangnier, Jari Stenman, and Riccardo Traverso. Moreover, I am deeply grateful for the hospitality of Prof. K. Narayan Kumar and his then student Dr. Prakash Saivasan during my visit to the Chennai Mathematical Institute. I would also like to show my gratitude towards past and present members of the Algorithmic Program Verification group (in alphabetical order): Ahmed, Aiswarya, Amendra, Carl, Diep, Frédéric, Jari, Jonathan, Lisa, Lukáš, Phong, Quy, Sofia, Yu-Fang and Yunyun. Ivan Christoff, although you would refute it, you have done much more than what your job had required, and I am deeply grateful for that, may you rest in peace. Thank you Mats Daniels for giving me the opportunity to teach in all the interesting courses in which I took part. In particular, I took great pleasure in assisting for courses given by Arnold Neville Pears, Edith Ngai, Joachim Parrow, Kai Lampka, Philippe Rümmer, and Mohamed Faouzi Atig. My gratitude also goes to the administrative staff of the Department of Information Technology (Ulrika Andersson, Ulrika Jaresund and Elisabeth Lindqvist to name a few) who patiently addressed each one of my administrative requests.
My profound gratitude goes to every friend and colleague who assisted me during the last few weeks before the submission of my thesis. In particular, Faouzi, Parosh, Diep, Phong, Yunyun and Quy, thank you for proof reading the chapters of my thesis. Ahmed, Rafik, Parosh, Yema and Caroline Wänlund thank you very much for helping me with the translations to Arabic, French and Swedish. I am grateful to all my friends and colleagues who made my PhD not only a scientific and an academic adventure, but also an enriching human experience.

Finally, I would like to express my deepest gratitude and love to my family. Ahmed, you are the first person who initiated me to mathematical proofs and you are one of the main reasons why I started thinking of doing a PhD. I am deeply grateful that you believed in me and in my capabilities. Rafik and Lalla Zhor, thank you for your continuous support and for all the warmth you have brought with you each time you visited. Yema, merci de m’avoir très tôt initié à la théorie des ensembles et au calcul en différentes bases et de m’avoir montré que les mathématiques, comme toute autre science, pouvaient être un jeu auquel on pouvait prendre plaisir. Beba, que ton âme repose en paix, merci de m’avoir appris la rigueur, le dépassement de soi ainsi que le droit et le devoir de questionner, d’interroger et de (se) remettre en cause. Thérèse and Ahmed, thank you so much for being a safe shelter I could use each time I needed it and thanks to Yasmine and to Hakim for bringing so much joy to our lives.

This thesis was funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC) and by the Programming Platform for Future Wireless Sensor Networks Project (PROFUN) projects.
Sammanfattning


nikationen mellan processerna i denna variant är asynkron eftersom meddelanden som skickas kan tas emot vid en senare tidpunkt. Resultaten presenteras i kapitel 5.

Les systèmes informatiques sont aujourd’hui présents presque partout et nous en dépendons de plus en plus. Ils peuvent être de grands systèmes utilisés, par exemple, pour prédire la météo, décoder le génome humain ou analyser les tendances financières du moment. Ils peuvent également être de petits appareils utilisés dans notre vie quotidienne pour contrôler la température dans notre réfrigérateur ou télécharger des courriels sur une montre sophistiquée. Même les équipements qui n’avaient pas de puissance de calcul dans le passé, tels que les fours à micro-ondes et les clés de voiture, sont maintenant équipés avec un microcontrôleur sur lequel tourne un logiciel embarqué. Tous ces exemples montrent à quel point l’utilisation de systèmes et de composants informatiques est devenue omniprésente.

Les utilisateurs de systèmes informatiques sont familiers avec le terme défaut (en anglais bug) qui dénote un comportement erroné ou un défaut du système. Les défauts sont souvent dus à une défaillance du logiciel qui contrôle ces systèmes. La défaillance d’un logiciel peut être due au fait qu’un programme ne gère pas une donnée ou une situation particulière. Afin de détecter les défaillances logicielles, la pratique la plus courante consiste à tester le programme pour différentes entrées de données et différentes situations possibles et à vérifier que le programme répond correctement dans tous ces cas de figure. Malheureusement, les tests ne peuvent que découvrir des défauts et ne peuvent pas garantir leur absence. L’absence de garantie de la méthode de test est problématique lorsqu’elle est appliquée à des systèmes critiques, c’est-à-dire à des systèmes dans lesquels une erreur peut induire de sérieuses conséquences qui peuvent parfois être dramatiques. Les logiciels en charge de la régulation de la vitesse d’une voiture, de la surveillance d’équipements médicaux ou du calcul de la trajectoire d’une fusée sont autant d’exemples de programmes utilisés dans des systèmes critiques. Les erreurs produites par ces programmes peuvent causer de graves pertes humaines et matérielles.

Les limites du test font appel à une approche qui peut garantir l’absence de défauts en analysant la validité des programmes de manière plus systématique. Idéalement, une telle approche devrait nous fournir des outils capables de trouver des défauts, même si ces défauts sont difficiles à détecter. Elle devrait également fournir des techniques pour prouver formellement la validité des programmes par rapport à certaines spécifications prédéfinies. Un ensemble de méthodes regroupées sous le nom générique de méthodes de vérification formelles essaie d’atteindre ces objectifs en utilisant des formalismes mathématiques. La vérification de modèles (en anglais Model-Checking) est l’une
de ces techniques. Elle prend comme données d’entrée un modèle mathématique d’un système et une propriété spécifiant comment le système doit se comporter. Basée sur des procédures automatiques, la technique de vérification de modèles vérifie si la spécification donnée en entrée est satisfaite par le modèle, c’est-à-dire si chaque comportement possible du modèle respecte la spécification. Une fois terminée, le résultat de l’application de la vérification de modèles doit soit fournir une preuve montrant la validité du modèle avec la propriété, soit fournir un contre-exemple démontrant un comportement du modèle qui viole la propriété. Un exemple de propriétés utilisées pour analyser la sûreté d’un modèle est la propriété d’atteignabilité d’un état. Si on considère un état particulier du système, la propriété d’atteignabilité caractérise le fait que le modèle puisse se retrouver dans cet état. Le problème qui consiste à vérifier si un modèle respecte ou non cette propriété s’appelle le problème d’atteignabilité. Étant donné un état considéré comme indésirable ou dangereux, si la réponse au problème d’atteignabilité de cet état est négative, alors nous sommes sûrs que le système ne finira pas dans ce mauvais état, ce qui renforce notre confiance dans la sûreté du système.

Nous pouvons faire face à deux défis si on veut appliquer la vérification de modèles sur une classe de programmes. Le premier défi consiste à savoir s’il est possible de concevoir un programme informatique capable de fournir une réponse au problème de vérification de modèles. Si c’est le cas, nous disons que le problème de vérification de modèles pour cette classe de systèmes est décidable. Sinon, le problème est indécidable. En supposant que le problème de vérification de modèles est décidable, le deuxième défi consiste à évaluer sa complexité. La complexité d’un problème consiste à évaluer combien de temps et d’espace mémoire est nécessaire à un programme informatique pour qu’il puisse fournir une solution. La décidabilité et la complexité de chaque instance du problème de vérification de modèles dépendent du type de programme pris en entrée et de la propriété considérée.

La technique de vérification de modèles a été appliquée avec succès à plusieurs classes de programmes pouvant être modélisés avec un nombre fini d’états. Le but de cette thèse est d’étendre l’applicabilité de la vérification de modèles aux modèles qui peuvent contenir un nombre illimité d’états, aussi appelés systèmes à état infinis. En particulier, nous considérons trois instances de tels systèmes qui modélisent des réseaux de processus communicants, c’est-à-dire un ensemble de processus qui communiquent entre eux. Un processus est un terme générique que nous utilisons pour désigner une entité indépendante de traitement informatique.

Le premier modèle, les Réseaux Ad-Hoc, modélisent un réseau de processus qui communiquent entre eux en échangeant des messages. Chaque processus dans un réseau Ad-Hoc est une machine à états finis avec des transitions qui modélisent des actions locales aux processus ou des actions de communication avec d’autres processus. Un graphe, appelé topologie du réseau, définit pour chaque processus l’ensemble des voisins directs du processus dans le réseau.
Lorsqu’un processus dans un réseau Ad-Hoc envoie un message, le message est diffusé à tous ses voisins directs. De plus, nous supposons que la topologie de tels réseaux est statique, c’est-à-dire qu’elle n’évolue pas durant la vie du réseau. Par ailleurs, la taille du réseau, définie comme le nombre de processus présents dans le réseau, n’est pas à priori bornée. La vérification de modèles d’un réseau ad-hoc pour toute propriété basique est indécidable. La question que nous abordons dans le Chapitre 4 et dans l’article I de cette thèse est de savoir s’il existe une catégorie particulière de réseaux Ad-Hoc pour laquelle il est possible de vérifier qu’aucun des processus dans le réseau ne finisse dans un état considéré comme mauvais.

Dans le deuxième type de réseaux que nous considérons, **Automates à Registres Dynamiques**, les processus sont des machines à états finis équipées d’un ensemble de registres. Les registres sont utilisés par les processus pour stocker les adresses d’autres processus dans le réseau. Un processus $a$ dans un tel réseau peut envoyer un message ou une adresse à un autre processus $b$ si $a$ connaît l’adresse de $b$. De plus, les processus peuvent créer de nouveaux processus et modifier la topologie du réseau. Par conséquent, les réseaux d’Automates à Registres Dynamiques sont dynamiques et leurs formes et leurs tailles peuvent évoluer au fil du temps. Dans cette thèse, nous abordons la décidabilité de la vérification du problème d’atteignabilité à un état pour deux variantes de tels réseaux. Dans l’article II, nous considérons la variante synchrone, dans laquelle les actions d’envoi d’un message et de sa réception se produisent simultanément. Dans l’article III, nous considérons la variante asynchrone, dans laquelle chaque processus est équipé d’une boîte aux lettres. La communication entre les processus de cette variante est asynchrone, car les messages envoyés peuvent être reçus ultérieurement. Les contributions de ces deux articles sont largement présentées dans le chapitre 5.

Enfin, le troisième modèle que nous considérons est le modèle d’**Automates à Piles Multiples**. Les automates à piles multiples sont utilisés pour modéliser des *programmes concurrents* qui utilisent des variables couvrant un domaine de données fini. Un programme concurrent peut être considéré comme un réseau composé d’un ensemble de programmes séquentiels et récursifs (appelés fils d’instruction ou *threads*). Les threads communiquent entre eux en manipulant un ensemble de variables partagées. Dans l’article 6, nous étudions la complexité de la vérification de modèles d’une sous-classe particulière de programmes concurrents. De plus, nous montrons comment transformer un programme concurrent de cette sous-classe en un programme séquentiel. Le but de cette transformation est de pouvoir analyser des programmes concurrents en utilisant des outils conçus pour la vérification de programmes séquentiels. Le chapitre 6 de cette thèse présente en détails nos contributions à ce sujet.
ملخص

أنظمة الحواسيب موجودة تقريبا في كل مكان ويزداد كل يوم اعتقادنا عليها. قد تكون أنظمة كبيرة تستخدم على سبيل المثال، للتدريب بالمخط، لفك شفرة الوراثة البشرية أو لتحليل الاتجاهات المالية. يمكن أيضا أن تكون أجهزة صغيرة قد نستخدمها يوميا للتحكم على درجة حرارة التلاجة أو لجلب رسائل البريد الإلكتروني إلى ساعة يد متطورة. حتى المعدات التي لم تكون تستخدم أي قدرة حساسية في الماضي، مثل أفران الميكروويف ومفاتيح السيارة، يتم بناؤها اليوم مع وحدة تحكم مدمجة التي تستخدم بعض البرمجيات. كل هذه الأمثلة تبين مدى انتشار استخدام أنظمة وأجهزة الحاسوب.

مستخدمي أنظمة الكمبيوتر يستعملون مصطلح علامة (bug) ليعبرون على سلوك خاطئ للبرامج المعلوماتي العلل غالبا ما يكون سببها فشل البرنامج المعلوماتي لكونه لا يتعامل مع مدخلات أو مواقف معينة. اختبار البرمجيات بواسطة مدخلات ومواقف مختلفة هو الطريقة الأكثر استعمالا للكشف عن إمكانية فشلها. لسوء الحظ، قد يمكّن الاختبار أن يكشف عن وجود عللا لكنه لا يستطيع ضمان غيابها صвозيا ضمان نجاعة البرمجيات بشكل مشكلة في حالة استعمالها في أنظمة حرة. أي أنظمة يمكن أن تؤدي فيها العلل إلى خسائر مادية وبشرية جسيمة. البرامج المسؤولّة عن تنظيم سرعة سيارة، على مراقبة أجهزة طبية أو على حساب مسار صاروخي، تكون أمثلة برامج مستخدمة في أنظمة حرة.

نقاش الاختبار تعودنا إلى البحث على منهج يضمن عدم وجود أخطاء في البرمجيات. مثلاً، ينبغي أن يوفر لنا هذا المنهج أدوات قادرة على إيجاد العلل، حتى العلل التي يصعب اكتشافها. ينبغي أيضا أن يوفر هذا المنهج تقنيات لإثبات صحة البرامج بشكلنا لمساواة محددة مسبقا. هناك مجموعة من التقنيات المعروفة بأساليب التحقق النظرية. تحول هذه الأساليب النظرية لتحقيق هذه الأهداف باستخدام نظريات رياضية. أحد هذه الأساليب هو فحص النموذج (Model-Checking).

فحص النموذج يأخذ كمداخلة نموذج رياضي لنظام معلوماتي وصناة تحكم للنظام. يتصرف. استنادا إلى إجراءات أو تفاضلية، تحقق تقنية فحص النموذج من ما إذا كانت الصفة المدخلة راضية عن النموذج. أي ما إذا كان لكل سلوك محتمل لنموذج النظام يفي بالمواصفات عند إنهاء تطبيق الفحص النموذجي، ينبغي أن تكون النتيجة إما ديلاً على صحة النموذج طبقاً للصفي المدخلة. أو أن يقدم مثالاً مضيفاً يدل على سلوك لنموذج ينتهك الصفة المدخلة.

يمكن أن نواجه تحديين عند تطبيق تقنية فحص النموذج على فئة من الأنظمة المعلوماتية. التحدي الأول هو التحقق مما إذا كان من الممكن تصميم برنامج حاسوب يمكنه الإجابة على مسألة فحص أي أنظمة من
لفحة. في هذا الحالة، نقول أن مشكلة فحص النموذج لهذه الفئة من الأنظمة تقريرية، وإذا فإنها غير تقريرية. بافتراض أن مشكلة فحص النموذج تقريرية، فإن التحليلي الثاني ينتمي في تقييم تفريعها. تعقيد مشكلة يتمثل في تعبير مقدار الوقت الألزام لبرنامج حاسوي لتوفير حل تقريري ودرجة التفعيل للمشكلة فحص النموذج تغيير حسب فئة النموذج وحسب النص المعمّنة التي تأخذهم كمضان.

لقد تم تطبيق تقنية فحص النموذج بنجاج على عدة فئات من البرامج التي يمكن تمدحها بعده محدود من الحالات. الهدف من هذا العمل هو توسيع نطاق تطبيق تكنولوجيا فحص النموذج للنماذج التي قد تحتوي على عدد غير محدود من الحالات، والتي تسمى أيضا نظم دو حالات انتهاجية وتعمير، على وجه الخصوص، ثلاث أنواع من هذه الأنظمة التي تشكل نماذج للشبكات عمليات متواصلة. أي مجموعة من العمليات التي تواصل مع بعضها البعض، العملية هي مصطلح عام يستخدم للدلالة على كيان حيوية مستقل.

النموذج الأول، وهو نموذج (Ad-Hoc Networks)، يشكل نموذج لشبكية الحركة التي تنتقل عبر طريق تبادل رسائل مع بعضها البعض. كل عملية في شبكة أدوهوك هي ذوي دوري مدخل ومخرج، تحوّل عمليات الأدوهوك أو قد تكون تفاعلات مع عمليات أخرى. بنية هذه الشبكة تحدد كل عملية مجموعة من الجيران المباشرين في الشبكة. عندما ترسل عملية في شبكة أدوهوك رسالة، يتم بث الرسالة إلى جميع جيرانها المباشرين. وبناءً على ذلك، نفترض أن بنية هذه الشبكة ثابتة، أي أن البنية لا تتغير أثناء تشغيل الشبكة. أيضاً، فإن حجم الشبكة الذي يعرف بأنه عدد العمليات في الشبكة، ليس محدد بشكل مسبق. فحص النموذج لشبكة أدوهوك لأي صعوبة بسبب غرب تقريري. السؤال الذي نتناوله في الفصل الرابع، وهو المقال الأول من هذه الأطروحة هو ما إذا كان هناك فئة معينة من شبكات الأدوهوك التي من الممكن التحقق من إمكانية وصول عمليات فيها إلى حالات تعتبر سلبة.

في النوع الثاني من الشبكات التي نعتبرها، الآلات ذاتية التشغيل (Dynamic Register Automata) تستخدم المتغيرات من طرف العمليات لتخزين عنوان العمليات الأخرى في الشبكة. يمكن لعملية (أ) في شبكة من هذا القبيل إرسال رسالة (أو عنوان) إلى عملية أخرى (ب) إذا كانت عملية (ب) موجودة. وعندما تصل إلى ذلك، يمكن لعملية في شبكة من هذا القبيل أن تنشئ عمليات جديدة وأو أن تغير بنية الشبكة. نتيجة لذلك، الشبكات من نوع الآلات الذاتية التشغيل هي ديناميكية ويمكن أن تتطور في البنية والحجم مع مرور الوقت. في هذه الأطروحة، ننظر إلى تقنيات التحقق من مشكلة الوصول إلى حالة لنوع من هذه الشبكات، في المجال الثاني، نعتبر النوع المنتظم. أي ينتمي فيه إرسال رسالة واستقبالها. في المجال الثالث، نعتبر النوع الغير المنتظم، حيث كل عملية مجهزة بصناديق بريد. يمكن الاتصال بين العمليات في هذا النوع غير متزامن لأن الرسائل التي يتم إرسالها يمكن استقبالها في وقت لاحق. نقدم المساهمات لكل المقالتين في الفصل الخامس.
وأخيراً، فإن النموذج الثالث الذي نعتبره هو نموذج أنظمة ذو الرفع السفلي المتعدد (Multi-Pushdown Systems). تستخدم أنظمة ذو الدفع المتعدد صناديق لبرامج المتزامنة (concurrent programs) التي تستخدم متغيرات تتراوح على نطاق معطيات محدودة. ويمكن اعتبار البرنامج المتزامن بمثابة شبكة تتكون من مجموعة متزامنة من البرامج المتسلسلة ذو الاستدعاء الذاتي، وكل واحدة منها مجهز بمكدس. تتواصل البرامج مع بعضها البعض من خلال قراءة وكتابة مجموعة من المتغيرات المشتركة. في المقال السادس، ندرس تقييم فحص النموذج لفئة معينة من البرامج المتزامنة. نبين أيضاً كيفية تحويل برنامج متزامن من هذه الفئة الفرعية إلى برنامج متسلسل. ومن خلال ذلك، نخول لاستخدام أدوات فحص النموذج الموجودة حالياً لبرامج المتسلسلة لتحليل البرامج المتزامنة. الفصل السادس من هذه الأطروحة يعرض بالتفصيل مساهماتنا في هذا الموضوع.
## Contents

Acknowledgements ........................................................................................................................ xi

Sammanfattning ............................................................................................................................ xiii

Résumé en français ......................................................................................................................... xvii

Summary in Arabic ......................................................................................................................... xxi

1 Introduction ................................................................................................................................. 27
   1.1 Background ........................................................................................................................ 27
   1.2 Testing .............................................................................................................................. 30
   1.3 Formal Verification ........................................................................................................... 31
   1.4 Model Checking ................................................................................................................ 33
       1.4.1 Transition systems ................................................................................................. 33
       1.4.2 The specification .................................................................................................... 36
       1.4.3 Infinite-state systems ............................................................................................ 37

2 Summary of the Contributions ................................................................................................. 40
   2.1 Ad-Hoc Networks ............................................................................................................. 41
   2.2 Dynamic Register Automata ......................................................................................... 45
   2.3 Multi Pushdown Systems ................................................................................................. 48

3 Preliminaries ............................................................................................................................... 51

4 Ad Hoc Networks ....................................................................................................................... 57
   4.1 Network Topology and Selective Broadcast ................................................................ 57
   4.2 Syntax and Semantics ..................................................................................................... 59
   4.3 Directed Acyclic Ad-Hoc Networks .............................................................................. 63
       4.3.1 COVER is undecidable for directed acyclic AHN ............................................ 63
       4.3.2 Coverability of bounded depth DAAHN is decidable ........................................ 66
   4.4 Related Works .................................................................................................................. 69

5 Dynamic Register Automata ..................................................................................................... 72
   5.1 Synchronous DRA .......................................................................................................... 73
       5.1.1 Syntax ..................................................................................................................... 73
       5.1.2 Operational semantics ........................................................................................... 74
       5.1.3 State reachability problem ...................................................................................... 78
   5.2 Undecidability of the State Reachability Problem for Synchronous DRA .................. 78
       5.2.1 The general case ..................................................................................................... 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2</td>
<td>Bounded DRA</td>
<td>79</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Strongly bounded DRA</td>
<td>80</td>
</tr>
<tr>
<td>5.3</td>
<td>Degenerative and (Strongly) Safe Synchronous DRA</td>
<td>82</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Definitions and results</td>
<td>83</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Decidability for strongly bounded degenerative DRA</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Buffered Dynamic Register Automata</td>
<td>86</td>
</tr>
<tr>
<td>5.5</td>
<td>State Reachability Problem for Buffered DRA</td>
<td>91</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Undecidability results</td>
<td>92</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Lossy BDRA</td>
<td>95</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Acyclic strongly bounded topology BDRA with bounded buffers</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>Multi-Pushdown Systems</td>
<td>101</td>
</tr>
<tr>
<td>6.1</td>
<td>Preliminaries</td>
<td>102</td>
</tr>
<tr>
<td>6.2</td>
<td>Bounded Budget MPDS</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Bounded Budget State Reachability is PSAPCE-Complete</td>
<td>107</td>
</tr>
<tr>
<td>6.4</td>
<td>LTL Model Checking</td>
<td>113</td>
</tr>
<tr>
<td>6.5</td>
<td>Code-to-Code Translation</td>
<td>116</td>
</tr>
<tr>
<td>6.5.1</td>
<td>The explicit approach</td>
<td>117</td>
</tr>
<tr>
<td>6.5.2</td>
<td>The eager approach</td>
<td>119</td>
</tr>
<tr>
<td>6.5.3</td>
<td>The lazy approach</td>
<td>121</td>
</tr>
<tr>
<td>6.5.4</td>
<td>Budget bounded code-to-code translation</td>
<td>124</td>
</tr>
<tr>
<td>6.5.5</td>
<td>Termination of bounded budget concurrent programs under fairness</td>
<td>130</td>
</tr>
<tr>
<td>6.A</td>
<td>Appendix</td>
<td>132</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and Future Work</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>140</td>
</tr>
</tbody>
</table>
1. Introduction

In this chapter, we present the background and the motivations of the thesis. We start by showing the importance and the difficulty in finding software failures in the light of the evolution of computing over recent decades. Then we describe software testing, a technique widely used by programmers in order to assess the correctness and the safety of programs. Furthermore, we give an overview of formal verification, a set of techniques that aim at achieving a better analysis of programs by using mathematical formalisms. Mainly, we focus on Model Checking, a particular formal verification approach that we use in the contributions presented in this thesis.

1.1 Background

Computer systems are nowadays present almost everywhere, and we are becoming more and more dependent on them. They can be large systems used, for instance, to predict the weather, decode human genetics and analyze financial trends. They can also be small devices used in our daily life to control the temperature in our fridge, fetch emails on a sophisticated watch or simply connect laptops and smartphones to the Internet through a WiFi router. Even equipment that did not use to have any computation power in the past, such as microwave ovens and car keys, are now built with a micro-controller in which some piece of software is running. All these examples show how pervasive the use of computer systems and devices has become.

In parallel with the dramatic increase in our use of computer systems, software applications have become more and more involved to meet an increasingly demanding user need in terms of functionalities and computing power. In order to match this demand, microprocessor speed has been consistently increasing over the years. Till very recently, single-core microprocessors have followed Moore’s law, an empirical law stipulating that the performance of a single microprocessor doubles every eighteen months. This law is not valid anymore due to the physical impossibility of increasing the density of transistors and also due to the non-reasonable energy consumption required by higher performance processors. Thus, single core processors cannot anymore satisfy the increasingly larger computing power demanded by more complex applications. To overcome this limitation, new computing solutions had to be found. In particular, two computing paradigms, concurrent and distributed programming, have been considered. In concurrent and distributed programs,
a computing task is conducted through the collaboration of many sequential programs. More precisely, concurrent programs use multicore processor technologies, in which concurrent programs can run sequential threads on different computing units (cores) while sharing a common main memory. Distributed programming, on the other hand, consists in distributing independent and complex computing tasks over a network of computers and let them collaborate to achieve a common goal. At a higher level, many functionalities provided by networked solutions such as clouds can profit from a significant degree of distribution of the computation task. As a result of this evolution, multicore processor technologies have become the new standard for personal computing while distributed architectures have become essential for heavy computation demanding applications and other inherently distributed and networked solutions.

Many computer systems we use are critical in the sense that any failure can cause severe human and material losses. This is, for instance, the case of computer devices embedded in cars that are in charge of monitoring essential functions such as speed control and braking. It is also the case for medical machines that administrate and monitor patient treatments. At a larger scale, we can also mention computer systems responsible for monitoring air traffic or the ones used to control large energy production and distribution units.

Reports and inquiries from [92] mention a number of significant accidents that have happened in the last decades showing that severe material and human consequences can result from software failures (or bugs). We recall three of these accidents in order to show how costly these bugs can be. The first example concerns a software failure that occurred with Therac-25, an oncology treatment machine. It has been shown that Therac-25 has delivered lethal radiation doses to cancer patients. The investigation conducted to reveal the causes of the fatal treatment has shown that a race condition, a typical sort of faulty behavior that appears in concurrent programs, was partly to blame for these accidents. The second example is a gasoline pipeline rupture that caused the loss of the life of three people and 45 million US dollars of damage. A mechanical accident due to nearby construction triggered this catastrophe. Nevertheless, the system monitoring the pipeline, mostly based on the supervisory control and data acquisition (SCADA) system, failed to report an accurate image of the situation. This prevented an early human intervention that could have mitigated the impact of the accident. The third example is the electricity blackout that occurred in August 2013 in some northeastern American states. This incident could have been prevented if a software fault in the alarm management system had not occurred. These three examples and many others show that we need to prevent software failures from happening in order to ensure the safety of a wide variety of systems including those used in the military, medical machinery and industrial monitoring.

The previous examples show how vulnerable we have become towards computer system failures and how important it is to detect them at an early
stage. However, the complexity of analyzing a program to detect failures and assess its safety and correctness has become harder. In fact, the complexity of program analysis for the purpose of safety and correctness varies depending on the complexity of the program. Some programs might have a small number of possible input combinations they can take and environments in which they can be used. The task of detecting erroneous behaviors and assessing the correctness of these programs can then be achieved in a reasonable amount of time. On the contrary, other programs can be much harder to analyze due to their complexity, to the big number of scenarios in which they can be executed and to the huge or even infinite number of input combinations they can take. The difficulty of finding program errors applies in particular to the class of concurrent and distributed programs. In fact, besides the need of providing concurrent and distributed architectures with algorithmic solutions to leverage the computation power of multicore processors and distributed networks, the drastic increase in the use of these solutions also raises new challenges regarding program analysis. A concurrent or a distributed program can be seen as a collection of sequential programs, or threads, running in parallel and communicating with each other through some means of communication (e.g. message passing or shared memory). Therefore, in order to analyze such programs, it is not sufficient to examine each thread separately from the others. Instead, the analysis of a concurrent or a distributed program should take into account all possible interleavings of its sequential threads, i.e. all possible relative orderings of the actions composing every thread. Indeed, the choice of which action to be executed next in a concurrent or distributed program is not known in advance. Instead, this choice is non-deterministic, i.e. arbitrary. Therefore, the number of possible runs of a concurrent or a distributed program is exponential in the number of threads composing the program. Moreover, it is not easy to reproduce a specific execution of such a program. The lack of reproducibility of program executions is the main reason for the difficulty of reproducing bugs that appear only for some particular relative orderings of the actions. These bugs are called Heisenbugs for their tendency to disappear as soon as an attempt is made to reproduce them [4].

Thus, the complexity of finding bugs can vary dramatically depending on the software application considered. In a wider perspective, the invasive use of computer devices, the higher complexity of software applications and our increasing dependency on them induce a new level of vulnerability for their users. This situation, combined with the gravity of the incidents that a software failure can cause, justifies the need for tools and techniques that could help software engineers to detect them, preferably at an early stage of their development. In fact, it has been shown from a software engineering perspective [69] that the later a bug is detected, the more expensive is the cost to fix it. We present hereafter software testing and formal verification, two approaches used in order to find software failures and to assess the correctness of programs.
1.2 Testing

Software testing is the most used technique by engineers to detect failures and to assess the quality and the correctness of software programs. Testing is often integrated in the software development cycle and is considered as an important sub-discipline of software engineering. The core idea behind software testing consists of writing a list of scenarios or protocols following which the software should be tested. Each such a scenario is called a test case and should mention: (i) the environment in which the program should undergo the test, (ii) the list of actions the tester should perform and the set of successive inputs the software shall take, and finally (iii) the expected correct output of the program according to the software specification.

Test cases can be written based on the program specification. Once the list of test cases has been created, a human tester or an automatic script can perform every test case and check whether the output of the program matches the expected one. If it is the case, then the test is considered to be successful; otherwise, an error has to be reported and eventually fixed.

Different testing approaches are used, we recall here for the sake of illustration two of them. **Black box testing** [5] consists in testing the software towards its specification without any prior knowledge of the source code of the program. On the other hand, **White box testing** [5] is carried out when the source code of the program is known. Besides testing the software against its specification, an auxiliary goal of white box testing approach is to achieve the highest possible coverage of the code.

Although testing is essential to any software development cycle, it has one major drawback. More precisely, testing can detect bugs that occur in the scenarios listed in the test cases, but it does not shed light on the presence of bugs in situations that are not covered by the test cases. Dijkstra puts it very well in one of his observations [30]: “Testing shows the presence, not the absence of bugs”. Observe that the number of hypothetical and distinct scenarios a software might follow can be enormous, if not infinite. This is due to the number of possible combined inputs a software can take and the number of ways the software can be used. Consequently, it is often impossible for testing to cover all possible scenarios in a reasonable amount of time. To make matters worse, the evolution of computing in the recent decades has made the task of discovering software failures even harder. As mentioned earlier, it is indeed much harder to find and reproduce bugs for the case of concurrent and distributed programs than it is for the case of sequential ones. Because of the exponential number of possible runs that concurrent and distributed programs have and the fact that their interleaving semantics make it harder to reproduce bugs, testing can hardly be used to find bugs for this class of programs.

The above mentioned shortcomings concerning testing efficiency in uncovering software failures, particularly for the case of concurrent and distributed programs, make the call for alternative techniques that could achieve a better
coverage of the system under analysis. Ideally, these techniques should pro-
vide us with tools that are able to find and uncover software bugs, even the
ones that are hard to reproduce. Moreover, these techniques should also pro-
vide tools and techniques to formally prove the correctness of programs with
respect to some pre-defined specification. A set of techniques grouped under
the generic name of formal verification techniques try to achieve these goals
by using mathematical formalisms.

1.3 Formal Verification

We give a brief overview of the two main approaches that have been consid-
ered to conduct formal verification of programs.

The first approach is deductive verification. The main principle here is to
let the programmer manually build a mathematical model and a proof of the
program to show that the program is correct or safe. For instance, program-
ners can build the proof by annotating the different blocks of the program
(e.g. functions, methods, etc.) using Hoare triples \{S\}P\{Q\}. Here, P stands
for the annotated program, S for a precondition that program P should satisfy
before it is executed, and Q for a postcondition that program P will satisfy if it
is assumed to terminate after its execution. Thus, the Hoare triple \{S\}P\{Q\}
of a program can be seen as a contract, where the precondition represents what
the program is expecting in terms of input, while the postcondition describes
the output that the program will ensure. Moreover, the programmer can use
loop invariants to annotate iterative structures of the program. Invariants are
necessary to carry out proofs through loop bodies and recursive functions.
Furthermore, the programmer can incrementally build the proof by annotat-
ing the program using assertions. Once the program has been annotated and
the property to be proven asserted, the sanity of the proof can be checked us-
ing theorem provers. It is often the case that the theorem prover, only using
the provided annotations, is not capable of proving the validity of the prop-
erty specified by the programmer. The programmer needs then to refine the
proof by either providing more annotations or by modifying the current ones.
Thus, there could be many refinements until the theorem prover manages to
validate the program proof. Although invariants can sometimes be automat-
ically generated, an input from the programmer is often necessary to prove
the correctness of the program. Thus, in many cases, this technique cannot
be fully-automated, which constitutes the main drawback of the deductive ap-
proach.

The second approach consists of fully automatic techniques, also called
push button techniques. We give here a brief description of two such tech-
niques: Model Checking and Abstract Interpretation.

The approach of abstract interpretation [36] takes as input a (concrete) pro-
gram and builds an abstraction out of it. The idea behind this approach is to
prove the correctness of the concrete program by showing the correctness of its abstraction. More precisely, an abstract domain is defined together with an abstraction and a concretization function. The abstraction function maps concrete states to abstract states, while the concretization function maps abstract states to sets of concrete states. The abstract domain comes with a partial order relation over the set of abstract states that is monotonic with respect to the concretization function. Moreover, an abstract transformer is induced on the abstract domain by consecutively concretizing an abstract state, concretely evaluating its successors with respect to the transitions of the concrete program and finally abstracting the result. Furthermore, widening techniques can be used to accelerate the computation of the fixpoint that characterizes the set of reachable states. Finally, narrowing techniques can be used to make the abstraction more precise and consequently reduce the number of false errors.

The idea of abstracting the concrete program and proving the correctness of the concrete program by proving the correctness of its abstraction is also used by other techniques, such as predicate abstraction [51]. The number of states of a concrete program is in general much bigger than the number of states of its abstraction. Therefore, working on the abstract model rather than on the concrete program helps to handle the big (or sometimes infinite) number of states the concrete program has. In the case of predicate abstraction, the abstraction of the concrete program uses predicates on the program variables to symbolically represent sets of concrete states. Furthermore, the abstract transition is built by allowing abstract transitions between two (abstract) states as soon as a concrete transition exists between two concrete states representing these two abstract states. Predicate abstraction analysis can return false-positives, i.e. spurious errors that do not exist in the concrete program. These false errors can be used to refine the abstraction and make it more precise. This refinement procedure is called the counter example guided abstraction refinement (CEGAR) approach.

While the abstraction of the concrete program is an important part of the abstract interpretation approach, Model Checking [84, 32, 21] approach, on the other hand, is a fully automatic verification technique that directly operates on the abstract model of the program. Model checking is in fact not in charge of the modeling part, i.e. the process of modeling the concrete program into an abstract mathematical formalism. However, the formal analysis of the model is itself fully automatic. Hence, Model Checking techniques can themselves be used by predicate abstraction and other verification techniques in order to conduct the correctness analysis of abstract programs or models.

Since the work presented in this thesis is exclusively based on Model Checking techniques, we give in what follows a more detailed description of this approach.
1.4 Model Checking

As depicted in Figure 1.1, Model-Checking takes two inputs, a model $M$ and a property $P$, also called specification. The goal of Model-Checking is to check whether property $P$ is satisfied by model $M$, i.e. whether property $P$ holds in every possible execution of the modeled system. We use $M \models P$ to denote that $M$ satisfies $P$. Checking whether the model meets its specification should be carried out in a fully automatic manner. The outcome can be either positive or negative, depending on whether the model satisfies the specification or not. In the latter case, a counterexample showing how the model violates the property shall be returned by the Model-Checking tool.

![Figure 1.1. Model-Checking Parts](image)

In the remaining of this section, we first present examples of models in terms of transition systems. Then we describe in more details types and examples of properties considered in Model Checking problems. Finally, we mention the class of infinite-state systems and show the challenge of Model Checking these models.

1.4.1 Transition systems

The model $M$ is a representation of the concrete program using some mathematical model. Table 1.1 enumerates examples of mathematical representations used to model different classes of programs.

<table>
<thead>
<tr>
<th>Mathematical model</th>
<th>Class of programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite-state automata</td>
<td>Non-recursive Boolean programs</td>
</tr>
<tr>
<td>Pushdown automata</td>
<td>Sequential programs with recursive function calls</td>
</tr>
<tr>
<td>Lossy channel systems</td>
<td>Communication protocols</td>
</tr>
<tr>
<td>Petri nets</td>
<td>Lock-based concurrent programs</td>
</tr>
</tbody>
</table>

Table 1.1. Examples of models used to represent different classes of programs.
Each of these models induces a transition system defined as a tuple

\[ (C, C^{\text{init}}, \rightarrow_{\text{trans}}) \]

where:

- \( C \) is the set of configurations of the system,
- \( C^{\text{init}} \) is the set of initial configurations from which the system can start its run, and
- \( \rightarrow_{\text{trans}} \) is the transition relation over the set of configurations \( C \). If two configurations \( c_1, c_2 \in C \) are related through \( \rightarrow_{\text{trans}} \), then we write \( c_1 \rightarrow_{\text{trans}} c_2 \) and say that \( c_2 \) is a successor of \( c_1 \) with respect to \( \rightarrow_{\text{trans}} \). This models the fact that the system may evolve from configuration \( c_1 \) to configuration \( c_2 \).

Given a program and its associated transition system \( (C, C^{\text{init}}, \rightarrow_{\text{trans}}) \), we can formally define and reason about runs of the program. Indeed, a run of the program can be described as a run of the transition system. A run of the transition system is in turn defined as a sequence \( c_0c_1c_2\ldots c_n \in C \) of configurations starting from an initial one \( c_0 \in C^{\text{init}} \), where every configuration \( c_i \) is a successor of \( c_{i-1} \) w.r.t. the transition relation \( \rightarrow_{\text{trans}} \) for \( i : 1 \leq i \leq n \).

Below, we give some examples of transition systems induced by the models of pushdown automata and Petri nets mentioned in Table 1.1.

![Diagram of a pushdown automaton](figure1.2.png)

**Figure 1.2.** Example of a pushdown automaton.

**Pushdown automata.**

A pushdown automaton is used to model a sequential and recursive program, i.e. a sequential program with recursive function calls. A pushdown automaton is a finite-state machine that operates on a stack. A state transition can either be pushing a symbol to the stack, popping a symbol from the stack or a transition that does not involve the stack. Symbols that are pushed and popped from the stack belong to a finite stack alphabet. Therefore, the content of the stack is defined as a finite word over the stack alphabet. Moreover, a configuration of a pushdown automaton is defined as the pair composed of the current state of the finite-state machine together with the content of the stack. Furthermore, an initial configuration of the system can be defined as the pushdown
configuration in which the finite-state machine is in one of its initial states and where the content of the stack is empty. Finally, the transition relation over the set of pushdown configurations is induced by the state transitions listed by the finite-state machine. Figure 1.2 shows an example of a pushdown automaton. It has four possible states $s_0, s_1, s_2$ and $s_3$. The initial state is $s_0$ and the stack alphabet is \{a, b, c\}. The configuration of the pushdown automaton on the left-hand side of Figure 1.2 is $\langle s_1, acba \rangle$. The finite-state machine can move from state $s_1$ to state $s_2$ by pushing symbol $b$ to the stack. This induces a transition from configuration $\langle s_1, acba \rangle$ to configuration $\langle s_2, babca \rangle$. The latter is depicted on the right-hand side of Figure 1.2.

**Petri nets.**

A Petri net is defined as a finite set of places $P$, a finite set of transitions $T$ and a flow relation $F \subseteq (P \times T) \cup (T \times P)$. For instance, the Petri net in Figure 1.3 has four places $C, L, I$ and $cnt$ and two transitions $t_1$ and $t_2$. Every place in a Petri net may contain any number of tokens. A configuration of a Petri net is defined by the number of tokens each place in the Petri net contains. For instance, the configuration of the Petri net on the left-hand side of Figure 1.3 can be described by the tuple $\langle C = 0, L = 1, I = 3, cnt = 2 \rangle$, or simply $\langle 0, 1, 3, 2 \rangle$.

---

**Figure 1.3.** Example of a Petri net.

Observe that the Petri net in Figure 1.3 can be seen as the model of a concurrent program composed of an unbounded number of threads. Every thread in this concurrent program can be either idle or in the critical section. A thread is in the critical section if it holds a lock to some exclusive shared resource. If a thread is not in the critical section, then it is idle. Moreover, the number of tokens in every place can be interpreted as follows: (i) the number of tokens in place $I$ represents the number of idle threads, (ii) the number of tokens in place $C$ represents the number of threads in a critical section, (iii) the number of tokens in place $L$ represents the number of available locks, and (iv) the number of tokens in place $cnt$ represents the number of times the lock has been taken. For instance, configuration $\langle C = 0, L = 1, I = 3, cnt = 2 \rangle$ can be interpreted as “no threads are in the critical section, a lock is currently available, there are three idle threads, and the lock has been used so far two times.”
A Petri net transition \( t \) can be \textit{fired}, only when, for every edge going from a place \( p \) to transition \( t \), there is at least one token in \( p \). Therefore, only transition \( t_1 \) in configuration \( \langle 0, 1, 3, 2 \rangle \) of the Petri net in Figure 1.3 can be fired. If \( t_1 \) is fired, then the new configuration of the Petri is configuration \( \langle 1, 0, 2, 3 \rangle \), depicted in the right-hand side of Figure 1.3. Observe that transition \( t_1 \) models acquiring a lock by an idle thread while transition \( t_2 \) models the release of a lock by a thread in the critical section. Each transition \( t_1 \) and \( t_2 \) induces a transition relation over the set of configurations of this Petri Net. Finally, an initial configuration for this model can be defined as a configuration in which the lock is available and there is an unbounded number of idle threads, i.e. any configuration of the form \( \langle C = 0, L = 1, I = i, cnt = 0 \rangle \) where \( i \geq 0 \) is the number of idle threads present in the system initially.

1.4.2 The specification

The specification or property \( P \) captures the notion of correctness of the model and describes how it is supposed to behave. Different logics have been used in order to express the specification. Such logics vary in power. They can be as simple as propositional logic. They can also be as involved as Monadic Second Order logic (MSO for short) or temporal logics, such as linear and branching time logics (LTL and CTL respectively). Temporal logics allow to causally relate events that happen during runs of the system. In the setting of a client-server application, using this type of logics, we can express the property that every client request is eventually met with an appropriate server answer at some point in time. Such logics can define a wide range of program properties. However, there are two categories of properties that emerge in practice: \textit{safety} and \textit{liveness} properties.

\textit{Safety properties.}
Informally, a system is safe when nothing bad happens. In many cases, this can be translated into the assertion that the system should not end up in a state or in a configuration considered as bad or dangerous. Thus, checking whether a model is safe or not can be reduced to the problem of checking whether the model can reach some bad configuration. The model is considered to be unsafe if a bad configuration is reachable. Otherwise, the system is considered to be safe. Checking whether some given state is reachable is known as the \textit{state reachability} problem and is one of the most basic verification questions to consider when analyzing a new class of systems.

For instance, consider the example of a concurrent program composed of three threads sharing a resource that can be used by at most one thread at a time. A model of the program is depicted by the Petri net in Figure 1.3. This program is considered as safe if there is at most one thread in the critical section at any time. This safety property is called the \textit{mutual exclusion} property.

36
With respect to this property, the situation in which at least two threads are simultaneously granted access to the same shared resource is a bad situation that should be avoided. This situation corresponds to any Petri net configuration where there are at least two tokens in place \( C \). If any of these configurations is reachable, then the system is unsafe. Otherwise, the system is safe.

Interestingly, the Petri net in Figure 1.3 can also model a concurrent program composed of an unbounded number of threads. The safety of such a concurrent program can be expressed through the question whether two or more threads can be in the critical section at the same time. This question amounts to checking whether there is a run of the Petri net that can reach a configuration in which there are at least two tokens in place \( C \), i.e., whether there is a reachable configuration of the form \( \langle c, l, i, cnt \rangle \) such that \( \langle 2, 0, 0, 0 \rangle \sqsubseteq \langle c, l, i, cnt \rangle \). This is an example of the coverability problem for the case of Petri nets. In general, given a transition system \( \mathcal{T} = \langle C, C^{\text{init}}, \rightarrow_{\text{trans}} \rangle \), an ordering \( \sqsubseteq \) over the set of configurations \( C \) and a bad configuration \( c_{\text{bad}} \), checking whether there is a reachable configuration \( c \) in \( \mathcal{T} \) such that \( c_{\text{bad}} \sqsubseteq c \) is called the coverability problem. In the case where the set of all bad configurations is characterized by the set \( \{ c \mid c_{\text{bad}} \sqsubseteq c \} \), proving the safety of the system amounts to checking the coverability of \( c_{\text{bad}} \).

**Liveness properties.**

A **Liveness** property captures whether something good will eventually happen. Consider again the example of a concurrent program composed of threads sharing some critical resource. It can be relevant to know whether every thread will eventually get the opportunity to enter the critical section. Another example of a liveness property is **termination**, which describes programs that terminate in every possible run.

### 1.4.3 Infinite-state systems

A model \( M \) can have an infinite state space if its associated transition system has an infinite set of configurations. In this case, we say that the model is an **infinite-state system** or simply **infinite**. We mention here two reasons for which a model is infinite.

**Data structure.**

A model can be an infinite-state system due to the infinite **data domain** of the data structures used by the model. For instance, a pushdown automaton is an infinite-state system because it is equipped with a stack, an unbounded last-in-first-out (LIFO for short) data structure. Another example is a network composed of a finite number of processes, each of which is equipped with a **buffer** with an unbounded capacity. A buffer is a first-in-first-out (FIFO for short) data structure. As in the case of counter machines, an abstract program
can also be considered as an infinite-state system when it is equipped with variables that range over an unbounded data domain such as real numbers or (unbounded) integers. Finally, a relational database is an infinite-state system if the data domain of the relations is infinite.

**Control structure.**

A model can also have an infinite state space because it may contain an unbounded number of processes. We give here two examples of such systems.

The first example is a network composed of a finite but unbounded number of finite-state processes. The number of processes defines the size of the network. Since this parameter is unbounded, the number of possible networks is infinite. Considering some network property \( P \), an interesting question is to ask whether property \( P \) is satisfied by any network, regardless of its size.

The second example is a concurrent program composed of a finite number of threads that share some resource which should be used by at most one program at a time. Such a program should satisfy the property of mutual exclusion, which states that not more than one thread can be using the shared resource. The Petri net of Figure 1.3 models an instance of such programs. Each instance of the concurrent program is finite, but there are infinitely many possible instances of the concurrent program. An interesting problem is to check whether the mutual exclusion property is satisfied by the concurrent program regardless of the number of threads it contains.

Concurrent programs that share a resource whose access is mutually exclusive, and networks of communicating finite-state processes, are examples of parametrized systems. In both examples, the parameter that defines the size of one instance of the system is the number of processes (or threads). Checking that some property \( P \) holds regardless of the value of that parameter is called parametrized verification. The need to check that a property is correct for any size of the system is typical for protocols designed for any number of processes, which is the case for a number of mutual-exclusion and communication protocols. The correctness of such protocols needs to be checked regardless of the number of processes.

**Decidability of model checking infinite-state systems.**

The Model Checking problem \( M \models P \) is the decision problem which asks whether the model \( M \) satisfies the property \( P \). This problem is decidable if there is an algorithm that is guaranteed to terminate and that can provide an answer to the problem. Otherwise, the problem is undecidable.

Consider a model \( M \) and a property \( P \) which asserts that none of the states reachable by \( M \) belongs to a set of bad states \( F_{\text{bad}} \). In the case where \( M \) has a finite state space, checking whether \( M \) satisfies \( P \) can be automatically achieved by systematically exploring all states reachable by \( M \) and checking whether any of them are in the set \( F_{\text{bad}} \). Hence, model checking \( M \) against \( P \) is decidable in the case where \( M \) has a finite state space. This is not trivial.
anymore when the model has an infinite state space. As an example, consider the two-counter Minsky machine, i.e. a two-counter machine which transitions can increment, decrement or test for zero any of the two counters. The two-counter Minsky machine is an infinite transition system due to the infinite range of the counters values. It is well known that checking state reachability for this model is not decidable [77].

Infinite-state systems are used to model systems from a wide range of application domains such as concurrent programs and database systems. However, as it is shown in the case of the Minsky counter machines, Model-Checking of infinite-state systems is in general undecidable. Therefore, an interesting problem is to find classes of infinite-state systems for which Model-Checking is decidable. Moreover, when the Model-Checking problem is undecidable for a specific class of infinite-state systems, another interesting problem is to define a subclass or an approximation of the system for which the problem is decidable. A subclass of a system is defined by syntactically restricting the definition of the system. An approximation is defined by restricting or relaxing the semantics of the system. Informally, an approximation of a model is another model that mimics the original one by allowing either more or less behaviors.

If an approximation contains more behaviors than the original model, then it is an over-approximation. Every behavior of the original model is also present in the over-approximation. For instance, assume that we manage to prove that every behavior of the over-approximation of some model $M$ satisfies some property $P$. This necessarily implies that every behavior of the original model $M$ also satisfies $P$. On the other hand, if a behavior is found in the over-approximation that does not satisfy $P$, it is not necessarily the case that this behavior is also present in the original model. If $P$ is the state reachability property, over-approximations can be used to show that the original model does not reach a bad state.

An approximation may also generate fewer behaviors, in which case it is called an under-approximation. An under-approximation only contains behaviors that are present in the original model. In that sense, the under-approximation might discard some behaviors from the original model but should not add new ones. Consequently, finding a faulty behavior in the under-approximation implies the existence of the same behavior in the original model. On the other hand, showing the correctness of the under-approximation does not imply the correctness of the original model. In that sense, under-approximations are similar to testing, they can help to find bugs in the original model but cannot show its correctness. An example of an under-approximation in the case of concurrent programs is when we only consider runs of the concurrent program in which the number of context switches is bounded by some predefined value. In the case of recursive programs, one can also consider the under-approximation in which the depth of the stack is bounded.
2. Summary of the Contributions

The goal of this thesis is to extend the applicability of Model Checking for three models of networks of communicating processes: Ad-Hoc Networks, Dynamic Register Automata and Multi pushdown systems.

The first model, Ad-Hoc Networks (AHN for short), models static networks of processes that communicate through message passing. Each process in an AHN is defined as a finite-state automaton with transitions that can model local moves or the communication between the processes. The connectivity between processes in an AHN is modeled by a network topology defined as a graph. The vertices in the network topology graph represent processes while the edges model the communication capabilities between the processes in the network. The communication in an AHN is based on a variant of message passing in which a process can broadcast a message only to its direct neighbors in the topology of the network. Moreover, we assume that the topology of an AHN is arbitrary but static, i.e., the topology does not evolve during the run of the network. Furthermore, the size of the network, defined as the number of processes in the network, is not a priori bounded. In Paper I, we address the decidability of the coverability problem, which asks whether an AHN can reach a configuration in which one of its processes is in an error state. Since the size of the network is arbitrary, the coverability of ad-hoc networks is a parametrized verification problem. We show that the problem is undecidable in the general case and define the class of directed acyclic AHN, in which we assume that the topology of the network is a directed acyclic graph. In Section 2.1, we introduce in more details the model of AHN and our contributions in the verification of directed acyclic AHN. Extensive details concerning the model definition, the verification problems and the decidability results can be found in Chapter 4 of this thesis.

The second model that we consider is the model of Dynamic Register Automata (DRA for short). Processes in a DRA network are finite-state machines equipped with a set of registers. Registers are used by processes to store the identifiers of other processes in the network. The identifier of a process is unique and can be seen as its address in the network. The communication in a DRA network is also carried via a variant of message passing by which a message is sent to at most one process at a time. Dynamic register automata model networks in which processes have the ability to (a) create new processes, and (b) modify the topology of the network. As a consequence, DRA networks are dynamic and can evolve in shape and in size over time, which makes the verification of this type of networks difficult. In Papers II and III,
we address the verification of the state reachability problem for two variants of dynamic register automata. Similar to the coverability problem, the state reachability problem asks whether a DRA network can evolve into a configuration in which at least one of its processes is in a bad state. In Paper II, we consider synchronous DRA networks, in which the communication is rendezvous based. In Paper III, we consider buffered DRA, a variant of the DRA model where every process is equipped with a mailbox and where the communication between processes is asynchronous. In Section 2.2 of this chapter we introduce in more details the two variants of the DRA model and briefly present the decidability results of the state reachability problem for each of these variants. The contributions of both papers are extensively presented in Chapter 5.

Finally, the third model that we consider is the model of Multi pushdown system (MPDS for short). A MPDS is a finite-state automaton that operates on more than one stack. Multi pushdown systems are used to model concurrent programs in which variables range over a finite data domain. We assume that a concurrent program is composed of a finite number of recursive sequential programs (threads) that communicate with each other by reading and writing to a set of shared variables. Concurrent programs that we consider can be seen as networks composed of a finite set of processes, where each process is a thread, and where communication uses a shared memory instead of message passing. Furthermore, a process in a network defined by a MPDS can be modeled with a finite-state automaton equipped with a stack, an unbounded data structure. In Paper 6, we study the complexity of Model-Checking budget bounded MPDS, a class of MPDS where each thread is given a budget of contexts that varies depending on the depth of its stack. A context is a segment of the execution of the concurrent program composed of actions performed by only one thread. To demonstrate the practicality of this class of MPDS, we propose a code-to-code translation that transforms a concurrent program into a sequential program that simulates any budget bounded behavior of the concurrent program for a given context budget and stack depth. In Section 2.3 of this chapter, we briefly introduce the class of budget bounded MPDS, the complexity results of Model-Checking budget bounded MPDS and the code-to-code translation. All these contributions are presented in more details in Chapter 6 of this thesis.

2.1 Ad-Hoc Networks

In this section, we start by presenting wireless ad-hoc and sensor networks, a specific type of autonomous wireless networks. Then, we present and motivate the use of the Ad-Hoc Network model to model the behavior of such networks. We informally define the notions of ad-hoc network configuration and the verification problem we consider for this class of networks of commu-
Communicating processes. Finally, we introduce the class of directed acyclic ad-hoc networks and summarize our decidability results for this class.

*Wireless ad-hoc and sensor networks.*

Wireless Ad-Hoc Networks (WANET for short) are wireless networks designed to provide communication features to the nodes composing the network without the help of a central communication infrastructure. More precisely, a WANET is a wireless network in which network-related tasks, such as data packet routing, are not delegated to a subset of dedicated nodes, such as network routers and switches, but are instead distributed over the nodes of the network. Besides sending and receiving messages, a node in a WANET is also in charge of relaying messages. A clear benefit from having a decentralized network is to be independent from the existing or non-existing telecommunication infrastructure. Moreover, depending on the underlying communication topology, an ad-hoc network might also tolerate more than one point of failure. Therefore, ad-hoc networks are used in situations in which the telecommunication infrastructure is either absent or unreliable.

Wireless Sensor Networks (WSN for short) are also wireless networks in which network tasks are distributed over the network. However, their primary goal is to collect, process and act upon sensed information. This information can, for instance, be physical measures such as temperature and humidity. This data is sensed using appropriate sensors embedded in wireless devices called sensors. The data sensed is usually collected and sent back to a subset of nodes in the network called sinks. A sink is in charge of processing the information collected and deciding on some actions to carry if any. Therefore, nodes in a WSN can be grouped into three categories: (i) sensors, sensing some information, (ii) sinks, collecting and processing the received information, and (iii) actuator nodes in charge of executing actions decided by the sinks. WSN are most often used to monitor the evolution of a natural, domestic or industrial environment in which information has to be sensed in different, often distant, and sometimes hardly accessible locations. For instance, in the case of the heating system of some building, WSN can be used to sense, monitor and precisely regulate the temperature in each room of the building.

*Motivations behind the AHN model.*

A number of contributions [46, 89, 43, 50, 72, 74, 88, 87, 39] have considered to model WANET and WSN as a network of processes communicating through message passing. These contributions use the notion of a finite-state process to model a node in a WANET or WSN. Moreover, they assume that a process has a finite number of local, message sending and message receiving transitions. Furthermore, the network in these models is often represented with a graph in which vertices correspond to the nodes in the network and edges correspond to the communication links between nodes. A communication link between two nodes represents the connectivity between the corresponding nodes in the
wireless network, i.e. it represents the communication capability induced by
the wireless medium of the network. Thus, an edge between a vertex \( v \) and
another vertex \( u \) models the fact that the process represented by \( u \) can “hear”
or receive a message from the process represented by \( v \). In this case, we say
that \( u \) is in the vicinity of \( v \) or, alternatively, that \( u \) is a direct neighbor of \( v \).
Hence, the graph of the network, also called topology of the network, defines
for each process \( p \) a set of direct neighbors, i.e. the set of processes in the
network that are in the vicinity of \( p \) and that can receive messages from \( p \). This
model induces the notion of local connectivity, where a node can communicate
directly only with its direct neighbors.

In order to take into account this idea of local connectivity, the notion of
selective broadcast has been considered. Informally, selective broadcast states
that a message sent by a process can only be heard by processes in its vicin-
ity. More specifically, selective broadcast assumes that when some process \( p \)
sends a message, the message can only be received by the direct neighbors of
the process \( p \). Moreover, a message that is sent by \( p \) can concretely be received
by its direct neighbor \( q \) only if \( q \) is in a state where a receiving transition cor-
responding to the same message is enabled. Concretely, when a process sends
a message, only the neighbors that are willing to receive that particular mes-
sage will receive it. The remaining neighbors will simply ignore the message.
Subsequently, the message can simply be lost if the sending process has no
neighbors or if none of its neighbors is in a state in which it can receive that
particular message.

Notice that the communication link between two nodes in a wireless net-
work is not necessarily symmetric. This is due to the fact that the wireless
capability of the nodes in the network varies. The model of AHN that we con-
sider in this thesis takes into account the possible asymmetry of the radio link
by using a directed graph instead of an undirected graph as it is considered
in [39]. Moreover, since symmetric (or undirected) graphs are one particular
class of directed graphs, our choice does not limit the power of the model.
Finally, we assume that the connectivity graph of the network is static, i.e.
meaning that the graph will not change during a run of the system. A static
graph topology implies that the connectivity between processes composing
the network does not evolve over time. Although inadequate for the general
class of mobile WANET, this condition is well-suited when nodes forming the
network are stationary and when the radio channel is free from radio interfer-
ences.

Ad-Hoc network processes and configurations.
An Ad-Hoc Network is defined by a pair \( \langle P, G \rangle \) composed of a process \( P \) and
a graph \( G \). Graph \( G \) represents the topology of the network. Process \( P \) is a finite-
state automaton which runs in every vertex of the graph \( G \). Process \( P \) has a
finite number of states, an initial state and a finite number of state transitions.
Each transition can be either an internal transition, a sending transition or a receiving transition.

An internal transition only changes the state of the process that performs the transition. When a process performs such a transition, we say that the ad-hoc network has performed a local transition. A sending transition, on the other hand, might impact the state of a neighboring process. In fact, if some process $p$ is broadcasting a message $m$, then this message will be received by every process $q$ such that (i) process $q$ is in the vicinity of process $p$, and (ii) $q$ is in a state where a receiving transition matching message $m$ is enabled. The sending and the receiving transitions happen simultaneously. Such an action is called a broadcast transition.

A configuration of an ad-hoc network $\langle P, G \rangle$ is defined as a mapping between the set of vertices of the graph $G$ and the set of states of process $P$. A configuration is initial if every process is in the initial state. We define a run of the ad-hoc network $\langle P, G \rangle$ as a succession of configurations, starting from an initial one, and such that each configuration in the run is obtained from the previous one after the execution of an internal or a broadcast transition.

**The coverability problem.**

An interesting problem for ad-hoc networks is to see how the network of processes evolves over time and whether, for instance, it can reach a configuration in which one of the processes is in a state considered as bad. This problem is called the coverability problem (COVER for short) for ad-hoc networks. An instance of this problem is given by (i) the definition of the process $P$, and (ii) a state $q_{bad}$ of process $P$. The verification problem is then formulated as follows: Is there a possible graph $G$ such that the ad-hoc network $\langle P, G \rangle$ can reach a configuration in which one of the processes is in state $q_{bad}$?

Notice that the analysis for the coverability problem should be carried for every possible initial configuration corresponding to every possible graph topology $G$. Moreover, although the number of possible graph topologies for a given number of processes is finite, the number of graph topologies the network can have in general is unbounded and so is the size of the network regarding the number of nodes. As a consequence, the COVER problem is a parametrized verification problem, in the sense that we are interested in the answer to the problem for every possible graph and for every possible size of the network.

**Directed acyclic ad-hoc networks.**

Delzanno et al. have shown in [39] that the coverability problem is undecidable for ad-hoc networks where the topology is an arbitrary (unbounded) undirected graph. This negative result also applies to the class of AHN where the topology is a directed graph. Therefore, we consider in Paper I the class of ad-hoc networks where the graph topology is acyclic. We motivate this constraint by considering that in WSN, data is either collected in an upstream manner,
from the sensors up to the sinks, or reversibly flood downstream from the sink towards the sensors when, for instance, an information request is sent. Although the directed acyclic class of AHN is a strict subclass of the general directed AHN model, we show that the coverability problem for this subclass remains undecidable. Therefore, we examine the class of directed acyclic AHN in which we only consider networks which have a bounded depth. Using a non-trivial instantiation of the Well-Structured Transition Systems framework [6], we show that the state reachability problem for this particular subclass is decidable.

2.2 Dynamic Register Automata.

The model of a Dynamic Register Automaton (DRA for short) has been proposed in [23, 24] as an extension of the classical register automaton for two purposes. The first purpose is to model networks in which processes have the ability of spawning new processes and where the topology of the network and the number of processes may evolve over time. The second purpose is to propose an implementation model for Dynamic Message Sequence Charts [23, 24]. In our work, we use the DRA model for the first purpose, i.e. we use DRA as a model for processes communicating in a network whose topology and size are dynamic, and where the communication is carried through message passing.

In the following paragraphs, we first introduce the DRA model, explain how message passing is performed in a network of DRA processes and why a DRA network is dynamic. Then, we define the state reachability problem for a DRA process and mention its undecidability in the general case. Later, we introduce the classes of strongly-bounded degenerative DRA and strongly safe DRA. Finally, we introduce the model of buffered DRA, an asynchronous variant of the DRA model, and mention our findings with regard to the decidability of the state reachability problem for this model.

Dynamic Register Automaton.

A DRA network is composed of a finite number of DRA processes. We assume that every process in the network has a unique identifier (ID for short) that serves as its address in the network. A DRA process is a finite-state automaton equipped with a finite set of registers. Registers of a DRA process are used to store identifiers of other processes in the network. A DRA has also a finite number of state transitions. A DRA process can perform a local transition, send and receive a message or an ID and create a new process.

Communication in a DRA network is rendezvous based, i.e. it is synchronous and happens between two processes. In order to address a message to one particular process, the sender needs to have the ID of the receiver stored in one of its registers. Informally, we say that a process $p$ knows another pro-
cess $q$ if a register of $p$ contains the ID of $q$. In that case, $p$ can send a message to process $q$. Thus, the content of the registers of the processes in the network defines the connectivity graph of the network, i.e. it defines which process can directly communicate with which other process.

Messages exchanged between processes can either belong to a finite set of messages or can be a process ID. In the later case, the ID can either be the ID of the sender or a copy of the content of one of its registers.

*Dynamic Topology.*

An important feature in a DRA process is process creation, i.e. a DRA process can create (or *spawn*) a new process. We assume that the newly created process gets a *fresh* identifier, i.e. an ID that is not used by any other process in the network. Moreover, when a process $p$ creates a new process $q$, the ID of $p$ is copied to one of $q$ registers while the ID of $q$ is copied to one of $p$ registers. Observe that the topology of a DRA network is dynamic due to three reasons: (i) New processes can be created, (ii) messages exchanged between processes may contain process IDs, and (iii) the topology of the network is based on the knowledge every process has of other processes IDs.

Furthermore, in order to study the evolution of the network, we assume that any run of the network starts from an initial configuration. An initial configuration is simply defined as one single process with empty registers in the initial state of the DRA. Due to the ability of processes to create new processes, the size of a DRA network may increase over time. Therefore, DRA networks belong to the family of infinite-state systems.

*Undecidability of the state reachability problem.*

Unsurprisingly, DRA networks are Turing powerful. Indeed, the most basic verification problems, such as the state reachability problem, are undecidable. Note that the state reachability problem for a DRA network asks whether there exists a run of the network that can reach a network configuration in which one of the processes is in some particular state. Inspired by the study of Ad-Hoc Networks in [39, 40, 10], we considered the case where we restrict the study of the state reachability problem to runs in which the graph topology of each network configuration is *(strongly) bounded* by some predefined value $k$. A graph is $k$ bounded if the length of every simple path in the graph is smaller than $k$. A graph is $k$ strongly bounded if the graph obtained after removing the directions from the edges is $k$ bounded. We show that the state reachability problem is undecidable in both cases.

*Degenerative DRA.*

A degenerative DRA is a DRA process capable of resetting any of its registers at any of its states. We define the *degenerative counterpart* of a DRA process as the same process in which a reset transition is added to every state and for
every register of the process. Interestingly, we show that a DRA and its degenerative counterpart reach the exact same set of states. Even more interesting, we show that the set of states that can be reached by a DRA in runs where only transitions to $k$ strongly bounded configurations are allowed is a subset of the states that can be reached by its degenerative counterpart under the same constraint. This makes the subclass of strongly bounded degenerative DRA an interesting candidate for the study of the state reachability problem. We show in Paper II that the state reachability problem for this particular subclass is decidable but non-primitive recursive. As a corollary, we also show the decidability of the state reachability problem for simply bounded degenerative DRA with one register.

(Strongly) Safe DRA.
Motivated by these positive results, we consider the class of DRA for which every run only generates $k$ (strongly) bounded network configurations for some $k \geq 1$. We call this class (strongly) safe DRA. Based on (i) the observation that a DRA and its degenerative counterpart reach the same set of states, and on (ii) the decidability result we obtained for the subclass of strongly bounded degenerative DRA, we show that the state reachability problem for strongly safe DRA is also decidable.

Buffered Dynamic Register Automata.
The communication in a DRA network is synchronous. Nevertheless, the message passing communication considered in the automata model defined in [23, 24], and from which the DRA model was inspired, is asynchronous. In order to study DRA networks in which processes communicate in an asynchronous manner, we extended in Paper III the model of dynamic register automata networks by equipping every process with a buffer. The buffer of a process is used to store messages sent to the process. Messages stored in the buffer can be read by the process following the first-in-first-out (FIFO for short) policy. We assume that the capacity of the buffer is unbounded. We use Buffered Dynamic Register Automata (BDRA for short) to denote the asynchronous variant of the DRA model.

It is well-known that most of the interesting verification problems are undecidable even for finitely many processes communicating through unbounded perfect FIFO buffers [28]. This result also applies to the case of buffered DRA networks. In order to find an interesting subclass of buffered DRA for which the state reachability problem becomes decidable, we investigate different directions including bounding the capacity of the buffer and bounding the length of the simple paths of the (underlying undirected) graph topology of the network. It turns out that even when we combine these two restrictions, i.e. bounding the capacity of the buffer and (strongly) bounding the topology of the network, the state reachability problem remains undecidable. Nonetheless, we delimit two subclasses of buffered DRA networks for which we prove the
state reachability problem to be decidable. First, we consider lossy buffered DRA networks in which every process can disconnect itself from the network in a non-deterministic manner. We show that the state reachability problem is decidable for bounded-buffer, strongly-bounded and lossy BDRA networks. The second subclass concerns BDRA networks where we bound the size of the buffers, disallow cycles and bound the simple path in the communication graph. We show that the number of possible shapes of the network in this case is finite, which makes the state reachability problem decidable.

2.3 Multi Pushdown Systems

We introduce in the following paragraphs our work on the verification of MPDS. First, we introduce the model of MPDS and describe the class of concurrent programs we consider. Second, we describe two approaches that have been considered in order to define under-approximations of MPDS for which some verification becomes possible. Third, we introduce our budget-bounded approach and present our results in the complexity of the state reachability and the LTL model-checking problems. Finally, we mention the concurrent-to-sequential code-to-code translation scheme that we propose in order to leverage the power of existing sequential program verifiers.

A model for concurrent programs.

A Multi Pushdown System is a finite-state automaton that operates on several stacks. A MPDS models a concurrent program whose variables range over a finite data domain. We assume that a concurrent program is composed of a finite number of processes that communicate through the means of a shared memory. Each process models a thread defined as a sequential and recursive program, i.e. a sequential program that has recursive procedure calls. A shared memory is a set of variables that are accessed in reading and writing by every thread.

Actions performed by a thread can have a local impact. This is the case when a thread reads or writes from or to one of its local variables or when it makes a procedure call. The action of a thread is global when a thread writes to a shared variable. A global action may impact the subsequent state of other threads composing the concurrent program.

We assume that the interleaving semantics applies to concurrent programs. These semantics have two main aspects. The first aspect is the atomicity of every thread action. An action is atomic if its execution cannot be interrupted by another action. A run of the concurrent program is defined by the sequence of actions executed by the threads. The second aspect is the order following which actions are executed. This order is non-deterministic, i.e. the order is not known in advance. The only order that prevails within a run of the concurrent program is the order of each sequential program. Actions of a
thread are executed in a sequential fashion, while the inter-thread order of actions in the run of the concurrent program is arbitrary.

**Bounding the number of contexts or bounding the stack depth.**
The state reachability problem of concurrent programs composed of at least two sequential recursive threads is undecidable. In fact, we can reduce the halting problem of a two-counters Minsky machine into the state reachability of a concurrent program by using the stacks to encode the counters. The complexity of the analysis of concurrent programs motivates the need for defining under-approximations and subclasses of MPDS for which verification problems become decidable. We mention here two such under-approximations in which the analysis of the concurrent program is performed up to a certain bound of some parameter of the system.

The first approach explores runs of the concurrent program in which the number of contexts is bounded. A context is defined as a segment of the run of the concurrent program in which only one of the threads is active. A context switch happens when a thread, currently executing its actions, pauses and another thread resumes the execution of its actions. The context-bounded model-checking approach was proposed by Qadeer and Rehof in [82]. Their approach consisted in only considering runs of the concurrent program that can be divided into a number of contexts bounded by some constant value $k$. It has been shown that the reachability problem for the context-bounded subclass of MPDS is NP-complete. This result holds even in the presence of recursive procedure calls, which may generate an unbounded number of configurations of the system. This approach is empirically justified since it has been shown in [79] that many concurrency errors appear in runs of the system with a few number of contexts.

The second approach considers runs of the concurrent program in which the depth of the stacks is bounded. The size of the stack of a recursive process illustrates how many nested procedure calls the process has performed. It is well-known that the state reachability problem becomes PSPACE-complete when a bound is put on the maximal depth of the stacks. This approach explores a finite number of reachable configurations and is, therefore, more suitable for the detection of shallow bugs.

The **Budgeted Multi Pushdown Systems** approach proposed in Paper IV consists in combining these two approaches. The reason behind combining these two approaches it to permit a larger exploration of the executions of the concurrent program.

**Budget-bounded under-approximation.**
The budget-bounded approach consists in allowing to each thread in the concurrent program a budget of contexts that varies depending on the depth of the stack.
The budget is defined as a pair \((k, d)\). Depending on the current stack depth, each thread is allowed to run in two possible modes. The first mode is active when the stack depth of the thread is smaller than \(d\), while the second mode is active when the stack depth goes strictly above that value. In the first mode, a thread is allowed to perform an unbounded number of context switches. As soon as the stack depth of the thread goes above \(d\), the second mode is activated and the thread is allowed to run at most \(k\) consecutive contexts. Observe that within a context, a process is allowed to use the stack as extensively as it wishes. Moreover, as soon as the depth of the stack goes beyond \(d\), the first mode is again reactivated and the thread may again perform an unbounded number of context switches.

We consider two verification problems for budget-bounded MPDS. The first problem is the control-state reachability problem which consists in checking whether a given control state of a thread can be reached in a budget-bounded run. We show that the problem is \(\text{PSAPCE}\)-complete. The second problem is the LTL Model-Checking problem of budget-bounded MPDS. We show that the problem is \(\text{EXPTIME}\)-complete.

**Code-to-code translation.**

We propose a code-to-code translation that builds a sequential program out of the concurrent program. The program transformation from concurrent to sequential is such that we can reduce the state reachability problem for the budget bounded concurrent program to the state reachability of the sequential program. Moreover, the code-to-code translation that we propose preserves the data domain of the variables of the original program which allows to use it on concurrent programs with any type of data range. The idea behind this translation is to leverage the power of existing sequential program verifiers in order to analyze concurrent programs. To show its use in practice, we have implemented the code-to-code translation into a prototype which automatically transforms input concurrent programs into sequential programs. We fed the resulting sequential programs to three back-end tools (MOPED [44], ES-BMC [35] and CBMC [34]). We compared our approach to two concurrent program verifiers, namely ESBMC and POIROT [64]. Our experiments show that bugs appear for small budgets. Finally, we give an extension to the code-to-code translation in order to check program termination under three fairness conditions. To check termination, we use SPIN model checker [58] as a back-end tool.
3. Preliminaries

In this chapter, we give definitions that are used throughout the thesis. In particular, we define words, multisets, orders, graphs, induced and embedded subgraph relations, finite-state automata and transducers. We also define the \textsc{Transd} problem, an undecidable decision problem used in Papers I, II, and III and in Chapters 4 and 5. Later, we recall the definition of a Minsky Machine and its associated undecidable halting problem. Finally, we introduce the framework of Well-Structured Transition Systems.

Words, multisets, (well-quasi) orderings and functions

We use $\mathbb{N}$ (respectively $\mathbb{N}_{>0}$) to denote the set of natural (respectively positive) numbers. Given a finite set $A$, we use $|A|$ to denote the number of elements in $A$. We use $A^\circ$ to denote the set of finite multisets over $A$. Sometimes, we write multisets as lists, e.g., $[a, a, b, b, b]$ is a multiset with two occurrences of $a$ and three occurrences of $b$. We use $A^*$ (resp. $A^\omega$) to denote the set of finite (resp. infinite) words over $A$. For words $w_1, w_2 \in A^*$ we use $w_1 \cdot w_2$ or simply $w_1 w_2$ to denote the concatenation of $w_1$ and $w_2$. The length of a word $w \in A^*$ is denoted by $|w|$. We assume that $|\varepsilon| = 0$ and $|w| = \omega$ if $w \in \Sigma^\omega$. For $k \in \mathbb{N}$, we use $A^\preceq_k$ to denote the set of all words $w \in A^*$ such that $|w| \leq k$.

A quasi-ordering (ordering for short) $\sqsubseteq$ on a set $A$ is a reflexive and transitive binary relation over $A$ (i.e. $\sqsubseteq \subseteq A \times A$, $a \sqsubseteq a$ for any $a \in A$ and $a \sqsubseteq c$ for every $a, b, c \in A$ such that $a \sqsubseteq b$ and $b \sqsubseteq c$). We extend the ordering $\sqsubseteq$ on $A$ to the ordering $\sqsubseteq^\circ$ on the set $A^\circ$ of multisets over $A$ such that $[a_1, \ldots, a_m] \sqsubseteq^\circ [b_1, \ldots, b_n]$ if there is an injection $h : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ with $a_i \sqsubseteq b_{h(i)}$ for all $i : 1 \leq i \leq m$.

An infinite sequence $a_1, a_2, \ldots$ of elements in $A$ is \textit{good} with respect to an ordering $\sqsubseteq$ if there are $0 \leq i < j$ such that $a_i \sqsubseteq a_j$, otherwise the sequence is bad. We say that $\sqsubseteq$ is a well-quasi ordering over $A$ if all infinite sequences of elements from $A$ are good.

Given a function $f : A \to \mathbb{N}$, we define $\max(f) := \max\{f(e) | e \in A\}$ to be the largest value taken by $f$ over $A$ if it exists, otherwise $\max(f) := \omega$. For a function $f : A \to B$ (respectively partial function $f : A \to B$), we use $f[a \leftarrow b]$ to denote the function (resp. partial function) $f'$ such that $f'(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$. 

51
Graphs and graph orderings

**Graphs.** A (directed) graph is a pair \( G = (V, E) \) where \( V \) is a finite set of vertices and \( E \subseteq V \times V \) is the set of edges.

A (directed) labeled graph \( G \) is a tuple \( (V, \Sigma_v, \Sigma_e, \lambda, E) \) where \( V \) is a finite set of vertices, \( \Sigma_v \) is a set of vertex labels, \( \Sigma_e \) is a set of edge labels, \( \lambda : V \to \Sigma_v \) is the vertex labeling function, and \( E \subseteq V \times \Sigma_e \times V \) is the set of labeled edges.

For a graph \( G = (V, E) \) (respectively labeled graph \( G = (V, \Sigma_v, \Sigma_e, \lambda, E) \)), we define its transpose to be the graph \( G^T := (V, E^T) \) (respectively \( G^T = (V, \Sigma_v, \Sigma_e, \lambda, E^T) \)) defined by \( E^T := \{ \langle u, v \rangle \mid \langle v, u \rangle \in E \} \) (respectively \( E^T := \{ \langle u, l, v \rangle \mid \langle v, l, u \rangle \in E \} \)), i.e. the graph \( G \) in which all edges have been reversed. \( G \) is said undirected or symmetric if \( E^T = E \).

Given a graph \( G \) with its set of edges \( E \), we use closure\( (G) \) to denote the underlying undirected graph of \( G \), where the set of edges \( E' \) is defined by \( E' = E \cup E^T \).

In the remaining of this section, we assume that \( G = (V, E) \) is a directed graph. Definitions can be extended in a natural manner to labeled graphs.

**Paths, successors and predecessors.** For vertices \( u, v \in V \), we use \( u \leadsto_G v \) to denote that \( \langle u, v \rangle \in E \), use \( ^* \) to denote the reflexive transitive closure of \( \leadsto \), and use \( \sim_G \) to denote the transitive closure of \( \leadsto \). A path in \( G \) is defined as a sequence \( \pi = v_1 v_2 \ldots v_k \) where \( v_i \leadsto_G v_{i+1} \) for every \( i : 1 \leq i < k \). The path \( \pi \) is simple if the vertices in the path are distinct, i.e. if \( v_i \neq v_j \) for every \( i, j : 1 \leq i < j \leq k \). In that case we define length\( (\pi) := k - 1 \) to be the length of the simple path \( \pi \).

For a vertex \( v \in V \), we define succ\( G (v) := \{ u \mid v \leadsto_G u \} \) to be its set of successor vertices, and define pred\( G (v) := \{ u \mid u \leadsto_G v \} \) to be its set of predecessor vertices.

We define the width of \( G \), denoted by \( \mathcal{W}(G) \), to be the largest \( k \) such that there is a simple path \( \pi \) in \( G \) with length\( (\pi) = k \).

**(Strongly) bounded graphs.** Let \( k \in \mathbb{N} \) be a natural number and \( G \) a graph. We say that \( G \) is \( k \)-bounded if the length of every simple path in the graph is bounded by \( k \). Furthermore, we say that \( G \) is \( k \)-strongly bounded if its underlying undirected graph is \( k \)-bounded.

**Directed acyclic graphs.** We say that \( G \) is a directed acyclic graph (DAG for short) if there are no cycles in \( G \), i.e there are no vertices \( v \in V \) such that \( v \sim_G v \).

For a vertex \( v \in V \), we define height\( G (v) := 0 \) if succ\( G (v) = \emptyset \), and define height\( G (v) := 1 + \max_{u \in \text{succ}(v)} \text{height}(G(u)) \) otherwise. We define height\( (G) := \max_{v \in V} \text{height}(G(v)) \). height\( (G) \) corresponds to the length of the longest simple path in \( G \).

We define depth\( G (v) := 0 \) if pred\( G (v) = \emptyset \), and depth\( G (v) := 1 + \max_{u \in \text{pred}(v)} \text{depth}(G(u)) \) otherwise. We define depth\( (G) := \max_{v \in V} \text{depth}(G(v)) \), which also corresponds to the length of the longest simple path in \( G \).
A leaf of $G$ is a vertex with no successors.

**Forests and trees.** A forest is a DAG where every distinct pair of vertices $u,v \in V$ is such that $\text{succ}_G(u) \cap \text{succ}_G(v) = \emptyset$. A tree is a forest such that $|\{v \mid \text{pred}_G(v) = \emptyset\}| = 1$. We say that $G$ is an inverted forest (respectively tree) if $G^T$ is a forest (respectively a tree). The root of an inverted tree is the unique vertex $v$ such that $\text{succ}_G(v) = \emptyset$ while a leaf $v$ of an inverted tree is such that $\text{pred}_G(v) = \emptyset$.

**Graph relations.** We define in the following two binary relations over the set of directed graphs, the definition are extended in the expected manner to directed labeled graphs.

Let $G_1 = \langle E_1, V_1 \rangle$ and $G_2 = \langle E_2, V_2 \rangle$ be two graphs.

The embedded subgraph relation. We say that $G_1$ is an embedded subgraph of $G_2$ and we write $G_1 \subseteq_{\text{sub}} G_2$ if there exist an injection $f : V_1 \rightarrow V_2$ such that $u \sim_{G_1} v \mapsto f(u) \sim_{G_2} f(v)$ for every $u,v \in V_1$.

The induced subgraph relation. We say that $G_1$ is an induced subgraph of $G_2$ and we write $G_1 \subseteq_{\text{ind}} G_2$ if there exist an injection $f : V_1 \rightarrow V_2$ such that $u \sim_{G_1} v \iff f(u) \sim_{G_2} f(v)$ for every $u,v \in V_1$.

Finite-state automata, transducers and TRANSD

We recall the definitions of finite-state automata and transducers, and define the TRANSD problem.

**Finite-state automaton.** A finite-state automaton is a tuple

$$A = \langle Q, \Sigma, \rightarrow_{\text{trans}}, q_{\text{init}}, Q_{\text{final}} \rangle$$

where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $q_{\text{init}} \in Q$ is the initial state, $Q_{\text{final}} \subseteq Q$ is the set of final states, and $\rightarrow_{\text{trans}} \subseteq Q \times \Sigma \times Q$ is the transition relation. We define the language $L(A)$ of $A$ as usual.

**Finite-state transducer.** A finite-state transducer is a tuple

$$T = \langle Q, \Sigma, \rightarrow_{\text{trans}}, q_{\text{init}}, Q_{\text{final}} \rangle$$

where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $q_{\text{init}} \in Q$ is the initial state, $Q_{\text{final}} \subseteq Q$ is the set of final states, and $\rightarrow_{\text{trans}} \subseteq Q \times \Sigma \times Q$ is the transition relation. A member of $L(T)$ is a word of pairs over $\Sigma$, i.e., a member of $(\Sigma^2)^*$. A transducer $T$ induces a binary relation $R(T)$ on the set $\Sigma^*$ such that $\langle a_1 \cdots a_n, b_1 \cdots b_n \rangle \in R(T)$ if $\langle a_1, b_1 \rangle \cdots \langle a_n, b_n \rangle \in L(T)$. For a word $w \in \Sigma^*$, we define $T(w) := \{v \mid \langle w, v \rangle \in R(T)\}$. For a language $L \subseteq \Sigma^*$, we define $T(L) := \cup_{w \in L} T(w)$. Given an automaton $A$ and a transducer $T$ (with identical alphabets $\Sigma$) we define $T(A) := T(L(A))$. For a natural number $i \in \mathbb{N}$ and a word $w \in M^*$, we define $T_i(w)$ inductively by $T_0(w) := \{w\}$ and $T_{i+1}(w) := T(T_i(w))$. In other words, it is the result of $i$ applications of the relation induced by $T$ on $w$. We extend the definition of $T_i$ to a language in the expected manner. Finally, for an automaton $A$, we define $T_i(A) := T_i(L(A))$. 53
The **TRANSD problem.** An instance of the TRANSD problem consists of two finite-state automata $A$ and $B$, and a transducer $T$, all with identical alphabets $\Sigma$. The task is to check whether there is an $i \in \mathbb{N}$ such that $T^i(A) \cap L(B) \neq \emptyset$. We show in Paper I that the TRANSD problem is undecidable.

Minsky machine

A Minsky 2-counter machine is a finite-state machine equipped with two counters $c_1$ and $c_2$ which range over the set of natural numbers. A Minsky machine has a finite number of transitions which can either increment or decrement one of the counters, or check if the value of one of the counters is zero. Formally, a Minsky 2-counter machine $\mathcal{M}$ is a tuple $\langle Q, q_{\text{init}}, q_{\text{halt}}, \Theta \rangle$ where $Q$ is a finite set of control states, $q_{\text{init}}$ is the initial state, $q_{\text{halt}}$ is the halting state, and $\Theta$ is a finite set of transitions. Each transition can have one of the following forms:

1. Increment of counter $c_i$: $\langle q, c_i++, q' \rangle$, $i \in \{1, 2\}$, $q, q' \in Q$.
2. Decrement of counter $c_i$: $\langle q, c_i--, q' \rangle$, $i \in \{1, 2\}$, $q, q' \in Q$.
3. Test for zero of counter $c_i$: $\langle q, \text{zerotest}(c_i), q' \rangle$, $i \in \{1, 2\}$, $q, q' \in Q$.

A configuration of $\mathcal{M}$ is an element of $Q \times \mathbb{N}^2$. The initial configuration is $\langle q_{\text{init}}, (0, 0) \rangle$. The counter machine moves from a configuration $\langle q, (m_1, m_2) \rangle$ to a configuration $\langle q', (m'_1, m'_2) \rangle$ in one step, written as $\langle q, (m_1, m_2) \rangle \rightarrow_M \langle q', (m'_1, m'_2) \rangle$, if one of the following cases holds:

1. $\langle q, c_i++, q' \rangle \in \Theta$, $i \in \{1, 2\}$, $m'_i = m_i + 1$ and $m'_{3-i} = m_{3-i}$
2. $\langle q, c_i--, q' \rangle \in \Theta$, $i \in \{1, 2\}$, $m_i > 0$, $m'_i = m_i - 1$ and $m'_{3-i} = m_{3-i}$.
3. $\langle q, \text{zerotest}(c_i), q' \rangle \in \Theta$, $i \in \{1, 2\}$, $m'_i = m_i = 0$ and $m'_{3-i} = m_{3-i}$.

A run of $\mathcal{M}$ is a finite sequence of configurations of the form $c_0 c_1 c_2 \cdots c_m$ such that: (1) $c_0 = \langle q_{\text{init}}, (0, 0) \rangle$, and (2) $c_i \rightarrow_M c_{i+1}$ for all $i : 0 \leq i < m$. The reachability problem for $\mathcal{M}$ asks whether there is a run $c_0 c_1 c_2 \cdots c_m$ of $\mathcal{M}$ such that $c_m = \langle q_{\text{halt}}, (0, 0) \rangle$. It has been shown in [77] that the state reachability problem for Minsky 2-counter machines is undecidable.

(Well-structured) transition systems

Here, we (i) recall the definition of a transition system, (ii) define the notion of monotonicity of a transition relation with respect to an ordering, (iii) define a well-structured transition system, (iv) recall the definition of the coverability problem, and (v) recall a procedure from [6] that decides the coverability problem, assuming that a number of conditions are provided.

**Transition system.** As mentioned in Subsection 1.4.1 of Chapter 1, a transition system is defined as a tuple $\mathcal{T} = \langle C, C^{\text{init}}, \rightarrow \rangle$ where $C$ is the set of configurations of the system, $C^{\text{init}}$ is the set of initial configurations from which the system can start its run, and $\rightarrow$ is the transition relation over the set of
configurations $C$. If two configurations $c_1, c_2 \in C$ are related by $\rightarrow$, then we write $c_1 \rightarrow c_2$ and say that $c_2$ is a successor of $c_1$ with respect to $\rightarrow$. A (finite) run of $T$ is a sequence of configurations $c_0 c_1 c_2 \ldots c_n$ that starts from an initial configuration $c_0 \in C^{\text{init}}$ such that $c_i \rightarrow c_{i+1}$ for every $i : 0 \leq i < n$. A configuration $c \in C$ is *reachable* in $T$ if there is a run $c_0 c_1 \ldots c_n$ of $T$ such that $c = c_n$.

**Monotonicity.** Let $\mathcal{T} = \langle C, C^{\text{init}}, \rightarrow \rangle$ be a transition system and $\sqsubseteq$ be an ordering defined over the set of configurations $C$. We say that the transition relation $\rightarrow$ is monotonic with respect to the ordering $\sqsubseteq$ if, for any configurations $c_1, c_2, c_3 \in C$ such that $c_1 \rightarrow c_2$ and $c_1 \sqsubseteq c_3$, there is a configuration $c_4 \in C$ such that $c_2 \sqsubseteq c_4$ and $c_3 \rightarrow c_4$.

**Well structured transition system.** A *Well-Structured Transition System* (WSTS for short) is a pair $\langle \mathcal{T}, \sqsubseteq \rangle$ where $\mathcal{T} = \langle C, C^{\text{init}}, \rightarrow \rangle$ is a transition system and $\sqsubseteq$ is a WQO over the set of configurations $C$ such that the transition relation $\rightarrow$ is *monotonic* with respect to the ordering $\sqsubseteq$.

**Coverability problem.** Given a transition system $\mathcal{T} = \langle C, C^{\text{init}}, \rightarrow \rangle$, a configuration $c_{\text{target}}$ and an ordering $\sqsubseteq \subseteq C \times C$, the *coverability* problem asks whether there is a configuration $c \in C$ reachable in $\mathcal{T}$ such that $c_{\text{target}} \sqsubseteq c$. As mentioned in Subsection 1.4.2 of Chapter 1, the coverability problem becomes a safety problem when the set $\{c \in C \mid c_{\text{target}} \sqsubseteq c\}$ represents the set of bad configurations in $\mathcal{T}$.

**Decision procedure.** Let $\mathcal{T} = \langle C, C^{\text{init}}, \rightarrow \rangle$ be a transition system and $\sqsubseteq \subseteq C \times C$ be an ordering such that $\langle \mathcal{T}, \sqsubseteq \rangle$ is a WSTS. Moreover, let $c_{\text{target}} \in C$. Abdulla and Jonsson define in [6] a procedure that decides the coverability of $c_{\text{target}}$. The procedure is based on a *backward reachability analysis* which consists in iteratively computing the set of configurations in $C$ that can reach a bad configuration. In [6], a number conditions, sufficient to show that the procedure terminates, are provided. Before giving these conditions, let us first define the following terms.

For a configuration $c \in C$, we define the *upward closure* of $c$ as the set $c^\uparrow := \{c' \in C \mid c \sqsubseteq c'\}$. For a given set of configurations $A \subseteq C$, we use $\text{Pre}(A) := \{c \in C \mid \exists c_a \in A. c \rightarrow c_a\}$ to denote the set of configurations which can reach in one transition a configuration in $A$. Moreover, given a set of configurations $A \subseteq C$ and a subset $M \subseteq A$, we say that $M$ is a *minor* set of $A$ if the following conditions hold:

- For every $c_a \in A$, there is $c_m \in M$ such that $c_m \sqsubseteq c_a$.
- If $c_1, c_2 \in M$ and $c_1 \sqsubseteq c_2$ then $c_1 = c_2$.

We use $\min$ to denote a function which, given a set of configurations $A$, returns a minor set of $A$. Given a configuration $c \in C$, we define:

$$\minpre(c) := \min(\text{Pre}(c \uparrow) \cup (c \uparrow))$$

The conditions that are sufficient to show the decidability of the coverability problem are:
1. The ordering $\sqsubseteq$ should be computable, i.e. for every $c_1, c_2 \in C$ it is possible to check whether $c_1 \sqsubseteq c_2$.

2. For every configuration $c \in C$, we can check whether there is an initial configuration in $c \uparrow$, i.e. whether $C^{\text{init}} \cap (c \uparrow) = \emptyset$.

3. For every configuration $c \in C$, the set $\text{minpre}(c)$ is finite and computable.
4. Ad Hoc Networks

In this chapter, we address the verification problem of Ad-Hoc networks (AHN). An Ad-Hoc network is a formal model of Wireless Ad-Hoc Networks (WANET) and Wireless Sensor Networks (WSN). A WANET is a wireless network composed of computers with a wireless communication capability. Moreover, communication in a WANET does not need a centralized wireless telecommunication infrastructure. Instead, a WANET relays on multi-hop routing protocols that use computers of the network as relays. WSN also adopt the multi-hop communication approach, by which every node in the network can relay messages and participate in the networking tasks. However, WSN are used to monitor environments by sensing and collecting various types of physical information such as temperature and humidity.

Outline.
In section 4.1, we introduce the notions of network topology and selective broadcast. In section 4.2, we give the formal syntax and the operational semantics of the AHN model. Then, we define the verification problem we are interested in, that is the coverability problem for this class of message passing networks. The coverability problem in the AHN setting is defined as a parameterized verification problem that captures some notion of safety. We prove that this problem is undecidable for the general class of AHN. In section 4.3, we define the directed acyclic Ad-Hoc networks and show its relevance particularly for wireless sensor networks. We prove that the coverability problem remains undecidable for this subclass of AHN. In section 4.3, we proceed by defining the subclass of bounded-depth directed acyclic AHN. We show that the coverability problem becomes decidable for this subclass. Finally, in Section 4.4, we compare our contribution with the work of Delzanno et al. in [39].

4.1 Network Topology and Selective Broadcast

An AHN consists of a finite set of processes communicating through message-passing and interconnected through a network topology. Each process in the network consists of a finite set of states and a finite set of transitions. A transition can be (i) local, when it is an internal transition to the process that has no impact on the rest of the network, (ii) a broadcast of a message, or (iii) a reception of a message. The network topology of an Ad-Hoc Network is defined
as a graph. The vertices of the graph represent the nodes in the network where processes are located. The edges of the graph represent the communication capabilities of the underlying wireless medium of the network. Therefore, if there is an edge between node $n_1$ and node $n_2$, then node $n_2$ will be able to “hear” any message broadcasted by $n_1$.

Figure 4.1 depicts two Ad-Hoc Networks $N_1$ and $N_2$ with their respective network topologies $G_1$ and $G_2$. The wireless capabilities of the nodes in network ($N_1$) induce the symmetric graph $G_1$ as a network topology. On the other hand, the network topology of ($N_2$) is the directed graph $G_2$. Observe that, in network ($N_2$), the network topology $G_2$ shows that node ($a$) can send a message to node ($d$) and that the reverse statement is not true. Thus, edges nodes are not necessarily symmetric.

![Network topologies](image)

**Figure 4.1.** Network topologies. Each network $N_1$ and $N_2$ is composed of four wireless nodes (here laptops) with various wireless ranges represented using disks and ellipses. Observe that every node in $N_1$ has the same wireless coverage which leads to the symmetric network topology $G_1$. This is not the case for $N_2$ where node ($c$) cannot send a message to node ($a$) while the reverse is possible.

The topology of a network defines for every node in the network a set of direct neighbors. A direct neighbor of a node $n$ is a node located in the vicinity of $n$. i.e. a node that is covered by the wireless range of $n$. The set of
direct neighbors of a node \( n \) defines which are the nodes in the network that can “hear” node \( n \) when it broadcasts a message. For instance, the set of direct neighbors of node \((a)\) in network \(N_2\) in Figure 4.1 is the set of nodes \(\{(b),(c),(d)\}\) while it is the singleton \(\{(d)\}\) for node \((c)\) and the empty set for node \((d)\).

When a node \( n \) sends a message, the message is broadcasted, i.e. simultaneously sent, to all its direct neighbors. We assume that the message sent is received by a direct neighbor only if the neighbor is in a state where a receiving transition of the same message is enabled. If the receiving transition is not enabled, the message is ignored by the neighboring process. We call this manner of message passing communication selective broadcast. Although selective broadcast does not model possible message collisions that could happen between two nodes emitting simultaneously, it does capture notions such as the vicinity of a node in the network and the notion of radio range. Selective broadcast is the model of message passing that we consider in this work in order to study Ad-Hoc Networks.

### 4.2 Syntax and Semantics

This section defines the syntax and the operational semantics of an Ad-Hoc Network. First, we define the syntax in terms of a process and a graph. Then, we define the transition system associated to an AHN by defining the set of possible AHN configurations and the transition relation over this set. Finally, we define the coverability problem.

**Syntax.**

An ad-hoc network is formally defined by a tuple \(\langle P,G \rangle\), where \(G\) is a graph representing the topology of the network and \(P\) is a finite-state process which is running on every node in the network.

The finite-state automaton \(P\) is defined as the tuple \(\langle Q,M,q_0,\delta \rangle\) where \(Q\) is a finite set of states, \(M\) is a finite set of messages, \(q_0 \in Q\) is the initial state and \(\delta \subseteq Q \times (\{\tau\} \cup \{b(m),r(m) \mid m \in M\}) \times Q\) is the transition relation over the set of states. Observe that \(\tau\) represent any internal transition to the process while \(b(m)\) and \(r(m)\) denote a transition corresponding, respectively, to a sending and a receiving of the message \(m \in M\). For instance, process \(P_1\) depicted in Figure 4.2 is defined by the set of states \(\{q_0,q_1,q_2,q_3,q_4,q_5\}\), the set of messages \(\{m\}\), the initial state \(q_0\) and the transition relation:

\[
\delta_1 = \{\langle q_0, \tau, q_1 \rangle, \langle q_1, b(m), q_3 \rangle, \langle q_0, r(m), q_2 \rangle, \langle q_2, b(m), q_4 \rangle, \langle q_4, \tau, q_5 \rangle\}
\]

The topology of the network, graph \(G = \langle V,E \rangle\), is defined by a tuple composed of a finite set of vertices \(V\) and the finite set of edges \(E \subseteq V \times V\). The vertices of the graph represent the nodes of the network while the edges rep-
resent the communication capabilities of each node in the network. Figure 4.3 shows an example of a graph (graph $G_1$).

As depicted in Figure 4.4, the ad-hoc network $ahn_1 = \langle P_1, G_1 \rangle$ can be seen as a network which topology is defined by the graph $G_1$ and where every node is running a copy of process $P_1$.

Configurations and runs of an AHN.
Let $ahn = \langle P = \langle Q, M, q_0, \delta \rangle, G = \langle V, E \rangle \rangle$ be an ad-hoc network. We define a configuration of $ahn$ as a mapping between the set of vertices $V$ and the set of states $Q$. Intuitively, a configuration defines the current state of the process running on each node of the network. Note that a configuration is initial when every process is in its initial state $q_0$. We use $C_{ahn}$ (respectively $C_{ahn}^{init}$) to denote the set of all possible configurations (respectively initial configurations) of the ad-hoc network $ahn$. Observe that all configurations in $C_{ahn}\langle P, G \rangle$ have the same network topology defined by the graph $G$. 

Figure 4.2. Process $P_1$

Figure 4.3. Graph $G_1 = \langle V_1, E_1 \rangle$

Figure 4.4. The ad-hoc network $ahn_1 = \langle P_1, G_1 \rangle$
Figure 4.5 shows some examples of configurations of the ad-hoc network $\text{ahn}_1 := \langle P_1, G_1 \rangle$. Note that configuration $c_0$ is the only possible initial configuration of the ad-hoc network $\text{ahn}_1$.

![Figure 4.5](image)

**Figure 4.5.** Example of a run of the ad-hoc network $\text{ahn}_1 = \langle P_1, G_1 \rangle$. Observe that the run starts with the initial configuration.

The transition relation $\delta$ of process $P$ induces a transition relation over the set of configurations of the ad-hoc network $\text{ahn}$. There are precisely two types of transitions that relate one configuration $c_1$ to another configuration $c_2$ for $c_1, c_2 \in C_{\text{ahn}}$, a *local* or a *broadcast* transition.

A **local transition** can happen in some configuration $c_1$ if there is a process $p$ in $c_1$ in a state $q$ for which a local transition $\tau$ is enabled, i.e. there exits a state $q'$ such that $(q, \tau, q') \in \delta$. In this case, process $p$ can perform the local transition $\tau$ and move to state $q'$. The new configuration $c_2$ only differs from configuration $c_1$ in the state of process $p$.

A **broadcast transition** can happen in some configuration $c_1$ if there is a process $p$ in $c_1$ in a state $q_1$ for which a broadcast transition is enabled, i.e. there exists a message $m \in M$ and a state $q_2$ such that $(q_1, b(m), q_2) \in \delta$. In this case, process $p$ can move to state $q_2$ and broadcast the message $m$ to all processes in its vicinity, i.e. all its direct neighbors. Simultaneously, each direct neighbor of process $p$ has to receive the message $m$ if it is in a state where a receiving transition of $m$ is enabled. Formally, if a direct neighbor $p'$ of process $p$ is in a state $q_3$ for which there exists $q_4$ such that $(q_3, r(m), q_4) \in \delta$, then process $p'$ has to receive the message $m$ and move to state $q_4$. Otherwise, the direct neighbor $p'$ remains in its current state. The remaining nodes in the ad-hoc network, i.e. nodes that are not in vicinity of the sending process, are not affected by the transition and will keep in $c_2$ the same state they had in $c_1$.

As an example, consider configuration $c_2$ in Figure 4.5. The process running in vertex $v_2$ (colored in red) is in state $q_2$ where the sending transition $b(m)$ is enabled (according to the process definition in Figure 4.2). Observe that the set of direct neighbors of $v_2$ is $\{v_3, v_4\}$ (colored in green in $c_2$). Processes in these two vertices are in state $q_0$ in which the receiving transition
\textbf{r}(m) is enabled. Therefore, when the process in \( v_2 \) sends message \( m \), the message is received by both nodes \( v_3 \) and \( v_4 \), the sending process moves to state \( q_4 \), the receiving processes move to state \( q_2 \) and we obtain configuration \( c_3 \).

Observe that none of the local or broadcast transitions do change the graph of the network.

The transition relation \( \sigma \) of process \( P \) induces a transition relation \( \longrightarrow_{\text{ahn}} \subseteq \mathcal{C}_{\text{ahn}} \times \mathcal{C}_{\text{ahn}} \) over the set of configurations \( \mathcal{C}_{\text{ahn}} \) for the ad-hoc network \( \text{ahn} = \langle P, G \rangle \). For two configurations \( c_1, c_2 \in \mathcal{C}_{\text{ahn}} \), we use \( c_1 \longrightarrow_{\text{ahn}} c_2 \) to denote that there is a (local or a broadcast) transition that generates \( c_2 \) from \( c_1 \). Moreover, we associate to the Ad-Hoc network \( \text{ahn} \) the transition system \( \mathcal{T}_{\text{ahn}} := \langle \mathcal{C}_{\text{ahn}}, \mathcal{C}^{\text{init}}_{\text{ahn}} \longrightarrow_{\text{ahn}} \rangle \) and use \( ^*_{\text{ahn}} \) to denote the reflexive transitive closure of \( \longrightarrow_{\text{ahn}} \).

We define a run of the transition system \( \mathcal{T}_{\text{ahn}} \) as a finite sequence of configurations \( c_0, c_1, \ldots, c_n \) such that \( c_0 \in \mathcal{C}^{\text{init}}_{\text{ahn}} \) and \( c_i \longrightarrow_{\text{ahn}} c_{i+1} \) for every \( i : 0 \leq i < n \). Figure 4.5 depicts one possible run of the Ad-Hoc network \( \text{ahn}_1 \).

Observe that in configuration \( c_3 \), message \( m \) has reached all nodes that are remotely connected to the node that first sent the message. Informally, process \( P_1 \) in Figure 4.2 implements a diffusion algorithm that simply tries to deliver some message \( m \) to every reachable node in the network. This message can, for instance, be a fire alarm or an information request.

We say that a configuration \( c \in \mathcal{C}_{\text{ahn}} \) is reachable in the transition system \( \mathcal{T}_{\text{ahn}} \) if there exists a run \( c_0, \ldots, c_n \) of \( \mathcal{T}_{\text{ahn}} \) such that \( c_n = c \). Moreover, we say that a state \( q \in Q \) of process \( P \) is reachable in \( \mathcal{T}_{\text{ahn}} \) if it is the current state of one of the processes of a configuration reachable in \( \mathcal{T}_{\text{ahn}} \).

The coverability problem.

Let \( P = \langle Q, M, q_0, \delta \rangle \) be a finite-state process and \( q_{\text{target}} \in Q \) be one of its states. The coverability problem (\text{COVER} for short) asks whether there exists a graph \( G \) such that the state \( q_{\text{target}} \) is reachable in the transition system \( \mathcal{T}_{\text{ahn}} \) associated with the ad-hoc network \( \text{ahn} := \langle P, G \rangle \).

Observe that there are no constraints on the network topology \( G \). Answering the coverability problem can tell us whether some states, considered as bad, can be reached in some network topologies. If a bad state is reachable for some network topology, then process \( P \) is considered as unsafe. Otherwise, it is considered as safe.

Delzanno et al. in [39] show that \text{COVER} is undecidable for the class of ad-hoc networks in which the graph topology is symmetric. They do so through a reduction from the halting problem for the two-counters Minsky machine. Since symmetric graphs are a subset of directed graphs, it can easily be shown that the reduction still holds if we consider ad-hoc networks with directed graphs.
4.3 Directed Acyclic Ad-Hoc Networks

In our quest towards finding a subclass of ad-hoc networks for which the cov-
erability problem is at least decidable, we consider in Paper I the subclass of
ad-hoc networks for which the topology is a Directed Acyclic Graph (DAG for
short). Recall that a directed graph is acyclic when there exists no simple path
in the graph of length strictly larger than one that starts and ends in the same
vertex.

The subclass of directed acyclic ad-hoc networks (DAAHN for short) is par-
ticularly meaningful for wireless sensors networks. In fact, it is often the case
in WSN that data flows in directed and acyclic paths. More precisely, data
flows following two possible scenarios (see Figure 4.6). It can flow down-
stream from a central node (or sink) down to the sensors in order to request
some information or to send some firmware updates and reconfiguration re-
quests. Data can also be collected and sent upstream from the sensors back
to the sink. We assume that the directed acyclic topology of the network in a
WSN is in general built beforehand by the preliminary use of routing proto-
cols.

![Diagram of directed and acyclic communication in a WSN.]

Although the class of directed acyclic AHN is a strict subclass of the general
class of AHN, we show in subsection 4.3.1 that the coverability problem for
DAAHN remains undecidable. Therefore, we introduce in subsection 4.3.2 the
class of bounded depth DAAHN and prove the decidability of the coverability
problem for this subclass.

4.3.1 COVER is undecidable for directed acyclic AHN

Since the flow of messages is acyclic in a DAAHN network, we cannot reuse
the two-counters Minsky machine reduction conducted by Delzanno et al. in
[39] in order to show the undecidability of COVER of undirected AHN. Instead,
we need to use an undecidable problem that can be encoded into a network in which messages flow in one direction only and where messages cannot be sent back. The TRANSD problem, mentioned in the preliminaries (Chapter 3), is an undecidable problem that can be used for that purpose.

Recall that an instance of the TRANSD problem is composed of two finite-state automata $A$ and $B$ and a transducer $T$ all operating on the same alphabet $\Sigma$. TRANSD asks whether there exists a natural number $i \geq 0$ such that $T^i(L(A)) \cap L(B) \neq \emptyset$. Equivalently, the TRANSD problem asks whether there exists a word $w$ accepted by automaton $A$ such that one of its repetitive possible transductions is accepted by automaton $B$. As shown in the preliminaries, the TRANSD problem is undecidable.

While the details of the proof can be found in Paper I, we give here a high-level description of the reduction of the TRANSD problem to the coverability problem for directed acyclic AHN. Let $\langle A, B, T \rangle$ be an instance of the TRANSD problem composed of two finite-state automata $A$ and $B$ and a transducer $T$, all defined over the same alphabet $\Sigma$. Using automata $A$ and $B$ and the transducer $T$, we build a process $P_{\langle A, B, T \rangle}$ such that any directed acyclic AHN defined using $P_{\langle A, B, T \rangle}$ will generate a transduction chain.

**Transduction chain.**

As depicted in Figure 4.7, a transduction chain denotes a set of nodes in the ad-hoc network containing at least two nodes:

$$n_1, n_2, n_3, \ldots, n_m, m \geq 2$$

Every node $n_i$ ($i : 1 \leq i \leq m - 1$) in the chain can send messages to the next node in the chain, i.e. node $n_{i+1}$. In the case of an ad-hoc network, this translates into the existence of an edge between $n_i$ and $n_{i+1}$ for every $i : 1 \leq i \leq m - 1$.

Moreover, every node in the chain has a particular role. For instance, the head of the chain, i.e. node $n_1$, is in charge of running automaton $A$, while the tail of the chain, node $n_m$, is in charge of simulating automaton $B$. Every other node in the chain, i.e. nodes $n_i, i : 2 \leq i \leq m - 1$, simulates transducer $T$.

Node $n_1$ simulates automaton $A$ by traversing it and sending letters read during the traversal. The letters are sent to the next node in the chain, i.e. node $n_2$. If at some moment, node $n_1$ is in an accepting state

![Figure 4.7. A transduction chain of length equal to 5. Node $n_1$ simulates automaton $A$, node $n_5$ simulates automaton $B$ and nodes $n_2, n_3, n_4$ simulate transducer $T$.](image)
of $A$, it may send a special acceptance message to $n_2$ and stop the simulation of $A$. The acceptance message is used to notify $n_2$ that the word $w$ it received so far is in $L(A)$.

Every node $n_i, i : 2 \leq i \leq m - 1$, traverses transducer $T$ by reading letters received from the previous node in the chain (node $n_{i-1}$) and sending the transduction of the received letters to the next node in the chain (node $n_{i+1}$). If $n_i$ receives the acceptance message while being in an accepting state of transducer $T$, then it forwards the same message to the next node in the chain to notify the node that the word it has received so far is from $T^{i-1}(L(A))$.

Node $n_m$ traverses automaton $B$ by reading letters it receives from the previous node in the chain (node $n_{m-1}$). If it receives the acceptance message while being in an accepting state of automaton $B$, then it moves to a special state $q_{\text{accept}}$. In that case, the word it has received so far is a word from both $T^{m-2}(L(A))$ and $L(B)$.

The construction of the chain, which details can be found in Paper I, isolates the nodes of the chain from the rest of the network. Moreover, since simple paths in the topology of the ad-hoc network are not bounded, there is no bound on the length of the transduction chain neither. Consequently, for every possible value of $i \geq 0$, there exists a graph $G$ such that the transduction chain can be encoded in the ad-hoc network $\text{ahn} := \langle P_{(A,B,T)}, G \rangle$. Finally, we show that the instance of the TRANS problem has a solution if and only if the state $q_{\text{accept}}$ is reachable, thus proving the undecidability of COVER for the subclass of directed acyclic AHN.

A key element in the reduction from the TRANS problem is the fact that the directed acyclic topology of the ad-hoc network is not a bounded depth topology, i.e. there is no bound on the length of simple paths in the graph of the networks. This motivates the study of the case in which we only consider DAAHN with bounded depth. Figure 4.8 shows examples of configurations and their associated depth. Observe that bounding the depth of the topology does not limit the size of the network. Indeed, after bounding the depth of the topology, we are still dealing with systems composed of an unbounded number of processes.

![Figure 4.8. Examples of configurations for fixed topology depths.](image-url)
Given a process $P = \langle Q, M, q_0, \delta \rangle$, a state $q_{\text{target}} \in Q$, and a natural number $k \geq 1$, we define the bounded-depth coverability problem (BOUNDED-COVER for short) as the problem which asks whether there exists a $k$-bounded depth graph $G$ such that the state $q_{\text{target}}$ is reachable in the transition system $\mathcal{T}_{\text{ahn}}$ defined by the Ad-Hoc network $\text{ahn} = \langle P, G \rangle$.

4.3.2 Coverability of bounded depth DAAHN is decidable
To show the decidability of BOUNDED-COVER, we proceed in two steps. First, we reduce BOUNDED-COVER to the coverability problem for bounded-depth inverted-trees (TREE-BOUNDED-COVER for short), in which we only consider DAAHN which topologies are inverted trees. Second, we show that TREE-BOUNDED-COVER is decidable through a nontrivial instantiation of the Well-Structured Transition System (WSTS for short) framework. Although the details of the reductions and the proofs can be found in Paper I, we give here a high-level description of every step.

From BOUNDED-COVER to TREE-BOUNDED-COVER
To show that BOUNDED-COVER reduces to TREE-BOUNDED-COVER, we first define a split operator that transforms a directed acyclic graph into an inverted forest. Using this split operator, we show that the BOUNDED-COVER problem reduces the coverability problem for ad-hoc networks which topology is an inverted forest (FOREST-BOUNDED-COVER for short). Finally, we show that FOREST-BOUNDED-COVER and TREE-BOUNDED-COVER are equivalent.

The split operator. Given a DAG $G = \langle V, E \rangle$, the split operator builds a graph $G^* = \langle V^*, E^* \rangle$ as follows. For every vertex $v$ in $V$ and for every distinct simple path $\pi$ in $G$ such that $v \cdot \pi$ ends in a leaf of $G$, the split operator associates a vertex $u = v \cdot \pi$ in $V^*$. Moreover, the set of edges in $G^*$ is given by $E^* := \{ \langle \pi_1, \pi_2 \rangle \mid \pi_1 = v \cdot \pi_2 \}$. Observe that any vertex in $F$ has at most one successor. Therefore, $F$ is an inverted forest. As an example, consider graphs $G$ and $F$ in Figure 4.9. The inverted forest $F$, composed of the two inverted tress $T_1$ and $T_2$, is the result of applying the split operator on the DAG $G$. Consider vertex 3 in $G$. From vertex 3, we can reach a leaf of $G$ through three distinct simple paths, 324, 325 and 35. Therefore, three copies of this vertex exist in $F$, namely vertices 324, 325 and 35, each one of them corresponds to a distinct simple path in the set $\{321, 325, 35\}$.

From BOUNDED-COVER to FOREST-BOUNDED-COVER. Let an instance of the BOUNDED-COVER problem be process $P = \langle Q, M, q_0, \delta \rangle$, state $q_{\text{target}} \in Q$ and depth $k \geq 0$. We show that BOUNDED-COVER and FOREST-BOUNDED-COVER are equivalent for the same instance. Let $G$ be a $k$-bounded depth DAG. Observe that $G^*$ is an inverted forest graph of depth bounded by $k$. We show in Paper I that any state reachable in the ad-hoc network $\text{ahn} = \langle P, G \rangle$ is also reachable in the ad-hoc network $\text{ahn}^* = \langle P, G^* \rangle$. Therefore, if
BOUNDED-COVER has a positive answer for the instance $\langle P, q_{\text{target}}, k \rangle$, then FOREST-BOUNDED-COVER has also a positive answer for the same instance. The reverse statement trivially holds since any $k$-bounded depth forest is also a $k$-bounded depth DAG.

**From FOREST-BOUNDED-COVER to TREE-BOUNDED-COVER.**

Consider an ad-hoc network with an inverted forest topology. Since processes located in different inverted trees cannot interact with each other, the set of states that are reachable by the ad-hoc network is the union of the sets of states reachable by every inverted sub-tree. Using this argument, we show that FOREST-BOUNDED-COVER is equivalent to TREE-BOUNDED-COVER.

**TREE-BOUNDED-COVER is decidable.**

As mentioned in the preliminaries, a WSTS is a transition system $\mathcal{T}_{\text{WSTS}} = \langle C_{\text{WSTS}}, C_{\text{init}}^{\text{WSTS}}, \rightarrow_{\text{WSTS}} \rangle$ together with an ordering $\sqsubseteq$ defined over the set of configurations $C_{\text{WSTS}}$. The ordering $\sqsubseteq$ should be a WQO and the transition relation $\rightarrow_{\text{WSTS}}$ should be monotonic with respect to that ordering. The framework of WSTS aims at solving the coverability problem which, given a configuration $c_{\text{bad}} \in C_{\text{WSTS}}$, consists in asking whether there exists a configuration $c \in C_{\text{WSTS}}$ reachable in $\mathcal{T}_{\text{WSTS}}$ such that $c_{\text{bad}} \sqsubseteq c$. Abdulla and Jonsson give in [6] a number of sufficient conditions in order to show the decidability of the coverability problem.

We show the decidability of the TREE-BOUNDED-COVER problem by reducing the problem to the coverability problem of a WSTS transition system that we define. This reduction is conducted for any instance of the TREE-BOUNDED-COVER problem. Let $\text{lst} = \langle P, q_{\text{target}}, k \rangle$ be such an instance defined by an AHN process $P = \langle Q, M, q_0, \delta \rangle$, the state $q_{\text{target}} \in Q$ and the depth bound $k \geq 0$. We start by defining the transition system $\mathcal{T}_{\text{lst}} = \langle C_{\text{lst}}, C_{\text{init}}^{\text{lst}}, \rightarrow_{\text{lst}} \rangle$. Then, in order to show that $\mathcal{T}_{\text{lst}}$ is a WSTS, we define an ordering $\sqsubseteq$ over the set of configurations $C_{\text{lst}}$ and show that the ordering is a WQO. Then, we show that the transition relation $\rightarrow_{\text{lst}}$ is monotonic wrt. to this ordering. Fi-
nally, we define a configuration $c_{\text{Ist}} \in C_{\text{Ist}}$ and show how the TREE-BOUNDED-COVER problem reduces to the coverability of $c_{\text{Ist}}$ in $\mathcal{T}_{\text{Ist}}$.

**Transition system $\mathcal{T}_{\text{Ist}}$.** We define $C_{\text{Ist}}$ to be the union of the sets of configurations of every ad-hoc network $\langle P, T \rangle$ defined by process $P$ and any $k$-bounded depth inverted tree $T$, i.e. $C_{\text{Ist}} := \bigcup_{k\text{-bounded depth inverted tree } T} C_{\langle P,T \rangle}$. In a similar manner, we define the set of initial configurations $C_{\text{Ist}}^{\text{init}}$ as the union of sets of initial configurations of every ad-hoc network $\langle P, T \rangle$ defined by process $P$ and any $k$-bounded depth inverted tree $T$. Finally, we set $\mathcal{T}_{\text{Ist}} := \bigcup_{k\text{-bounded depth inverted tree } T} T_{\langle P,T \rangle}$.

**The ordering.** In order to show that $\mathcal{T}_{\text{Ist}} = \langle C_{\text{Ist}}, C_{\text{Ist}}^{\text{init}}, \rightarrow_{\text{Ist}} \rangle$ is a WSTS, the first step is to define an ordering $\subseteq$ over the set $C_{\text{Ist}}$ and show that this ordering is a WQO. The ordering we define over the set of $k$-bounded depth inverted tree configurations is the induced subgraph relation. To show that this ordering is a WQO, we first encode inverted-tree configurations into higher order multisets. Then we define an ordering $\sqsubseteq'$ over higher order multisets that we show is equivalent to the induced subgraph relation. Finally, we show that $\sqsubseteq'$ is a WQO over the set of (bounded height) higher order multisets. Figure 4.10 shows examples of inverted-tree configurations related through the induced subgraph relation and their higher order multiset encoding.

**Monotonicity.** In order to show that $\mathcal{T}_{\text{Ist}}$ is a WSTS, the second step is to prove the monotonicity of the transition relation $\rightarrow_{\text{Ist}}$ with respect to the ordering $\subseteq$. Recall that proving the monotonicity of $\rightarrow_{\text{Ist}}$ wrt. $\subseteq$ amounts to show that for every $c_1, c_2$ and $c_3$ from $C_{\text{Ist}}$, if $c_1 \rightarrow_{\text{Ist}} c_2$ and $c_1 \subseteq c_3$, then there should be a configuration $c_4 \in C_{\text{Ist}}$ such that $c_3 \rightarrow_{\text{Ist}} c_4$ and $c_2 \subseteq c_4$. Moreover, observe that a transition in $\rightarrow_{\text{Ist}}$ can either be a local or a broadcast transition. Therefore, proving the monotonicity of $\rightarrow_{\text{Ist}}$ wrt. $\subseteq$ amounts to show that both local and broadcast transitions are monotonic wrt. $\subseteq$.

The case of a local transition is straightforward since it only involves the process performing the transition and it has no impact on the rest of the network.
Assume that the transition in $\rightarrow_{\text{lst}}$ is a broadcast transition of a message $m \in M$. Let $v$ be the vertex in the inverted tree of configuration $c_1$ that performs this transition. Observe that every vertex in an inverted tree topology has at most one direct neighbor, its successor. In the case where $v$ has no successor, i.e. $v$ is the root of the inverted tree, then the broadcast transition has no impact on the rest of the network. In this case, a broadcast transition behaves as a local transition. In the other case, i.e. if $v$ has precisely one successor, then the only other process in the network that might be impacted by this transition is the successor of $v$. Since $c_1 \subseteq c_3$ and $\subseteq$ corresponds to the subgraph ordering, then $v$ and its successor are embedded in $c_3$ and can therefore perform the same transitions as in $c_1$. Figure 4.11 shows the case where the successor of $v$ is in a state where it can receive message $m$.

The reduction. Given an instance $\text{lst} = \langle P, q_{\text{target}}, k \rangle$ of the TREE-BOUNDED-COVER problem, we defined a transition system $\mathcal{T}_{\text{lst}} = \langle C_{\text{lst}}, C_{\text{lst}}^{\text{init}}, \rightarrow_{\text{lst}} \rangle$, an ordering $\subseteq$ over the set of $k$-bounded depth inverted tree configurations $C_{\text{lst}}$ and proved that $\mathcal{T}_{\text{lst}}$ is well structured. Let $c_{\text{lst}}$ be the configuration associated with a graph $\langle V, E \rangle$ where the set of vertices $V$ is a singleton $\{v\}$, $E = \emptyset$ and $v$ is in state $q_{\text{target}}$, i.e. $c_{\text{lst}}(v) = q_{\text{target}}$. The configuration $c_{\text{lst}}$ is clearly an inverted tree configuration that is $k$-depth bounded for any $k \geq 0$. The set of $k$-bounded depth inverted tree configurations in which there is a process in state $q_{\text{target}}$ exactly corresponds to the set $\uparrow c_{\text{lst}} := \{c \in C_{\text{lst}} \mid c_{\text{lst}} \subseteq c\}$. This is the reason why TREE-BOUNDED-COVER for the instance $\text{lst}$ reduces to the coverability of $c_{\text{lst}}$ in $\mathcal{T}_{\text{lst}}$.

Finally, we show in Paper I that the sufficient conditions listed in [6], needed to show the decidability of the coverability problem, hold.

4.4 Related Works

The Ad-Hoc Network model we use in this work is mainly inspired by the model proposed by Delzanno et al. in [39]. While the network topology considered by Delzanno et al. is an undirected graph, we considered in our work
the wider setting of a directed graph as a network topology. Directed graphs allow having asymmetric communication capabilities in the network. Delzanno et al. studied in [39] three problems in three different network topology settings.

The COVER problem is the first problem considered in [39]. This problem is identical to the coverability problem that we consider in our work. The second problem, TARGET, asks whether there exists a reachable configuration in which the state of every process in the configuration belongs to a given subset of target states. Answering this problem can help checking whether the system can end up in a deadlock configuration. Observe that the lack of cycles in directed acyclic graphs reduces TARGET for the case of directed acyclic AHN to the simple state reachability problem in a finite-state machine. Finally, the third problem considered in [39], i.e. the REPEATED-COVER problem, consists of checking whether there exists a run of the system in which some given state is infinitely often reachable. We did not consider this problem for the case of directed acyclic graphs.

The first setting considered in [39] is the general (unbounded) static topology for which Delzanno et al. proved the three problems to be undecidable. In the second setting, Delzanno et al. assumed that the nodes in the network are mobile. To model this assumption, the set of vertices of the network topology graph have the possibility to non-deterministically change at any point in time during a run of the network. Surprisingly enough, they showed that the three verification problems become decidable in this setting, a result that is similar to what has been shown in [6] concerning channel systems when they become lossy. Finally, they consider in the third setting networks which topologies are static and have bounded simple path length. Delzanno et al. proved that only the COVER problem is decidable for the third setting. This result was established through an instantiation of the WSTS framework. They use for that purpose the induced subgraph relation as an ordering. A result by Ding in [42] has shown that the induced subgraph relation is a WQO over the set of undirected bounded simple path graphs.

Because of the local broadcast type of communication considered in AHN, the choice of the induced subgraph relation as an ordering is more appropriate than the weaker embedded subgraph relation in order to ensure the monotonicity of the transition relation.

We also use the induced subgraph relation as an ordering. However, we cannot adopt the same instantiation of the WSTS for our case of bounded-depth directed acyclic AHN. In fact, Figure 4.12 gives a sequence
of directed acyclic depth-bounded configurations that shows that the induced subgraph ordering is not a WQO for the set of bounded-depth directed acyclic graphs. Nevertheless, we could have used the result of [42] on bounded depth inverted-trees, but this would have required some extra steps in order to encode the direction of the edges. Instead, we encoded bounded-depth inverted trees into higher order multiset{s} and defined an ordering over (bounded) higher order multiset{s} that we proved to be a WQO.
5. Dynamic Register Automata

We present in this chapter our work on the verification of dynamic networks of processes communicating through message passing (Paper II and III). More precisely, we are interested in networks in which each process is capable of creating new processes and acting upon the topology of the network. We use the formal model of Dynamic Register Automata (DRA for short), a model similar to the model of Dynamic Communicating Automata, which was introduced in [23, 24] as an extension of the classical register automaton. A DRA process is a finite-state automaton equipped with a set of registers. We assume that every process in the network has a unique identifier that serves as its address in the network. The registers of a process may contain the identifier of a process in the network. A process can send a message to another process only if it knows the ID of the receiver. Therefore, the content of the registers define the communication capability of each process, hence the communication topology of the network. By sending and receiving process identifiers, processes change the topology of the network. Besides sending and receiving messages and process identifiers, DRA processes may also create new processes. These two features make the network topology of such networks dynamic.

The present work focuses on the state reachability problem. The state reachability problem for DRA networks consists in checking whether it is possible for the network to evolve into a configuration in which one of the processes is in a state considered to be bad. Since the number of processes in the network is unbounded, this problem is undecidable in the general case. We analyze this problem for two variants of message passing.

In the first message passing variant, considered in Paper II, we assume that the communication between processes is rendezvous based, meaning that messages are sent and received between processes in a synchronous manner. We call DRA processes using rendezvous communication synchronous DRA. We show that the state reachability problem for synchronous DRA is undecidable in the general case. In order to find a subclass of DRA for which the problem becomes decidable, we explore different restrictions on the graph topology of the network. We show that the state reachability problem remains undecidable even if we restrict the network topologies to strongly bounded graphs. Later, we define the class of degenerative DRA in which processes can nondeterministically reset any of their registers. We show through an instantiation of the Well Structured Transition Systems [6] framework that the state reachability problem becomes decidable for the subclass of degenerative DRA when the network topology is strongly bounded.
In the second variant of message passing (addressed by Paper III), we assume that the communication in the DRA network is asynchronous. In this setting, messages that are sent to a process are not received immediately by the receiver but are instead stored in a buffer. The buffer plays the role of a mailbox for the receiver and is implemented using a FIFO queue. We call this sort of DRA buffered DRA. We show that the state reachability problem for buffered DRA is undecidable in general. We show that the problem becomes decidable when (i) we restrict the topology of the network to strongly bounded graphs, (ii) we bound the size of the buffers, and (iii) we allow processes to disconnect themselves in a non-deterministic manner from the network.

Outline.
In Section 5.1, we give the syntax and the operational semantics of the synchronous DRA. In Section 5.2, we show that the state reachability problem of synchronous DRA is undecidable not only in the general case but also when we (strongly) bound the graph topology of the network. In Section 5.3, we introduce the class of degenerative synchronous DRA and show that the state reachability problem becomes decidable for degenerative synchronous DRA when the graph topology of the network is strongly bounded. In Section 5.4, we define buffered DRA, the asynchronous variant of DRA, and give its operational semantics. In Section 5.5, we present our findings concerning the decidability of the state reachability problem for buffered DRA.

5.1 Synchronous DRA
In this section, we first give the syntax of synchronous DRA and the definition of its associated network configuration. Then, we give the operational semantics associated with the synchronous DRA model. Finally, we define the verification problem we are interested in, i.e. the state reachability problem.

5.1.1 Syntax
In the following, we give the formal definition of a synchronous DRA and define its associated DRA network configurations.

Process definition.
A DRA is a finite-state automaton defined as a tuple $\langle Q, q_{init}, M, X, \delta \rangle$ where $Q$ is a finite set of states, $q_{init}$ is the initial state, $M$ is a finite set of messages, $X$ is a finite set of registers and $\delta$ is the transition relation. Every transition from $\delta$ is of the form $\langle q_1, action, q_2 \rangle$ where $q_1$ and $q_2$ are two states from $Q$ and $action$ is the action performed by the process. Let $x, y \in X$ be two registers, $z \in X \cup \text{self}$ be a register or the ID of the process, $q \in Q$ be a state
and $m \in M$ be a message. An action can be either: (i) a local action $\tau$, (ii) a process creation of the form $x \leftarrow \text{create}(q,y)$, (iii) a message sending of the form $x!\langle m \rangle$, (iv) an ID sending of the form $x!\langle z \rangle$, (v) a selective message reception of the form $x?\langle m \rangle$, (vi) a non-selective message reception of the form $*?\langle m \rangle$, (vii) a selective ID reception of the form $x?\langle y \rangle$, (viii) a non-selective ID reception of the form $*?\langle y \rangle$, or (ix) a register reset of the form $\text{reset}(x)$.

5.1.2 Operational semantics

Let $D = \langle Q, q_{\text{init}}, M, X, \delta \rangle$ be a synchronous DRA. We mentioned earlier that a transition $t \in \delta$ is of the form $\langle q_1, \text{action}, q_2 \rangle$ where $q_1, q_2 \in Q$ and $\text{action}$ can be one of the following: (i) a local action, (ii) a process creation, (iii) a

Network configuration and graph encoding.

A DRA network configuration is composed of a finite and unbounded number of DRA processes. Each process in a DRA network comes with a unique identifier. Let $P$ be the domain of all possible process IDs. The set of registers $X$ of each DRA process are used to store process IDs from the domain $P$. Formally, a DRA network configuration is defined as a tuple $\langle \text{procs}, s, r \rangle$ where $\text{procs}$ is the set of processes present in the network, $s : P \rightarrow Q$ is a partial function that maps each process $p \in \text{procs}$ to its current state and $r : P \rightarrow \{ X \rightarrow \text{procs} \}$ is a partial function that maps every process $p \in \text{procs}$ to its registers contents.

As depicted in Figure 5.1, we associate to every DRA configuration $c$ an encoding $\text{enc}(c)$ in the form of a labeled graph. Notice that the network configuration in Figure 5.1 is composed of three processes of respective IDs 0, 1 and 7 in, respectively, states $s_1$, $s_2$ and $s_3$. Moreover, each process has two registers, $x$ and $y$. Every vertex in the graph encoding $\text{enc}(c)$ of a configuration $c$ represents a process in the network. A vertex is labeled with the state of the process it represents. For instance, process 0 is represented in the graph encoding of $c_{\text{example}}$ with a vertex labeled with $s_1$, the state of process 0. If some register $x$ of some process $p$ contains the ID of some other process $p'$, then this is encoded with an edge labeled with $x$ between the vertex representing $p$ and the vertex representing $p'$. In configuration $c_{\text{example}}$, register $x$ of process 7 is empty while register $y$ of the same process contains the ID of process 0. Therefore, the content of register $y$ of process 7 is represented in the graph encoding $\text{enc}(c_{\text{example}})$ with an edge between the vertex representing process 7 and the vertex representing process 0. Furthermore, this edge is labeled with $y$, i.e. the name of the register.

The graph encoding $\text{enc}$ informs about the current state of each process and the content of their registers. We use this encoding to define the network topology of a configuration.
message or ID sending, (iv) a (selective or non-selection) message or ID reception, or (v) a register reset. We define in the following paragraphs the semantics of each one of the following actions: a local action, a register reset, a process creation, an ID sending and (selective or non-selection) reception. The semantics of the actions of message sending and reception are similar to the semantics of the actions of ID sending and reception. The details of the semantics of these actions can be found in Paper II.

In the following, we assume that $t = q_1, action, q_2 \in \delta$ is a transition of $D$ and that $c = \langle procs, s, r \rangle$ is a network configuration associated to $D$.

**Local action.**
Assume that $t = \langle s_1, \tau, s_2 \rangle$ corresponds to a local transition and that there exists a process $p \in procs$ in configuration $c$ which state is $s_1$ (i.e. $s(p) = s_1$). Process $p$ can perform the local action $\tau$ and move to state $s_2$. The configuration that results from this action only differs from the original configuration $c$ in the state of $p$.

**Register reset action.**
Assume that transition $t$ which corresponds to the action of resetting a register $x \in X$, i.e. $t = \langle s_1, \text{reset}(x), s_2 \rangle$ for some states $s_1, s_2 \in Q$. Moreover, assume that there exists in configuration $c$ a process $p \in procs$ in state $s_1$ (i.e. $s(p) = s_1$). By performing transition $t$, process $p$ can reset its register $x$ to the undefined value $\perp$ and move to state $s_2$. The configuration that results from this action only differs from the original configuration $c$ in the state of process $p$ and in the content of its register $x$.

Observe that both local and register reset actions only impact the state or the register content of the acting process. This is not the case anymore when the transition corresponds to a process creation or to a sending (and reception) of...
a message (or an ID), where the transition can have an impact on the network. We illustrate in the following two paragraphs the impact on a DRA network configuration of a process creation and the sending (and reception) of an ID.

**Process creation.**
Assume that $t = \langle s_1, x \leftarrow create(s_3, y), s_2 \rangle$ and that there is a process $p \in \text{procs}$ in configuration $c$ such that $s(p) = s_1$. By executing transition $t$, process $p$ creates a new process $p' \notin \text{procs}$ in state $s_3$ and its state becomes $s_2$. Moreover, $p$ sets its register $x$ to point to the newly created process $p'$ while $p'$ sets its register $y$ to the ID of the creating process $p$.

Figure 5.2 shows an example of a process creation that took place in a configuration $c_1$ and resulted in a configuration $c_2$. According to its graph encoding, configuration $c_1$ contains two processes $p_1$ and $p_2$, respectively in states $s_1$ and $s_3$. We assume that $\langle s_1, x \leftarrow create(s_4, y), s_2 \rangle$ is a transition of the DRA. Since process $p_1$ is in $s_1$, $p_1$ can create a new process $p_3$ in state $s_4$. By doing so, process $p_1$ moves to state $s_2$ and sets its register $x$ to the ID of the newly created process $p_3$. Moreover, the newly created process $p_3$ in the new configuration $c_2$ has empty registers besides register $y$, which is set to the ID of $p_1$, the creating process. The graph encoding of configuration $c_2$ in Figure 5.2 shows the resulting configuration. Notice that the content of register $x$ of process $p_1$ has been erased and replaced with the ID of the newly created process $p_3$ in configuration $c_2$.

![Graph encoding of $c_1$](image1.png) ![Graph encoding of $c_2$](image2.png)

*Figure 5.2. Process creation.*

**ID sending and receiving.**
Assume that there are two transitions $t_{\text{snd}}, t_{\text{rcv}} \in \delta$ such that $t_{\text{snd}} = \langle s_1, x! \langle y \rangle, s_2 \rangle$ is an ID sending transition and $t_{\text{rcv}} = \langle s_3, \ast? \langle z \rangle, s_4 \rangle$ corresponds to a (non-selective) ID reception transition. Moreover, assume that there are two distinct processes $p_1$ and $p_2$ in the set of processes $\text{procs}$.
of configuration $c$ that are respectively in state $s_1$ and $s_3$ (i.e. $s(p_1) = s_1$ and $s(p_2) = s_3$), and assume that register $x$ of $p_1$ contains the ID of $p_2$ (i.e. $r(p_1)(x) = p_2$). In this setting, it is possible for process $p_1$ to perform transition $t_{\text{snd}}$ when process $p_2$ simultaneously performs transition $t_{\text{rcv}}$. The execution of these two transitions, $t_{\text{snd}}$ and $t_{\text{rcv}}$ has the following impact: (i) the content of register $y$ of process $p_1$, either an ID or the empty value $\bot$, is sent to process $p_2$, (ii) process $p_2$ receives this content and stores it in its register $z$, and (iii) processes $p_1$ and $p_2$ move to, respectively, states $s_2$ and $s_4$.

Figure 5.3 shows an example of an ID communication that took place in the depicted configuration $c_2$ and resulted in configuration $c_3$. According to its graph encoding, configuration $c_2$ contains three processes $p_1$, $p_2$ and $p_3$, respectively in states $s_2$, $s_3$ and $s_4$. We assume that $\langle s_2, y!\langle x\rangle, s_5 \rangle$ and $\langle s_3, *?\langle x\rangle, s_6 \rangle$ are two transitions of the synchronous DRA. Since process $p_1$ is in state $s_2$, $p_2$ is in state $s_3$ and register $y$ of $p_1$ points to $p_2$, $p_1$ can send the content of its register $x$ to $p_2$ and $p_2$ can receive it and store it in its register $x$. Observe that register $x$ of process $p_1$ contains the ID of process $p_3$. In the resulting configuration $c_3$, processes $p_1$ and $p_2$ have moved respectively to states $s_5$ and $s_6$ while register $x$ of process $p_2$ is now pointing to process $p_3$. The graph encoding of configuration $c_3$ in Figure 5.3 shows the result of this synchronous ID sending and receiving.

Observe that the transition $\text{action}_{\text{rcv}} = \langle s_3, *?\langle z\rangle, s_4 \rangle$ is a non-selective ID reception since there is no condition on where the ID to be received should come from. This is not the case for the selective type of ID reception in which a condition is put on the ID of the sending process. For instance, the selective $\langle s_3, x?\langle z\rangle, s_4 \rangle$ ID reception transition can be triggered only if register $x$ of the receiver contains the ID of the sender.

![Figure 5.3. Sending and receiving of an ID.](image)
5.1.3 State reachability problem

Given a DRA $D = (Q, q_{init}, M, X, \delta)$, we use $C_D$ to denote the set of all possible network configurations running the DRA $D$, $c_D^{init}$ to denote the network configuration composed of one single process in the initial state $q_{init}$ and $\rightarrow_D \subseteq C_D \times C_D$ to denote the transition relation induced by $\delta$.

We say that a state $q \in Q$ is reachable in the transition system $\mathcal{T}(D) := \langle C_D, \{c_D^{init}\}, \rightarrow_D \rangle$ if there exist a configuration $c_n = (\text{procs}, s, r) \in C_D$ reachable in the transition system $\mathcal{T}(D)$ and a process $p \in \text{procs}$ such that $s(p) = q$.

Given a synchronous DRA $D = (Q, q_{init}, M, X, \delta)$ and a state $q_{target} \in Q$, the state reachability problem asks whether $q_{target}$ is reachable in the transition system $\mathcal{T}(D) = \langle C_D, \{c_D^{init}\}, \rightarrow_D \rangle$.

5.2 Undecidability of the State Reachability Problem for Synchronous DRA

We present in this section our findings concerning the (un)decidability of the state reachability problem for networks of synchronous DRA. We show that the problem is undecidable in the general case. The problem remains undecidable even if the DRA process has only one register. We also prove that the state reachability problem remains undecidable: (i) When we restrict the analysis to network configurations whose topology graphs are bounded and when each process in the network has at least two registers, or (ii) if we restrict the analysis to network configurations whose topology graphs are strongly bounded and if we assume that each process in the network has at least four registers.

We provide in the following an idea on the undecidability proof for each one of the mentioned three cases: The general case, the bounded restriction and the strongly bounded restriction.

5.2.1 The general case

We show in Paper II that the state reachability problem for DRA networks is undecidable in general, even if the DRA has only one register. We obtain this result through a reduction from the TRANSD problem defined in the preliminaries (Chapter 3).

Recall that an instance of the TRANSD problem is a tuple $\langle A, B, T \rangle$ composed of a transducer $T$ and of two finite-state automata $A$ and $B$ all defined over the same alphabet $\Sigma$. TRANSD asks whether there is a natural number $i \in \mathbb{N}$ such that $T^i(L(A)) \cap L(B) \neq \emptyset$, i.e. whether there exist $i \in \mathbb{N}$ and a word $w \in \Sigma^*$ accepted by $A$ such that the $i$th transduction of $w$ is also accepted by $B$.

We associate to the TRANSD instance $\langle A, B, T \rangle$ a DRA $D_{(A,B,T)}$ such that every run of the transition system
\[ \mathcal{T}(D_{(A,B,T)}) := \left\langle \mathcal{C}_{D_{(A,B,T)}}, \{c_{D_{(A,B,T)}}^{\text{init}}\}, \rightarrow_D \mathcal{C}_{D_{(A,B,T)}} \right\rangle \]

will reach a configuration that encodes a transduction chain of an arbitrary length \( m \). The notion of a transduction chain has been introduced in Section 4.3.1 of Chapter 4. For instance, Figure 5.4 shows a network configuration that encodes a transduction chain of length equal to 5. The first process in the chain (process \( p_A \)) sends letter by letter a word accepted by automaton \( A \) followed by an acceptance symbol. The last process in the chain (process \( p_B \)) traverses automaton \( B \) by reading the letters sent by process \( p_T_3 \). If process \( p_B \) receives the acceptance symbol and is currently in an accepting state of automaton \( B \), then it moves to the special state \( q_{\text{accept}} \). Every other process in the chain, i.e. processes \( p_1, \ldots, p_3 \), encodes transducer \( T \) by traversing it, receiving letters and sending their transduction. When one of these processes receives the acceptance symbol, it might forward the same symbol to the next process in the chain if the word it received so far is accepted by transducer \( T \).

We show that the instance \( \langle A, B, T \rangle \) of the TRANSD problem has a positive answer if and only if the state \( q_{\text{accept}} \) is reachable in the transition system associated to the synchronous DRA \( D_{(A,B,T)} \).

\[ \text{Figure 5.4. A DRA network configuration encoding a transduction chain of length 5.} \]

### 5.2.2 Bounded DRA

In our quest of a subclass of a dynamic register automata for which the state reachability problem becomes decidable, we considered the case of bounded DRA. Given a synchronous DRA \( D \) and a natural number \( k \), we define the \( k \)-bounded synchronous DRA transition system:

\[ \mathcal{T}(D)^{\leq k} := \left\langle \mathcal{C}_{D}^{\leq k}, \{c_{D}^{\text{init}}\}, \rightarrow_D \cap \mathcal{C}_D^{\leq k} \times \mathcal{C}_D^{\leq k} \right\rangle \]

where \( \mathcal{C}_{D}^{\leq k} \) is the set of network configuration in \( \mathcal{C}_D \) whose graph topology is \( k \)-bounded. Observe that this restriction forbids network configurations in the form of chains of arbitrary length, such as the ones used to show the undecidability of the state reachability problem in the general case. Nevertheless, we show that the problem remains undecidable even for this subclass. To prove this result, we show that a reduction from the TRANSD problem is still possible. In fact, using the rendezvous type of communication of a DRA network, we can define a DRA \( D \) that generates 2-bounded network configurations that encode transduction chains of arbitrary lengths. Figure 5.5 shows the encoding of a transduction chain of length equal to 5. While the details of this encoding...
can be found in Paper II, observe that this encoding uses a sequence of relay processes \((r_1, r_2, r_3 \text{ and } r_4 \text{ in Figure 5.5})\) responsible for relaying messages between a process and its successor in the transduction chain.

![Figure 5.5. A 2-bounded configuration encoding a transduction chain of length 5.](image)

### 5.2.3 Strongly bounded DRA

The reduction of the transduction problem to the state reachability problem is not possible anymore if we consider strongly bounded network configurations. In fact, as it is depicted in Figure 5.6, the underlying undirected graph of a network configuration is obtained by removing the direction from the edges of the graph. The longest simple path of the transduction chain from Figure 5.5 has a length equal to 1 while the longest simple path of its underlying undirected graph is of length 8.

![Figure 5.6. The underlying undirected graph of the transduction chain from Figure 5.5.](image)

Following this intuition and motivated by results from the study of Ad-Hoc networks [39, 10, 40], we examined the case of strongly bounded DRA networks in which we only consider configurations of which the underlying undirected network graph is bounded by some number \(k \in \mathbb{N}\). We show that the state reachability problem is still undecidable even for this subclass. We establish this negative result by a reduction from the state reachability problem for the two-counters Minsky Machine.

Chapter 3 recalls the definition of a two-counter Minsky machine and the associated undecidable state reachability problem. In short, a two-counter Minsky machine comes with two (unbounded) counters \(c_1\) and \(c_2\) and a finite-state machine which transitions can be either increment, decrement or test for zero of one of the two counters. Given a Minsky machine \(\mathcal{M}\) and one of its
control states $q_M$, the state reachability problem asks whether $q_M$ is reachable in the transition system induced by $\mathcal{M}$. This problem is undecidable.

The details of the reduction can be found in Paper II. However, we give here a brief overview of the reduction. Let $\mathcal{M}$ be a Minsky machine and $q_M$ one of its control states. We associate to $\mathcal{M}$ and to $q_M$ a DRA process $D_{(\mathcal{M}, q_M)}$ designed in such a way that its transition system generates configurations composed of a controller process $p_\mathcal{M}$, two counter processes $p_{c_1}$ and $p_{c_2}$, and a set of satellite processes $p_{\text{sat}}$. The controller process $p_\mathcal{M}$ simulates the state transitions of the Minsky machine $\mathcal{M}$ and send the increment, decrement and test for zero instructions to the corresponding counter process. Furthermore, satellite processes can either be in state # or in state idle. The number of satellite processes in state # attached to each counter process encodes the value of the corresponding counter.

Figure 5.7 shows how the incrementation and decrementation transitions are encoded. Notice $c_1 = 2, c_2 = 1$ in the right hand-side configuration in the figure. The incrementation of counter $c_1$ results in the creation of a new satellite process attached to $p_{c_1}$ labeled with #. The decrementation of counter $c_2$ results in the transition of one of the satellite processes attached to $p_{c_2}$ from state # to idle. This satellite process can later reset the register in which the ID of $p_{c_2}$ is stored.

Assume that the bound on the underlying undirected graph of the network configurations is equal to 4. To encode the test for zero, we use the bound on the underlying undirected graph of the configuration graph. For instance, assume that $\mathcal{M}$ has a test for zero of counter $c_2$. This transition is encoded in $D_{(\mathcal{M}, q_M)}$ through two steps: (i) The controller $p_\mathcal{M}$ requests process $p_{c_2}$ to create a new satellite process in state test, and (ii) $p_{c_2}$ sends the ID of $p_\mathcal{M}$ to the newly created process in state test. Observe that the second step is only allowed if there are no other satellite processes attached to $p_{c_2}$, i.e. if the value of counter $c_2$ is zero. Otherwise, the network configuration graph would contain the simple path # $\rightarrow$ $p_{c_2}$ $\rightarrow$ test $\rightarrow$ $p_\mathcal{M}$ $\rightarrow$ $p_{c_1}$ $\rightarrow$ # which has a length equal to 5, strictly bigger than the bound 4.
Figure 5.8 shows two configurations on which an attempt of the test for zero has been made. In the first case (upper-left configuration), process $p_{c_2}$ has no satellite processes in state # attached to it, encoding the fact that counter $c_2$ is equal to zero. In this case, the two steps encoding the test for zero can take place and the resulting configuration graph is still 4-strongly bounded. In the second case (lower-left configuration), process $p_{c_2}$ has one satellite process attached in state #, encoding the fact that counter $c_2 = 1$. In that case, the second step encoding the test for zero cannot take place because this would result in a 5-strongly bounded configuration not allowed by the considered bound 4.

![Figure 5.8. Encoding of the test for zero in 4-strongly bounded networks.](image)

5.3 Degenerative and (Strongly) Safe Synchronous DRA

We present in this section our (positive) results on the decidability of the state reachability problem for some interesting subclasses of synchronous DRA. More precisely, we introduce in this section the classes of degenerative synchronous DRA and (Strongly) safe synchronous DRA. We start by defining these classes, mentioning our results and how we established them. We also summarize in a table our findings concerning the decidability of the state reachability problem for the synchronous DRA. Finally, we give in details the proof of the decidability of the state reachability problem for strongly safe DRA.
5.3.1 Definitions and results

A synchronous DRA $D = \langle Q, q_{init}, M, X, \delta \rangle$ is degenerative if for every state $q \in Q$ and for every register $x \in X$ there is a register reset transition $\langle q, \text{reset}(x), q \rangle \in \delta$. Processes in a degenerative DRA network can reset any of their registers at any time. We associate to every synchronous DRA $D = \langle Q, q_{init}, M, X, \delta \rangle$ its degenerative counterpart defined by:

$$\text{Deg}(D) = \langle Q, q_{init}, M, X, \delta \cup \{q, \text{reset}(x), q \mid q \in Q, x \in X\} \rangle$$

Informally, the degenerative counterpart of a DRA is the same DRA in which register reset transitions have been added for every state and for every register.

We show in Paper II that a DRA and its degenerative counterpart reach the same set of control states. Consequently, the state reachability problem for degenerative DRA remains undecidable. Furthermore, the encoding we use in order to reduce the TRANSD problem into the state reachability problem for (simply) bounded DRA (see Figure 5.5) remains valid when we consider degenerative DRA. Hence, the state reachability for the bounded degenerative DRA is undecidable. This is not the case anymore for the class of strongly bounded degenerative DRA. Indeed, we show in Paper II that the state reachability problem is decidable for the class of strongly bounded degenerative DRA. We show this positive result through a reduction to the coverability problem of a Well-Structured Transition System (WSTS for short) that we instantiate. Furthermore, we find a lower bound (non-primitive recursive) for the problem through a reduction from lossy counter machines.

We also consider in Paper II the class of $k$ (strongly) safe DRA, in which a DRA only generates $k$ (strongly) bounded network configurations. Since a DRA and its degenerative counterpart reach the same set of states, we get as a corollary of the previous results that the state reachability problem is:

- Decidable and non-primitive recursive for $k \geq 4$ strongly safe DRA with at least two registers, and
- undecidable for $k \geq 2$ safe DRA with at least two registers.

Finally, our last result for synchronous DRA concerns the particular case of $k$ safe DRA equipped with one register. We show that this class of DRA is $2 \times k$-strongly safe. By consequence, the state reachability problem for a degenerative $k$ safe DRA is decidable and non-primitive recursive.

We summarize our results regarding synchronous DRA in Table 5.1. Details of the proofs can be found in Paper II. Nevertheless, we give hereafter an overview of the reduction of the state reachability problem for $k$ strongly bounded degenerative and of the instantiation of the WSTS.

5.3.2 Decidability for strongly bounded degenerative DRA

We show that the state reachability problem is decidable for the class of strongly bounded degenerative DRA. We show this positive result through a
<table>
<thead>
<tr>
<th>DRA</th>
<th>Non-Degenerative</th>
<th>Degenerative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded</td>
<td>undecidable</td>
<td>undecidable</td>
</tr>
<tr>
<td>Strongly bounded</td>
<td>undecidable</td>
<td>decidable</td>
</tr>
<tr>
<td>Safe</td>
<td>undecidable</td>
<td>undecidable</td>
</tr>
<tr>
<td>I register &amp; Safe</td>
<td>decidable</td>
<td>decidable</td>
</tr>
</tbody>
</table>

Table 5.1. State reachability for different subclasses of synchronous DRA.

reduction to the coverability problem of a Well-Structured Transition System $\mathcal{T}_{WSTS}$ that we instantiate.

Definition of the WSTS and the reduction.
Let $D = \langle Q, q_{\text{init}}, M, X, \delta \rangle$ be a degenerative DRA, $q_{\text{target}} \in Q$ be one of its control states and $k \in \mathbb{N}$ be a natural number. We use $C^k_D$ to denote the set of all possible $k$-strongly bounded network configurations of the DRA $D$, i.e. the set of all possible network configurations of the DRA $D$ which underlying undirected graph is bounded by $k$. We define $\rightarrow^{\square}_D := \rightarrow_D \cap C^k_D \times C^k_D$ as the transition relation $\rightarrow_D$ of the DRA $D$ restricted to the set of $k$-strongly bounded network configurations. We are interested in the $k$-strongly bounded reachability of state $q_{\text{target}}$. Formally, we would like to know whether the state $q_{\text{target}}$ is reachable in the transition system:

$$\mathcal{T}^\square_D := \langle C^k_D, \{c_{\text{init}}^D \}, \rightarrow_D \rangle$$

As mentioned in the preliminaries, a WSTS is a transition system $\mathcal{T}_{WSTS} = \langle C^D_{WSTS}, C^\text{init}_{WSTS}, \rightarrow_{WSTS} \rangle$ together with an ordering $\sqsubseteq$ defined over the set of configurations $C^D_{WSTS}$. The ordering $\sqsubseteq$ should be a WQO and the transition relation $\rightarrow_{WSTS}$ should be monotonic with respect to that ordering. Given a configuration $c_{\text{bad}} \in C^D_{WSTS}$, the coverability problem asks whether there exists a configuration $c \in C^D_{WSTS}$ reachable in $\mathcal{T}_{WSTS}$ such that $c_{\text{bad}} \sqsubseteq c$. Abdulla and Jonsson give in [6] a number of sufficient conditions in order to show the decidability of the coverability problem.

We set $C^D_{WSTS} := C^k_D$ and $C^\text{init}_{WSTS} := \{c_{\text{init}}^D \}$. Let $\rightarrow_{D}^{\text{reset}}$ be the reset transition relation induced by all the register self-reset transitions of the degenerative DRA $D$, i.e the transition relation induced by the set of transitions $\{\langle s_1, \text{reset} \langle x \rangle, s_1 \rangle \mid s_1 \in Q, x \in X \}$. We define:

$$\rightarrow_{WSTS} := \rightarrow_D^{\square} \circ (\rightarrow_D^{\text{reset}})^*$$

A transition from $\rightarrow_{WSTS}$ can be decomposed into a number of reset transitions followed by a transition from $\rightarrow_D^{\square}$. Thus, the transition system $\mathcal{T}_{WSTS}$ is defined by:

$$\mathcal{T}_{WSTS} := \langle C^k_D, \{c_{\text{init}}^D \}, \rightarrow_D^{\square} \circ (\rightarrow_D^{\text{reset}})^* \rangle$$

84
Because $D$ is a degenerative DRA, we have that $\xrightarrow{\text{reset}}_D \subseteq \xrightarrow{\square}^k_D$. Using this fact, we show that the reachability of $q_{\text{target}}$ in the transition system $\mathcal{T}_D^\square$ is equivalent to the reachability of $q_{\text{target}}$ in the transition system $\mathcal{T}_{\text{WSTS}}$.

We show in the following two paragraphs that $\mathcal{T}_{\text{WSTS}}$ is indeed a WSTS by proving that $\longrightarrow_{\text{WSTS}}$ is monotonic w.r.t. a (well-quasi) ordering that we define over the set of configurations $C^\square_D$. Finally, we prove in Paper II the sufficient conditions listed in [6] needed to show the decidability of the coverability problem.

**Ordering.**

In order to show that $\mathcal{T}_{\text{WSTS}}$ is in fact a WSTS, we need first to define an ordering over the set of $k$-strongly bounded configurations. We use for that purpose the embedded subgraph relation $\subseteq_{\text{sub}}$ (defined in the preliminaries in Chapter 3) over the set of directed labeled graphs. More precisely, we define the ordering $\sqsubseteq$ over the set of configurations $C_{\text{WSTS}}$ as follows: For two configurations $c_1, c_2 \in C_{\text{WSTS}}$, $c_1 \sqsubseteq c_2$ if and only if $\text{enc}(c_1) \subseteq_{\text{sub}} \text{enc}(c_2)$.

Furthermore, using the graph transformation shown in Figure 5.9, we encode a configuration $c$ into an undirected graph that we denote by $\text{enc}_2(c)$. The two encodings $\text{enc}$ and $\text{enc}_2$ are equivalent, in the sense that $\text{enc}(c_1) \subseteq_{\text{sub}} \text{enc}(c_2)$ if and only if $\text{enc}_2(c_1) \subseteq_{\text{sub}} \text{enc}_2(c_2)$. Observe that, if a configuration is $k$-strongly bounded, then its encoding $\text{enc}_2$ into an undirected graph is $3 * k$-bounded. This will allow us to use the result by Ding in [42] stating that the embedded subgraph relation is a WQO over the set of bounded undirected graphs. Consequently, $\sqsubseteq$ is also a WQO over the set of $k$-strongly bounded configurations $C_{\text{WSTS}}$.

![Figure 5.9. Encoding of a configuration into an undirected graph.](image)

We use $c_{\text{target}}$ to denote the configuration that contains (exactly) one process with empty registers in state $q_{\text{target}}$. The coverability of configuration $c_{\text{target}}$ in $\mathcal{T}_{\text{WSTS}}$ with respect to the ordering $\sqsubseteq$ means that there is a configuration $c \in C_{\text{WSTS}}$ reachable in the transition system $\mathcal{T}_{\text{WSTS}}$ such that $c_{\text{target}} \sqsubseteq c$. Given the definition of $\sqsubseteq$ as the embedded subgraph ordering, $c_{\text{target}} \sqsubseteq c$ means that $c$
contains a process in state $q_{\text{target}}$. Thus, the definitions of state and reachability and coverability coincide.

**Monotonicity.**

We need to show that the transition relation $\rightarrow_{\text{WSTS}}$ is monotonic with respect to the WQO $\sqsubseteq$.

Assume that $c_1, c_2, c_3 \in C_{\text{WSTS}}$ are three $k$-strongly bounded DRA configurations such that $c_1 \sqsubseteq c_3$, i.e. $\text{enc}(c_1) \sqsubseteq_{\text{sub}} \text{enc}(c_3)$, and $c_1 \rightarrow_{\text{WSTS}} c_2$. In order to show the monotonicity of $\rightarrow_{\text{WSTS}}$ wrt. $\sqsubseteq$, it suffices to define a configuration $c_4$ such that $c_2 \sqsubseteq c_4$ and $c_3 \rightarrow_{\text{WSTS}} c_4$.

Since $\rightarrow_{\text{WSTS}} = \mathbin{\square}^k_D \circ \left( \mathbin{\xrightarrow{\text{reset}}} D \right)^*$, then $c_1 \rightarrow_{\text{WSTS}} c_2$ means that there are $r \geq 0$ and $c_1^0 \in C_D^k$ such that $c_1 \left( \mathbin{\xrightarrow{\text{reset}}} D \right)^r c_1^0$ and $c_1^0 \rightarrow_{\text{D}}^k c_2$. Moreover, since $c_1^0$ is obtained from $c_1$ after resetting some registers and since $\text{enc}(c_1) \sqsubseteq_{\text{sub}} \text{enc}(c_3)$, then we also have that $\text{enc}(c_1^0) \sqsubseteq_{\text{sub}} \text{enc}(c_3)$. Therefore, it is possible to obtain from $c_3$ a configuration $c_3^o$ after resetting a number of registers $r'$ such that $c_3^o$ contains a configuration $c_{\text{sub}}$ identical to $c_1^0$ together with a number of isolated processes, i.e. processes that are disconnected from the rest of the network. Finally, observe that any transition $t \in \rightarrow_{\text{D}}^k$ that $c_1^0$ could take in order to obtain $c_2$ can also be performed by the configuration $c_{\text{sub}}$, identical to $c_1^0$, to obtain a configuration $c_{\text{sub}}'$ identical to $c_2$. Consequently, configuration $c_3^o$, which contains $c_{\text{sub}}$, can take the same transition $t$ and result in a configuration $c_4$ such that $\text{enc}(c_2) \sqsubseteq_{\text{sub}} \text{enc}(c_4)$. Figure 5.10 depicts the case where $r = 1$, $r' = 4$ and where the transition performed by $c_1^0$ is a sending and a receiving of an ID.

### 5.4 Buffered Dynamic Register Automata

The model of Dynamic Register Automata is inspired by the model of Dynamic Communicating Automata (DCA for short) introduced in [23, 24]. The (synchronous) DRA model considered in Paper II and presented in the first part of this chapter is based on a rendezvous type of communication. The communication used by the original model, the DCA, is asynchronous, i.e., messages sent by a process are not instantaneously received by a process. Instead, the message is stored in a queue (or buffer) and can be read by the process in a first-in-first-out manner.

We consider in the remaining part of this chapter the model of Buffered Dynamic Register Automata (Buffered DRA or BDRA for short), a model closer to the original Dynamic Communicating Automaton. More precisely, every process in a BDRA network is equipped with a mailbox, a queue in which messages addressed to the process are stored in a FIFO manner. Besides the operations of process creation and sending and receiving messages and IDs, we assume that processes in a buffered DRA network can also disconnect
In this section, we give the formal definition of a BDRA, its associated network configuration and the semantics of each one of its possible actions. In the following and last section of this chapter, we define the state reachability problem for buffered DRA and present our findings concerning its decidability.

**BDRA Definition.**

The syntax of a buffered DRA only differs from the syntax of a (synchronous) DRA in the type of actions that label each state transition. More precisely, a BDRA $D$ is given by a tuple $\langle Q, q_{init}, M, X, \delta \rangle$ where $Q$ is the finite set of states, $q_{init}$ is the initial state, $M$ is a finite set of messages, $X$ is a finite set of registers, and $\delta$ is the transition relation. Moreover, every transition $t \in \delta$ is of the form $\langle s_1, \text{action}, s_2 \rangle$, where $s_1, s_2 \in Q$ and action can be either a local action $\tau$, a create action $x \leftarrow \text{create}(q)$ where $x \in X$ and $q \in Q$, a send action $m \langle v \rangle \ll x$ where $m \in M, v \in \{?, \text{self}\} \cup X$ and $x \in X$, a receive action $m?x$ where $m \in M$ and $x \in X$, or a disconnect action disconnect.
Let $\mathcal{P}$ be the domain of process names and $D = \langle Q, q_{init}, M, X, \delta \rangle$ be a BDRA. A network configuration of processes defined from $D$ is given by the tuple $\langle \text{procs}, s, r, \text{ch} \rangle$ where $\text{procs} \in \mathcal{P}$ is the set of processes, $s : \mathcal{P} \rightarrow Q$ is a partial function that maps each process $p \in \text{procs}$ to its current state, $r : \mathcal{P} \rightarrow \{ X \rightarrow \text{procs} \}$ is a partial function that maps every process $p \in \text{procs}$ to the content of its registers, and $\text{ch} : \mathcal{P} \rightarrow (M \times (\mathcal{P} \cup \{ \perp \}))^*$ maps each process $p \in \text{procs}$ to the content of its mailbox, a word in $(M \times (\mathcal{P} \cup \{ \perp \}))^*$. We use $C_D$ to denote the set of all possible network configurations defined from the buffered DRA $D$. Moreover, we define the initial configuration $c_D^{\text{init}}$ as the configuration containing one single process $p \in \text{procs}$ in state $q_{init}$ and whose registers and mailbox are empty.

Figure 5.11 shows $c_{\text{example}}$, an example of BDRA configuration, and its graph representation, $\text{rep}(c_{\text{example}})$. Similarly to the case of synchronous DRA, a process $p$ in the configuration of a buffered DRA is encoded using a vertex labeled with its current state. Moreover, the non-empty content of a register $x$ of process $p$, which contains the ID of some process $p'$, is represented using an edge from $p$ to $p'$ labeled with $x$. In the case of buffered DRA, we also need to represent the content of the buffer of every process in the configuration. For visibility reasons, we use colors instead of natural numbers to represent process IDs.

In order to reason about present and future communication capabilities of a network configuration, we need to be able to graphically represent the content of the buffers. Therefore, we define an alternative graph representation $\text{rep}_2$, which differs from $\text{rep}$ in the way buffers are encoded. Every message $m\langle p' \rangle$ contained in the buffer of some process $p$ of a configuration $c$ is represented in the graph encoding $\text{rep}_2(c)$ by a vertex $v$.
labeled with the symbol $\rightarrow$, an edge from the vertex representing $p$ to $v$ and an edge from $v$ to the vertex representing $p'$. Figure 5.12 shows the encoding $\text{rep}_2(c_{\text{example}})$ of the configuration $c_{\text{example}}$ shown in Figure 5.11. The representation $\text{rep}_2$ precisely tells about the current state of the processes, the content of their registers, and the IDs contained in the buffer of each process. We use this representation to define the topology of the network configuration. On the other hand, the representation $\text{rep}_2$ does not inform about the labels ($m_1, m_2, \ldots$) of the messages and the order between them in the mailbox. We define later a more precise graph encoding when we only consider configurations in which buffers are bounded by some constant.

Meanwhile, we use the graph encoding $\text{rep}_2$ to define the set of (strongly) bounded BDRA network configurations. More precisely, a BDRA configuration $c$ is $k$ (strongly) bounded for some $k \geq 0$ if its graph encoding is $k$ (strongly) bounded.

Using the graph representation $\text{rep}$, we illustrate in the following paragraphs the semantics of each BDRA action. We assume that $D = \langle Q, q_{\text{init}}, M, X, \delta \rangle$ is a buffered DRA and that $\langle \text{procs}, s, r, ch \rangle$ is a configuration from $C_D$.

**Process creation.**

Let $t = \langle s_1, x \leftarrow \text{create}(s_3), s_2 \rangle \in \delta$ be a process creation transition where $s_1, s_2, s_3 \in Q$ and $x \in X$.

Assume there is $p \in \text{procs}$ such that $s(p) = s_1$. Process $p$ can then create a new process $p' \not\in \text{procs}$ in state $s_3$, move to state $s_2$ and save the (new) ID of $p'$ in its register $x$. The resulting network configuration is given by

$$\langle \text{procs} \cup \{p'\}, s[p' \leftarrow s_3][p \leftarrow s_2], r[p \leftarrow r(p)[x \leftarrow p']], ch \rangle$$

**Message sending.**

Let $t = \langle s_1, m \langle v \rangle!x, s_2 \rangle \in \delta$ be a message sending action with $m \in M$ a message, $s_1, s_2$ two states from $Q$, $x \in X$ a register and $v \in \{\bot, \text{self}\} \cup X$.

Assume that there exists $p \in \text{procs}$ such that $s(p) = s_1$ and $r(p)(x) = p'$ where $p' \in \text{procs}$ is another process. In this case, $p$ can send the message $m \langle v \rangle$ to $p'$ by appending the message to the mailbox of $p'$ while moving from state $s_1$ to $s_2$. If $v = \bot$, then the message sent is $m \langle \bot \rangle$. If $v = \text{self}$, then the message sent is $m \langle p \rangle$. Otherwise, if $v$ is a register $y \in X$, then the message sent is $m \langle v \rangle$ where $v$ is the content of register $y$ of process $p$. 

Figure 5.12. Graph encoding of $c_{\text{example}}$. 

89
The new configuration is defined as follows. The new state of process $p$ is $s_2$ while the channel of process $p'$ is appended with the new message. For instance, in the case where $v$ is a register $y \in X$ that contains the ID of another process $p''$, then the resulting configuration is:

$$\langle \text{procs}, s[p \leftarrow s_2], r, \text{ch}[p' \leftarrow m \langle p'' \rangle \cdot \text{ch}(p')] \rangle$$

Figure 5.13 illustrates this case.

---

**Message receiving.**

Let $t = \langle s_1, m!x, s_2 \rangle \in \delta$ be a message receiving action with $m \in M$ a message, $s_1, s_2$ two states from $Q$ and $x \in X$ a register.

Assume that there exists $p \in \text{procs}$ such that $s(p) = s_1$ and such that $\text{ch}(p) = w.m\langle v \rangle$ where $v \in \text{procs} \cup \{\bot\}$ and $w \in (M \times (\mathcal{P} \cup \{\bot\}))*$. In this case, $p$ can read the message $m\langle v \rangle$ from its mailbox by setting its register $x$ to the value $v$ and moving from state $s_1$ to $s_2$. The new configuration is defined as follows: Process $p$ moves to state $s_2$, message $m\langle v \rangle$ is removed from the buffer of $p$ and register $x$ in $p$ is updated to the value $v$.

For instance, if $v = \bot$, then the new configuration is defined by:

$$\langle \text{procs}, s[p \leftarrow s_2], r[p \leftarrow r(p)[x \leftarrow \bot]], \text{ch}[p \leftarrow w] \rangle$$

This particular case is illustrated in Figure 5.14. Observe that the old value of the recipient register $y$ is lost.

---

**Process disconnection.**

Let $t = \langle s_1, \text{disconnect}, s_2 \rangle \in \delta$ be a process disconnection action where $s_1, s_2$ are two states from $Q$. 

90
Assume that there is $p \in \text{procs}$ such that $s(p) = s_1$. Then process $p$ can perform the disconnect action and get disconnected from the rest of the network. Concretely, by performing a disconnect action, a process loses every information concerning the rest of the network while every other process in the network loses every information concerning the disconnected process. More precisely, the new configuration is defined as follows: (a) Process $p$ moves to state $s_2$, (b) all registers of $p$ are reset to $\bot$, (c) the mailbox of $p$ is emptied, (d) every register $x$ of every other process $p' \in \text{procs}$ which value was equal to $p$ is reset to $\bot$, and (e) every message $m \langle p \rangle$ in every other process $p' \in \text{procs}$ mailbox is replaced with $m \langle \bot \rangle$.

Figure 5.14 gives an example of how such action transforms a configuration. Notice that, after performing the disconnect action, the process (in green) becomes isolated from the rest of the network in the resulting configuration. In fact, it has no knowledge of other processes, and other processes in the configuration have no knowledge of this process. Thus, no further communication can happen between the disconnected process and the rest of the network.

### 5.5 State Reachability Problem for Buffered DRA

Let $D = \langle Q, q_{init}, M, X, \delta \rangle$ be a BDRA. We use $\rightarrow_D$ to denote the transition relation induced by every possible transition from $\delta$. More precisely, if $c_1$ and $c_2$ are two configurations from $C_D$ such that $c_2$ can be obtained from $c_1$ after some process in $c_1$ executes some transition $t \in \delta$, then $\langle c_1, c_2 \rangle \in \rightarrow_D$ and we write $c_1 \rightarrow_D c_2$.

We associate to $D$ the transition system $T_D := \langle C_D, \{c_D^{init}\}, \rightarrow_D \rangle$ where $C_D$ is the set of configurations, $\{c_D^{init}\}$ is the singleton containing the initial configuration and $\rightarrow_D$ is the transition relation defined earlier.
Consider now some state $q_{\text{target}} \in Q$. We say that $q_{\text{target}}$ is reachable by $D$ if there exists a configuration $c = (\text{procs}, s, r, ch)$ reachable in $T$ such that one of its processes $p \in \text{procs}$ is in state $q_{\text{target}}$, i.e. $s(p) = q_{\text{target}}$. Given a BDRA $D$ and one of its states $q_{\text{target}}$, the state reachability problem asks whether $q_{\text{target}}$ is reachable.

We present in this section our findings concerning the decidability of the state reachability problem for buffered DRA. We start by showing that this problem is undecidable in the general case and for some subclasses of the BDRA model. In particular, we show that the problem is undecidable for the subclass of BDRA in which we only consider configurations whose graph representation is strongly bounded and where buffers have a bounded size. Then, we define the subclass of lossy BDRA in which processes can disconnect themselves from the rest of the network at any point in time. We show that the state reachability problem becomes decidable for the subclass of bounded-buffer strongly-bounded lossy BDRA through a non-trivial instantiation of the WSTS framework. Finally, we show that the state reachability problem is also decidable for the subclass of bounded buffer and strongly bounded acyclic BDRA, in which we only consider configurations where communication cycles are absent.

5.5.1 Undecidability results.
Unsurprisingly, the state reachability problem for BDRA is undecidable in the general case. In fact, the state reachability problem is undecidable as soon as a cycle is allowed in the topology of the configuration networks generated by a BDRA. We can show this negative result through a reduction from the halting problem of a two-counters Minsky machine. Let $\mathcal{M}$ be a Minsky machine equipped with two counters $c_1$ and $c_2$. We encode the two-counters Minsky machine into a BDRA network composed of two processes $p_1$ and $p_2$. 

![Figure 5.15. Disconnection of a process.](image-url)
The mailbox of process $p_1$ encodes the value of counter $c_1$ and the buffer of process $p_2$ encodes the value of counter $c_2$. Moreover, process $p_1$ simulates the operation of incrementation of counter $c_2$ by sending a message to $p_2$. The decrementation of counter $c_1$ is performed by a reception (i.e. reading) of a message by process $p_1$ from its own mailbox. Finally, process $p_1$ tests for zero counter $c_1$ by checking that the delimiters # are directly successive. Process $p_2$ encodes in a similar manner the operations of incrementing $c_1$, decrementing $c_2$ and testing for zero $c_2$. Figure 5.16 shows the encoding of the values $c_1 = 2, c_2 = 3$.

The previous result shows that allowing cycles in the network topology and an unlimited buffer capacity makes the state reachability problem undecidable. We show in Paper III that even if we disallow these two features, i.e., if we forbid communication cycles and if we bound the capacity of every buffer in the network, the problem remains undecidable. In fact, it is possible to reduce the transduction problem TRANSD mentioned in the preliminaries (Chapter 3) into the state reachability problem for a bounded buffer BDRA where only acyclic network configurations are considered. Although details of the reduction can be found in the paper, Figure 5.17 shows for example how a transduction chain (of length equal to 5) is encoded for one instance $\langle A, B, T \rangle$ of the TRANSD problem.
For the reduction from the \textsc{Transd} problem to work, it is necessary for the length of the transduction chain to be unbounded. Therefore, and inspired by our previous works on Ad-Hoc networks (Paper I) and on synchronous DRA (Paper II), we also prove that the state reachability problem remains undecidable if we restrict the analysis:

- To network configurations whose graph representation $\mathbf{rep}_2$ is strongly bounded by 1 and where buffers are unbounded. We obtain this result through a reduction from the \textsc{Transd} problem. As depicted in Figure 5.18, instead of transmitting each word in the transduction chain letter by letter, the word is transmitted completely to the next process in the chain.

- To network configurations whose graph representation $\mathbf{rep}_2$ is bounded by 2 and where buffers may contain at most one message. This result is obtained by a reduction from the transduction problem similar to the case of synchronous DRA as depicted in Figure 5.5. In this encoding, relay processes $r_i$ are used to allow a process $p_i$ to send a message to process $p_{i+1}$, its successor in the transduction chain. In the case of synchronous DRA, rendezvous communication is used to ensure that the message sent by process $p_i$ to the relay process $r_i$ is identical to the message sent by process $p_{i+1}$ to $r_i$. In order to ensure this, instead of rendezvous communication, we use in the case of buffered DRA two components. The first component is the fact that the buffer has a maximal capacity of one message. This constraint ensures a message-by-message simulation of the transduction chain. The second component consists in allowing process $p_{i+1}$ to confirm the well-reception of the first message by resending the same message to the relay.

- To network configurations whose graph representation $\mathbf{rep}_2$ is strongly bounded by 4 and where buffers may contain at most one message. This result is obtained through a reduction from the state reachability problem for two-counters Minsky machines. The encoding is similar to the case of synchronous DRA as depicted in Figures 5.7 and 5.8.

Table 5.2 summarizes our undecidability results.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Encoding of a transduction chain of length equal to 5. The graph representation of the configurations are 1-strongly bounded while process buffers are unbounded.}
\end{figure}
### Table 5.2. List of BDRA subclasses for which the state reachability problem is undecidable.

<table>
<thead>
<tr>
<th>Network topology</th>
<th>Buffer</th>
<th># of registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Acyclic but unbounded</td>
<td>1-Bounded</td>
<td>≥ 1</td>
</tr>
<tr>
<td>(b) Acyclic, 1-Strongly Bounded</td>
<td>Unbounded</td>
<td>≥ 1</td>
</tr>
<tr>
<td>(c) 2-Bounded</td>
<td>1-Bounded</td>
<td>≥ 2</td>
</tr>
<tr>
<td>(d) 4-Strongly Bounded</td>
<td>1-Bounded</td>
<td>≥ 2</td>
</tr>
</tbody>
</table>

5.5.2 Lossy BDRA

A BDRA \( D = (Q, q_{\text{init}}, M, X, \delta) \) is lossy if there is a disconnect transition \( \langle q, \text{disconnect}, q \rangle \) in the transition relation \( \delta \) of \( D \) for each state \( q \in Q \). Informally, any process in a lossy BDRA network can, at any time during the run of the network, disconnect itself and forbid any further communication between itself and the rest of the network.

The state reachability problem for lossy BDRA remains undecidable for the subclasses (a), (b) and (c) listed in Table 5.2. In fact, the ability of processes to disconnect does not invalidate the transduction chain constructions. This is not the case for the two counters Minsky machine reduction used in the class of lossy BDRA where only strongly bounded network configurations with bounded buffers are considered. The state reachability problem is, in fact, decidable for this class. We show this positive by reducing the state reachability problem to the coverability problem of a WSTS. Although details of the proof can be found in Paper III, we show in the remaining of this section the main components of the proof. We start in the first paragraph by defining the transition system which we prove later to be a WSTS. Then we define a new graph representation for strongly bounded buffered DRA network configurations whose buffers are bounded. We use this graph representation to define the ordering over the set of configurations that we prove to be a WQO. We show that the transition relation of the WSTS is monotonic wrt. the ordering that we have defined. Finally, we recall the main aspects of the reduction and conclude.

The (well-structured) transition system.

Let \( D = (Q, q_{\text{init}}, M, X, \delta) \) be a lossy BDRA and \( l, k \geq 0 \) be two natural numbers. We use \( C_{D}^{l \leq l, \leq k} \) to denote the set of \( l \)-bounded buffer, \( k \)-strongly-bounded network topology configurations. We define \( \longrightarrow_{D}^{l \leq l, \leq k} := \longrightarrow_{D} \cap C_{D}^{l \leq l, \leq k} \times C_{D}^{l \leq l, \leq k} \) to be the transition relation induced by \( \delta \) and restricted to the set of \( l \)-bounded buffer, \( k \)-strongly-bounded network topology configurations. We use \( \text{disconnect} \) to denote the transition relation induced by all the set of self disconnect transitions from \( \delta \), i.e. the transitions \( \{ \langle s_{1}, \text{disconnect}, s_{1} \rangle \in \delta \} \). We define the transition system \( \mathcal{T}_{\text{WSTS}} = \langle C_{\text{WSTS}}, C_{\text{WSTS}}^{\text{init}}, \longrightarrow_{\text{WSTS}} \rangle \) where \( C_{\text{WSTS}} := C_{D}^{l \leq l, \leq k} \),
\(C_{WSTS}^{\text{init}} = \{c_D^{\text{init}}\}\) is the singleton containing the initial configuration, and 
\(\rightarrow_{WSTS} = (\rightarrow_D^{\leq l, \leq k}) \circ (\text{disconnect})^*\). A transition from \(\rightarrow_{WSTS}\) is the composition of an unlimited number of self-disconnect transitions followed by a transition from \(\rightarrow_D^{\leq l, \leq k}\).

**Graph representation.**

Let \(c = \langle \text{procs}, s, x, ch \rangle \in C_{WSTS}\) be an \(l\)-bounded buffer, \(k\)-strongly bounded network configuration from \(C_{WSTS}\). We associate to \(c\) the graph encoding \(\text{enc}_{\leq l}(c)\) defined as follows. The graph representations \(\text{enc}_{\leq l}\) and \(\text{rep}_2\) are identical in the way process states and register contents are represented. In fact, every process \(p \in \text{procs}\) in configuration \(c\) is represented in the graph encoding \(\text{enc}_{\leq l}\) using a vertex \(v_p\) labeled with \(s(s)\), the state of \(p\). Moreover, if not empty, every register \(x\) of \(p\) is encoded using an edge labeled with \(x\) from vertex \(v_p\) to \(v_{p'}\), the vertex representing the process \(p'\) pointed by \(x\). The only difference between the two graph representations \(\text{enc}_{\leq l}\) and \(\text{rep}_2\) resides in the way buffers are encoded.

Since \(c\) is an \(l\)-bounded buffer configuration, every process \(p\) in \(c\) has a buffer of length \(n_p = |\text{ch}(p)|\) such that \(0 \leq n_p \leq l\). Moreover, for every process \(p\) in configuration \(c\), the content of the buffer of process \(p\) is a word of the form \(\text{ch}(p) = m_{n_p} \langle id_{n_p} \rangle \ldots . m_2 \langle id_2 \rangle \cdot m_1 \langle id_1 \rangle\) where, for every \(i : 1 \leq i \leq n_p\), \(m_i \langle id_i \rangle\) is a pair composed of a message \(m_i\) from \(M\) and an ID \(\langle id_i \rangle\). The ID \(id_i\) can be either empty (i.e. \(id_i = \bot\)) or can be the ID of another process \(p' \in \text{procs}\). The index \(i\) in every message \(m_i \langle id_i \rangle\) represents the position of the message in the buffer. In particular, \(m_1 \langle id_1 \rangle\) is at the head of the queue, i.e. it is the message that will be received first from the buffer. On the other hand, \(m_{n_p} \langle id_{n_p} \rangle\) is at the tail of the queue.

To represent the content of this buffer, we proceed in two steps. First, we encode every message \(m_i \langle id_i \rangle\) using (i) a vertex \(v_{p,i}\) labeled with \(\langle m_i, i \rangle\), i.e. a pair encoding the message its position in the buffer, (ii) an edge from \(v_p\), the vertex representing \(p\), to \(v_{p,i}\), and (iii) if \(id_i = p'\) for some \(p' \in \text{procs}\), an edge from \(v_{p,i}\) to \(v_{p', i}\), i.e. the vertex that represents process \(p'\). Second, we represent every empty message slot in the buffer by (i) a vertex \(v_{p,i}\) labeled with \(\langle e, i \rangle\), and (ii) an edge from \(v_p\), the vertex representing \(p\), to \(v_{p,i}\). Observe that the position index of an empty message slot \(i\) is such that \(n_p < i \leq l\).

An example of a configuration and its encoding are given in Figure 5.19.

The buffer of each one of the processes in the configuration has a capacity of two messages. Therefore, we can encode this configuration using a 2-bounded buffer graph encoding \((l = 2)\). For instance, the buffer of the process in red (in state \(s_3\)) contains only one message, the message \(m_3\) with the ID of the process in blue (in state \(s_2\)). This message is represented in the graph representation \(\text{enc}_{\leq 2}\) by a vertex labeled with the pair \(\langle m_3, 1 \rangle\) composed of the message \(m_3\) and its position in the queue of the buffer (1 for the first position). Moreover, we use an edge from the process in red (owner of the buffer) to the
message-vertex and an edge from the message-vertex to the process in blue (corresponding to the ID in the message). Observe that empty message slots are also encoded using message-vertex labeled with a pair composed of \( \varepsilon \) (to denote that the message slot is empty) and the position of the empty message slot in the queue.

![Graph encoding](image.png)

**Figure 5.19.** A BDRA network configuration and its graph encoding.

**Well-quasi ordering.**

We use the induced subgraph relation \( \sqsubseteq_{\text{ind}} \) (defined in Chapter 3) in order to define \( \sqsubseteq_{\text{buf}} \), the ordering over the set of \( l \)-bounded buffer, \( k \)-strongly bounded network configurations. Let \( c_1, c_2 \in \mathcal{C}_{\text{WSTS}} \) be two such configurations. We have that \( c_1 \sqsubseteq_{\text{buf}} c_2 \) if and only if \( \text{enc}_{\leq l}(c_1) \sqsubseteq_{\text{ind}} \text{enc}_{\leq l}(c_2) \).

Let \( c_1, c_2 \in \mathcal{C}_{\text{WSTS}} \) be two \( k \)-strongly bounded configurations whose buffers are \( l \)-bounded. If \( c_1 = \langle \text{procs}_1, s_1, r_1, \text{ch}_1 \rangle \sqsubseteq_{\text{buf}} c_2 = \langle \text{procs}_1, s_1, r_1, \text{ch}_1 \rangle \) then there exists an injective mapping \( t : \text{procs}_1 \rightarrow \text{procs}_2 \) such that states, register and channel content are preserved in the following way:

1. **States are preserved.** \( s_1(p) = s_2(t(p)) \) for every process \( p \in \text{procs}_1 \).
2. **Register content is either preserved (modulo \( t \)), empty or pointing to an external process.** For every process \( p \in \text{procs}_1 \) and every register \( x \in X \) we have either:
   a) \( r_1(p)(x) \in \text{procs}_1 \) and \( t(r_1(p)(x)) = r_2(t(p))(x) \),
   b) \( r_1(p)(x) = \bot \) and \( r_2(t(p))(x) = \bot \), or
   c) \( r_1(p)(x) = \bot \) and \( r_2(t(p))(x) = p' \in \text{procs}_2 \setminus t(\text{procs}_1) \).
3. **Channel content has the same length, same content modulo \( t \).** For every process \( p \in \text{procs}_1 \), there is \( n_p : 0 \leq n_p \leq l \), \( m_1, \ldots, m_{n_p} \in M \), \( id_1^1, \ldots, id_{n_p}^1 \in \text{procs}_1 \cup \{ \bot \} \) and \( id_1^2, \ldots, id_{n_p}^2 \in \text{procs}_2 \cup \{ \bot \} \) such that:
   a) \( \text{ch}_1(p) = m_{n_p} \langle id_{n_p}^1 \rangle \ldots m_1 \langle id_{n_p}^1 \rangle \).
b) \( ch_2(t(p)) = m_{n_p} \langle id_{n_p}^2 \rangle \ldots m_1 \langle id_1^2 \rangle \), and

c) if \( id_1^i \in \text{procs}_1 \) then \( id_2^i = t(id_1^i) \), otherwise if \( id_1^i = \perp \) then either \( id_2^i = \perp \) or \( id_2^i \in \text{procs}_2 \setminus t(\text{procs}_1) \).

If all processes in \( \text{procs}_2 \setminus t(\text{procs}_1) \) disconnect themselves from configuration \( c_2 \), then we obtain a configuration \( c' \) composed of the set of isolated processes \( \text{procs}_2 \setminus t(\text{procs}_1) \) together with a configuration \( c'_1 \) identical to \( c_1 \) modulo process renaming.

Since configurations are buffer bounded, vertex labels in the graph encoding belong to the finite set \( Q \cup (M \times \{1, \ldots, l\}) \cup (\{e\} \times \{1, \ldots, l\}) \). Moreover, since the network graph of the configurations in \( C_{WSTS} \) are strongly bounded, so are their graph representation \( \enc \). This allows us to use the result by Ding in [42] stating that the induced subgraph relation is a WQO over the set of bounded undirected graphs. As a consequence, \( \sqsubseteq_{\text{buf}} \) is also a WQO over the set \( C_{WSTS} \).

**Monotonicity.**

To prove that the transition relation \( \rightarrow_{WSTS} \) is monotonic with respect to the ordering \( \sqsubseteq_{\text{buf}} \), we proceed in a similar manner as for the case of synchronous DRA. The only difference resides in using disconnect transitions instead of register reset transitions.

**Proof.** Let \( c_1, c_2, c_3 \) be three configurations from \( C_{WSTS} \) such that \( c_1 \rightarrow_{WSTS} c_2 \) and \( c_1 \sqsubseteq_{\text{buf}} c_3 \). To show the monotonicity of \( \rightarrow_{WSTS} \) with respect to the ordering \( \sqsubseteq_{\text{buf}} \), we need to prove the existence of a configuration \( c_4 \in C_{WSTS} \) such that \( c_3 \rightarrow_{WSTS} c_4 \) and \( c_2 \sqsubseteq_{\text{buf}} c_4 \).

Since \( c_1 \rightarrow_{WSTS} c_2 \) and \( \rightarrow_{WSTS} = (\rightarrow_{\leq l, \leq k}^D) \circ (\text{disconnect})^* \), there should be a \( j \in \mathbb{N} \), a transition \( t \in \rightarrow_{\leq l, \leq k}^D \) and a configuration \( c_1^j \) such that \( c_1^j \rightarrow_{\text{disconnect}} c_1^j \) and \( c_1^j \rightarrow_{\text{disconnect}} c_2 \).

Since \( c_1 \sqsubseteq_{\text{buf}} c_3 \), there should be an injective mapping \( t \) from the set of processes of \( c_1 \) to the set of processes of \( c_3 \) satisfying conditions (1), (2) and (3) listed in the previous paragraph. Moreover, there should be a set of processes \( \text{diff} \) in configuration \( c_3 \) composed of processes that are not the image by \( t \) of any process in \( c_1 \). As mentioned in the previous paragraph, there should also be a configuration \( c_3^0 \) such that \( c_3^0 \rightarrow_{\text{disconnect}} \text{diff} \) and such that \( c_3^0 \) is composed of the set of disconnected processes \( \text{diff} \) together with a configuration \( c_3^0 \) identical to \( c_1 \) modulo process renaming.

Since \( c_3 \) is identical to \( c_1 \) modulo process renaming, it can perform the same transitions \( c_1 \) did in order to obtain, successively, \( c_3^0 \) and \( c_2 \). In other words, there should be two configurations \( c'_\text{sub} \) and \( c''\text{sub} \) respectively identical to \( c_3^0 \) and to \( c_2 \) (modulo process renaming), such that \( c'_\text{sub} \rightarrow_{\text{disconnect}} c'_\text{sub} \) and \( c'_\text{sub} \rightarrow_{\text{disconnect}} c''\text{sub} \).
Since $c_3$ is composed of the configuration $c_{\text{sub}}$ together with the set of disconnected processes $\text{diff}$, there should be two configurations $c_4^0$ and $c_4^0$ such that:

- $c_3 \xrightarrow{\text{disconnect}} c_4^0$ and $c_4^0 \xrightarrow{t} c_4$,
- $c_4^0$ is composed of $c_{\text{sub}}'$ together with the set of disconnected processes $\text{diff}$, and
- $c_4$ is composed of $c_{\text{sub}}''$ together with the set of disconnected processes $\text{diff}$.

Since $c_3 \xrightarrow{\text{disconnect}} c_4^0$ and $c_4^0 \xrightarrow{t} c_4$, we have that $c_3 \xrightarrow{\text{disconnect}} c_4^0$ and $c_4^0 \xrightarrow{t} c_4$. Hence, $c_3 \xrightarrow{\text{WSTS}} c_4$. Moreover, since $c_4$ is composed of $c_{\text{sub}}''$ together with the set of disconnected processes $\text{diff}$, and since $c_{\text{sub}}''$ is identical to $c_2$ (modulo process renaming), we deduce that $c_2 \equiv_{\text{buf}} c_4$. □

Remarks concerning the choice of the ordering $\equiv_{\text{buf}}$ and the graph representation $\text{enc}_{\leq l}$:

- Let $c_1 = \langle \text{procs}_1, s_1, r_1, \text{ch}_1 \rangle, c_2 = \langle \text{procs}_2, s_2, r_2, \text{ch}_2 \rangle \in C_{\text{WSTS}}$ be two configurations such that $c_1 \equiv_{\text{buf}} c_2$, i.e. $\text{enc}_{\leq l}(c_1) \equiv_{\text{ind}} \text{enc}_{\leq l}(c_2)$. There should be an injective mapping $t$ between processes in $c_1$ and $c_2$. Moreover, every process $p$ in $c_1$ and its image $t(p)$ in $c_2$ have the same buffer length and the same buffer content modulo the mapping $t$. This is due to the definition of the graph representation $\text{enc}_{\leq l}$ in which empty buffer slots are also encoded. Encoding empty buffer slots is deliberate. The reason behind this choice is to avoid the situation in which the image $t(p)$ of a process $p$ has a full buffer while $p$ can receive messages. This situation will break the monotonicity of the sending transition with respect to the chosen ordering.
- Both embedded subgraph and induced subgraph relations are WQO over strongly bounded graphs. We choose the induced subgraph relation to define the ordering $\equiv_{\text{buf}}$ because if fits with the model of lossy BDRA in which disconnect transitions are allowed. The choice of the embedded subgraph relation would be possible if lossy BDRA allowed register reset transitions, which is not the case.

The reduction.

Given a target state $q_{\text{target}} \in Q$, we use $C_{q_{\text{target}}} \subseteq C_{\text{WSTS}}$ to denote the set that contains all configurations composed of a single process $p$ in state $q_{\text{target}}$, whose registers are empty and whose buffer contains any word $w \in (M \times \{\bot\})^{\leq l}$, i.e. any word of length at most equal to $l$ and which letters are from $(M \times \{\bot\})$. We show that we can reduce the reachability of $q_{\text{target}}$ in the transition system induced by the lossy BDRA $D$ and restricted to $l$-bounded buffer, $k$-strongly-bounded network configurations, to the
coverability problem of the finite set $C_{\text{target}}$ in the transition system $\mathcal{T}_{\text{WSTS}}$. Moreover, we have shown that the transition relation $\rightarrow_{\text{WSTS}}$ of the transition system $\mathcal{T}_{\text{WSTS}}$ is monotonic with respect to the well-quasi ordering $\sqsubseteq_{\text{buf}}$. This shows that the transition system $\mathcal{T}_{\text{WSTS}}$ is well-structured. Finally, we show in Paper III that $\mathcal{T}_{\text{WSTS}}$ satisfies the sufficient conditions listed in [6] to show the decidability of the coverability problem for a finite set of configurations.

5.5.3 Acyclic strongly bounded topology BDRA with bounded buffers.

We consider the subclass of bounded-buffer strongly-bounded acyclic BDRA network configurations. A BDRA network configuration $c$ is acyclic if there are no cycles in the underlying undirected graph of the encoding $\text{rep}_2(c)$. We show that the state reachability problem for this subclass is decidable. In fact, we show that this problem reduces to the reachability problem of a finite-state system.

This reduction follows from the following observations. First, the lack of cycles in the configurations forbids the sending of process IDs. As a consequence, the only operations allowed in this subclass are local actions, process creations, plain message (i.e., with no ID) sending and receiving and process disconnections. Therefore, we can prove by induction that the graph encoding of any configuration reachable in this subclass has the shape of a forest, i.e., a set of trees. Second, since we only consider strongly bounded configurations, the depth of the trees in these configurations are bounded. Third, since the number of registers per process is fixed, every process in the forest has at most $|X|$ children, where $|X|$ is the number of registers per process. Thus, any configuration reachable in this subclass is a forest which depth is bounded and where nodes have a bounded number of children. This allows us to define a transition system composed of the finite set of bounded-depth bounded-buffer acyclic BDRA tree configurations. The transition relation of this system inherits the same transition relation as for the BDRA, except when a tree splits into a finite number of subtrees, in which case the successor configuration is chosen non deterministically among the subtrees. Since there are finitely many of such trees, the state reachability problem for this system is decidable. Finally, the state reachability problem for bounded-buffer strongly-bounded acyclic BDRA is equivalent to the state reachability problem of the finite-state transition system we just defined.
6. Multi-Pushdown Systems

We present in this chapter our work on the verification of concurrent programs (Paper IV). In this work, we assume that a concurrent program is composed of a finite number of threads communicating through a shared memory. Each thread is a sequential program that may contain recursive procedure calls. Threads communicate with each other by reading and writing to the finite number of shared variables that compose the shared memory.

We use Multi-Pushdown Systems (MPDS for short) to model concurrent programs equipped with variables ranging over a finite data domain. Informally, a MPDS is a finite-state automaton that operates on many stacks. MPDS are Turing powerful and therefore come with undecidability for most verification problems. Several under-approximations of MPDS have been considered in the literature to get the decidability of some verification problems. For instance, context bounding [82] approach consists in only considering runs of the MPDS that can be decomposed in a bounded number of contexts, where a context is a continuous segment in which only one of the stacks is active. Checking the reachability of a control state of a MPDS under this assumption is NP-complete. Another under-approximation consists in bounding the depth of the stacks. By only considering runs of the MPDS in which the depth of the stacks is bounded by some constant value, it is well-known (and probably folklore) that the state reachability problem becomes PSAPCE-complete.

The approach that we propose combines these two ideas. More precisely, given a pair of natural numbers \((k,d)\), every thread is allowed a budget of context switches that varies depending on the depth of its stack. If the depth of the stack is less or equal than \(d\), then the thread is allowed to perform an unbounded number of context switches. Otherwise, if the depth of the stack is strictly larger than \(d\), then the thread can only perform \(k\) consecutive contexts. When the depth of the stack goes beyond \(d\), the budget of context-switches becomes again unbounded. By combining the context bounded approach and the depth bounded approach, the aim is to explore a larger set of behaviors of the MPDS.

The contributions of Paper IV that we present in this chapter are of two sorts. The first contribution is theoretical and consists in defining the class of bounded budget MPDS and evaluating the complexity of two verification problems: The state reachability problem and the LTL model-checking. We show that the state reachability problem for the class of bounded budget MPDS is PSAPCE-complete and that LTL model-checking of the same class is EXP-TIME-complete. The second contribution is a code-to-code translation scheme.
It takes as input a concurrent program $P$ and a budget $(k,d)$ and produces a sequential program $P'$. The translation is such that the state reachability problem in the concurrent program $P$ under the $(k,d)$ budget bounded restriction is reducible to the state reachability problem in the sequential program $P$. To demonstrate the utility of this code-to-code translation, we apply it to a set of concurrent programs. We feed the resulting sequential program translation to three different backend sequential program verifiers and compare the verification results with the ones provided by concurrent program verifiers. Finally, we show how to extend the code-to-code translation to check for program termination under different fairness conditions.

Outline of the chapter.
In Section 6.1, we start by listing definitions and preliminary results used in this chapter. In Section 6.2, we formally define the bounded budget MPDS and the related state reachability problem. In Section 6.3, we show that the state reachability problem for bounded budget MPDS is PSAPCE-complete. In Section 6.4, we prove that the LTL model checking of budget bounded MPDS is EXPTIME-complete. In Section 6.5, we present the concurrent-to-sequential code-to-code translation inspired by the bounded budget under-approximation.

6.1 Preliminaries
We define in this section the shuffle operator, the notions of finite-state automata and pushdown automata, the synchronization of depth-bounded PDA problem, two preliminary results in the form of two lemmas that we use later in this chapter, and, finally, the notion of multi-pushdown systems.

Shuffle operator.
Let $\Sigma$ be a finite alphabet. Let $L$ be a set of (possibly infinite) words over $\Sigma$ and let $w \in \Sigma^*$ be a word over $\Sigma$. We define $w.L := \{w.w' \mid w' \in L\}$. We define the shuffle operator $\sqcup$ over two (possibly infinite) words over $\Sigma$ inductively as follows: $\sqcup(\epsilon, u) = \sqcup(u, \epsilon) = \{u\}$ and $\sqcup(a.u, b.v) = a.\sqcup(u, b.v) \cup b.\sqcup(a.u, v)$. Given two sets of (possibly infinite) words $L$ and $L'$, we define their shuffle as $\sqcup(L, L') = \bigcup_{u \in L, v \in L'} \sqcup(u, v)$. The shuffle operator for multiple sets can be extended in a similar manner.

Finite-state automaton.
A finite-state automaton is a tuple $\langle Q, \Sigma, \Delta, I, F \rangle$ where: (i) $Q$ is a finite set of states, (ii) $\Sigma$ is a finite input alphabet, (iii) $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a transition relation, (iv) $I$ is a set of initial states, and (v) $F$ is a set of final states. The (regular) language of finite words accepted by $A$ is denoted by $L(A)$. We may also interpret the set $F$ as a Büchi acceptance condition, and we denote by
$L^o(A)$ the set of infinite words accepted by $A$. The size of $A$ is defined by $|Q| + |\Sigma|$.

**Pushdown Automaton.**

A pushdown automaton (PDA for short) $P$ is defined as a tuple $\langle Q, \Sigma, \Gamma, \delta, I, F \rangle$ where (i) $Q$ is a finite set of states, (ii) $\Sigma$ is a finite input alphabet, (iii) $\Gamma$ is a finite stack alphabet, (iv) $\delta$ is a finite set of transition rules of the form $(q, u) \xrightarrow{a} (q', u')$ where $q, q' \in Q$, $a \in \Sigma \cup \varepsilon$, $u, u' \in \Gamma^*$ and $|u| + |u'| \leq 1$, (v) $I \subseteq Q$ is a set of initial states, and (vi) $F \subseteq Q$ is a set of final states. The size of $P$ is defined by $|P| = |Q| + |\Sigma| + |\Gamma|$.

**Configurations and operational semantics.** A configuration of the pushdown automaton $P$ is a pair $\langle q, u \rangle$ of a state $q \in Q$ and a stack content $u \in \Gamma^*$. We define $State(\langle q, u \rangle) := q$ and $Stack(\langle q, u \rangle) := u$. When a configuration $\langle q, u \rangle$ is such that $|Stack(\langle q, u \rangle)| = |u| = d$, we say that the configuration $\langle q, u \rangle$ is of depth $d$. We define the transition relation $\rightarrow$ between configurations of $P$ as follows: $\langle q, u, w \rangle \xrightarrow{a} \langle q', u', w \rangle$ iff $(q, u) \xrightarrow{a} (q', u')$. We use $\rightarrow^*$ to denote the reflexive transitive closure of $\rightarrow$.

**Language.** Let $\sigma \in \Sigma^*$ and $c, c'$ be two configurations of $P$. We write $c \xrightarrow{\sigma}_P c'$ if one of the following cases hold: (i) $\sigma = \varepsilon$ and $c = c'$, or (ii) there is a sequence of input symbols $a_1, a_2, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$ such that $\sigma = a_1 a_2 \ldots a_n$ and there is a sequence of configurations $c_0, c_1, \ldots, c_n$ such that $c_0 = c$, $c_n = c'$, $c_i \xrightarrow{a_i} c_{i+1}$ for every $0 \leq i < n - 1$. The language $L(P)$ accepted by $P$ is defined by the set of finite words $\sigma \in \Sigma^*$ such that $\langle q_{init}, \varepsilon \rangle \xrightarrow{\sigma}_P \langle q_{final}, \varepsilon \rangle$ where $q_{init}$ is an initial state and $q_{final} \in Q_{final}$ is a final state. Similar to the case of finite-state automaton, we can interpret the set $F$ as a Büchi acceptance condition and we denote by $L^o(P)$ the language of infinite words accepted by $P$.

**Bounded depth languages.** Let $d \in \mathbb{N}$. We define the transition relation $\rightarrow_{\leq d}$ (resp. $\rightarrow_{> d}$) between configurations of $P$ as follows: $\langle q, u \rangle \xrightarrow{\leq d} \langle q', v \rangle$ (resp. $\langle q, u \rangle \xrightarrow{> d} \langle q', v \rangle$) iff $|u| \leq d$ and $|v| \leq d$ (resp. $|u| > d$ and $|v| > d$). We use $\rightarrow^*_{\leq d}$ (resp. $\rightarrow^*_{> d}$) to denote the reflexive transitive closure of $\rightarrow_{\leq d}$ (resp. $\rightarrow_{> d}$). Let $a_1, a_2, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$, $\sigma = a_1 a_2 \ldots a_n \in \Sigma^*$ and $c, c'$ two configurations of $P$. We write $c \xrightarrow{\sigma}_{\leq d} c'$ (resp. $c \xrightarrow{\sigma}_{> d} c'$) if one of the following cases holds: (i) $\sigma = \varepsilon$, $c = c'$ and $|Stack(c)| \leq d$ (respectively $|Stack(c)| > d$), or (ii) there exists a sequence of configurations $c_0, c_1, c_2, \ldots, c_n$ of $P$ such that $c = c_0$, $c' = c_n$ and $c_i \xrightarrow{a_i}_{\leq d} c_{i+1}$ (resp. $c_i \xrightarrow{a_i}_{> d} c_{i+1}$) for every $0 \leq i < n$.

Given $k, d \in \mathbb{N}$ and an input word $\sigma \in \Sigma^*$, we define the relation $\xrightarrow{\sigma}_{(k,d)}$ between configurations of depth $d$ as follows: $\langle q, u \rangle \xrightarrow{\sigma}_{(k,d)} \langle q', u' \rangle$ iff (i) $|\sigma| \leq k$, (ii) $u = u'$ and $|u| = d$, (iii) there exist two input letters $a_1, a_2 \in \Sigma \cup \{\varepsilon\}$ and a word $\sigma' \in \Sigma^*$ such that $\sigma = a_1 \sigma' a_2$, and (iv) there exist two configurations $\langle q_1, u_1 \rangle$ and $\langle q_2, u_2 \rangle$ such that $\langle q, u \rangle \xrightarrow{a_1} \langle q_1, u_1 \rangle \xrightarrow{\sigma'} \langle q_2, u_2 \rangle \xrightarrow{a_2} \langle q', u' \rangle$. 103
We use $L_{(k,d)}(P)$ to denote the set of words $\sigma \in \Sigma^*$ such that there is a sequence of configurations $c_0, c_1, \ldots, c_n$ of $P$ and a sequence of words $\sigma_1, \sigma_2, \ldots, \sigma_n$ such that: (i) $\sigma = \sigma_1 \sigma_2 \ldots \sigma_n$, (ii) $c_0 = \langle q_0, \epsilon \rangle$ with $q_0 \in I$, (iii) $c_n = \langle q_n, \epsilon \rangle$ with $q_n \in F$, (iv) for every $i : 0 \leq i < n$ we have either $c_i \xrightarrow{\sigma_i \sigma_{i+1}}_{(k,d)} c_{i+1}$, or $c_i \xrightarrow{\sigma_{i+1}} \leq_d c_{i+1}$. We call $L_{(k,d)}(P)$ the $(k,d)$-bounded language of $P$.

We define the language $L_{(k,d)}^o(P)$ to be the set of infinite words $\sigma \in \Sigma^*$ such that there are an infinite sequence of configurations $c_0, c_1, \ldots$ and an infinite sequence of input words $\sigma_1, \sigma_2, \ldots \in \Sigma^*$ where: (i) $\sigma = \sigma_1 \sigma_2 \ldots$, (ii) $c_0 = \langle q_0, \epsilon \rangle$ with $q_0 \in I$, (iii) there are infinitely many indices $i_1, i_2, \ldots$ such that $\text{State}(c_{i_j}) \in F$ for every $j \geq 1$, and (iv) for every $i : 0 \leq i < n$ we have either $c_i \xrightarrow{\sigma_{i+1}}_{(k,d)} c_{i+1}$, or $c_i \xrightarrow{\sigma_{i+1}} \leq_d c_{i+1}$.

We also define the language $L_d(P)$ to be the set of words $\sigma \in \Sigma^*$ such that $\langle q_0, \epsilon \rangle \xrightarrow{\sigma} \langle q_n, \epsilon \rangle$ for some $q_0 \in I$ and $q_n \in F$. This language contains all the words accepted through runs of $P$ in which the stack depth is bounded by $d$. Similar to the case of finite-state automaton, we denote by $L_{(d)}^o(P)$ the language of infinite words accepted by $P$ by interpreting the set $F$ as a Büchi acceptance condition.

**Synchronization of depth-bounded PDA.**

Given pushdown automata $P_0, P_1, \ldots, P_n$ and depth-bounds $d_0, d_1, \ldots, d_n \in \mathbb{N}$, we define the problem which consists in checking the emptiness of the language $L_{d_0}(P_0) \cap \sqcup \{L_{d_1}(P_1), L_{d_2}(P_2), \ldots, L_{d_n}(P_n)\}$ as the non-emptiness test of the synchronization of depth-bounded PDA.

**Preliminary results.**

The following lemmas are needed to prove the theoretical results presented in the rest of this chapter. The proofs of both lemmas can be found in the Appendix of this chapter (Section 6.A).

**Lemma 6.1.1.** Let $k, d \in \mathbb{N}$ and $P$ be a pushdown automata. It is possible to build in polynomial time a pushdown automaton $P'$ such that $L_{k+d+2}(P') = L_{(k,d)}(P)$ and $L_{(k,d)}^o(P') = L_{(k,d)}^o(P)$.

**Lemma 6.1.2.** The non-emptiness test of the synchronization of depth-bounded pushdown automata is PSPACE-complete.

**Multi-Pushdown Systems.**

A Multi Pushdown System is a tuple $M = \langle n, Q, \Gamma, \Delta, q_{\text{init}} \rangle$ where $n \geq 1$ is is the number of stacks, $Q$ is a finite set of states, $\Gamma$ is the stack alphabet, $\Delta \subseteq (Q \times \{1, \ldots, n\} \times Q) \cup (Q \times \{1, \ldots, n\} \times Q \times \Gamma) \cup (Q \times \Gamma \times \{1, \ldots, n\} \times Q)$ is the transition relation and $q_{\text{init}}$ is the initial state.
MPDS are finite-state machines that operate over a finite set of stacks. Every transition of a MPDS is either (i) a push operation of a symbol γ on stack i of the form \((q_1,i,q_2,γ)\), (ii) a pop operation of a symbol γ from stack i of the form \((q_1,γ,i,q_2)\), or (iii) an internal operation of the form \((q_1,i,q_2)\). The last type of transitions does not change the content of any of the stacks. The stack index \(i\) that appears in its syntax denotes that this transition models a local action of the thread that operates on stack \(i\).

Note that since we are not interested in using the MPDS model as a language acceptor, we do not include in its definition an input alphabet nor a final state.

A configuration \(c\) of an MPDS \(M = \langle n,Q,Γ,Δ,q_{init}\rangle\) is given by a tuple \(\langle q,w_1,\ldots,w_n\rangle\) where \(q \in Q\) is the current state and \(w_i \in Γ^*\) is the content of stack \(i\). We use \(State(c)\) and \(Stack_i(c)\) to denote respectively the state \(q\) and the content \(w_i\) of stack \(i\), for \(i : 1 \leq i \leq n\). The initial configuration of \(M\) is defined as \(c_{init}^M := \langle q_{init},ε,\ldots,ε\rangle\).

We define the transition relation \(\rightarrow_M\) on the set of configurations of the MPDS \(M\) as follows. For two configurations \(c = \langle q,w_1,\ldots,w_n\rangle\) and \(c' = \langle q',w'_1,\ldots,w'_n\rangle\) of \(M\), a stack index \(i : 1 \leq i \leq n\) and a transition \(t \in Δ\), we write \(c \xrightarrow{t}_M c'\) to denote that one of the following cases hold:
- Internal operation: \(t = (q,i,q')\) and \(w_j = w_j'\) for all \(j : 1 \leq j \leq n\).
- Push operation: \(t = (q,i,q',γ)\), \(γ.w_i = w_i'\) and \(w_j = w_j'\) for all \(j : 1 \leq j \leq n\) such that \(j \neq i\).
- Pop operation: \(t = (q,γ,i,q',q)\), \(γ.w_i' = w_i\) and \(w_j = w_j'\) for all \(j : 1 \leq j \leq n\) such that \(j \neq i\).

A computation \(π\) of \(M\) from a configuration \(c\) to a configuration \(c'\) is a sequence \(c_0t_1c_1t_2c_2\ldots c_m\) with: (i) \(c_0 = c\), (ii) \(c_m = c'\), and (iii) \(c_{i−1} \xrightarrow{t_i}_M c_i\) for all \(i : 1 \leq i \leq m\). Each configuration \(c_i\), \(i : 0 \leq i \leq m\) is said to be reachable from \(c\). We use initial(\(π\)) and target(\(π\)) to denote \(c_0\) and \(c_m\), respectively. Given two computations \(π_1 = c_0t_1c_1\ldots c_m\) and \(π_2 = c_{m+1}t_{m+2}c_{m+2}\ldots c_k\) of \(M\), we say that \(π_1\) and \(π_2\) are compatible if \(c_m = c_{m+1}\). In this case, we write \(π_1 ∗ π_2\) to denote the computation \(c_0t_1c_1\ldots c_m t_{m+2} c_{m+2}\ldots c_k\).

### 6.2 Bounded Budget MPDS

In this section, we introduce the class of bounded budget MPDS and define the bounded budget state reachability problem.

Let \(M = \langle n,Q,Γ,Δ,q_{init}\rangle\) be a MPDS and \((k,d)\) be a pair of natural numbers. In the \((k,d)\)-bounded-budget under-approximation of the MPDS \(M\), we only consider computations of \(M\) that are \((k,d)\)-budget bounded. In order to define a \((k,d)\) budget bounded computation, let us first define the notions of context, sequence of compatible contexts with respect to some stack, depth of a context and block of contexts of depth \(d\).
Context. A context of a stack \( i : 1 \leq i \leq n \) is a computation of \( M \) of the form \( c_0t_1c_1 \ldots c_m \) in which every transition \( t_j \) \((j : 1 \leq j \leq m)\) is in the subset:

\[
\Delta_i := \Delta \cap \left((Q \times \{i\} \times Q) \cup (Q \times \{i\} \times \Gamma) \cup (Q \times \Gamma \times \{i\} \times Q)\right)
\]

Informally, a context models a consecutive sequence of actions performed by one single thread. Observe that every computation of \( M \) can be seen as concatenation of a sequence of contexts \( \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_l \) where every \( \pi_j \) \((j : 1 \leq j \leq l)\) is a context of some stack \( i : 1 \leq i \leq n \).

Sequence of compatible contexts. For any two contexts \( \pi_1 \) and \( \pi_2 \) of the same stack \( i : 1 \leq i \leq n \), we say that \( \pi_1 \) and \( \pi_2 \) are compatible with respect to stack \( i \) if \( \text{Stack}_i(\text{target}(\pi_1)) = \text{Stack}_i(\text{initial}(\pi_2)) \), and we write \( \pi_1 \bullet \pi_2 \). This notion of compatibility with respect to a stack of two contexts is extended in a straightforward manner to sequences of contexts.

Let \( \pi = \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_m \) be a computation of \( M \) such that each \( \pi_j \), \( j : 1 \leq j \leq m \), is a context of some stack \( i : 1 \leq i \leq n \). If \( 1 \leq j_1 < j_2 < \ldots < j_k \leq m \) are all the indices \( j \) such that \( \pi_j \) is a context of some stack \( i \), then \( \pi_{j_1} \bullet \pi_{j_2} \bullet \ldots \bullet \pi_{j_k} \).

We call this sequence the projection of \( \pi \) on stack \( i \), denoted \( \downarrow(\pi, i) \).

Context depth. A context \( \pi = c_0t_1c_1t_2 \ldots t_mc_m \) of a stack \( i : 1 \leq i \leq n \) is said to be of depth at most (resp. least) \( l \in \mathbb{N} \) if and only if for every \( j : 0 \leq j \leq m \), \( |\text{Stack}_i(\pi_j)| \leq l \) (resp. \( |\text{Stack}_i(\pi_j)| \geq l \)). The definition is extended in the straightforward manner to sequences of contexts.

Block of contexts of depth \( d \). A block of contexts (or simply block) \( \rho \) of size \( m \in \mathbb{N} \) and depth \( d \in \mathbb{N} \), relative to a stack \( i : 1 \leq i \leq n \), is a sequence of compatible contexts of the form \( \pi_a \bullet_i \pi_1 \bullet_i \pi_2 \bullet_i \ldots \bullet_i \pi_m \bullet_i \pi_b \) of stack \( i \) such that:

- \( \pi_j \) is a context of depth at least \( d + 1 \) for all \( j : 1 \leq j \leq m \).
- \( \pi_a = c_0t_ac'_a \) is a context where (i) \( c_a \) and \( c'_a \) are two configurations such that \( |\text{Stack}_i(\pi_a)| = |\text{Stack}_i(\pi_a)| + 1 = d + 1 \), and (ii) \( t_a \) is a push transition. \( \pi_a \) is called a pusher context of depth \( d \).
- \( \pi_b = c(bt_bc'_{b}) \) is a context where (i) \( c_b \) and \( c'_b \) are two configurations such that \( |\text{Stack}_i(\pi_b)| = |\text{Stack}_i(\pi_b)| + 1 = d + 1 \), and (ii) \( t_b \) is a pop transition. \( \pi_b \) is called a popper context of depth \( d + 1 \).

Bounded budget computations.

We can now define a bounded budget computation. Intuitively, a \( (k,d) \)-bounded budget computation is a computation in which any symbol that is pushed in a stack of depth at least \( d \), has to be popped from the stack within \( k \) context switches. This implies that, in a bounded budget computation, each computation of the stack \( i \) is a concatenation of contexts of depth at most \( d \) and blocks of depth \( d \) and of size smaller than \( k \). The formal definition is as follows:

Let \( \pi = \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_m \) be a computation of \( M \) and \( (k,d) \in \mathbb{N}^2 \). We say that \( \pi \) is \( (k,d) \)-bounded budget if for every stack \( i : 1 \leq i \leq n \), if \( \downarrow(\pi, i) = \pi_i^1 \bullet_i \)}
\(\pi_i^2 \cdot \pi_i \cdots \pi_i^{m_i}\) is the maximal sub-sequence of contexts in \(\pi\) belonging to the stack \(i\), then \(\perp(\pi, i)\) can written as a compatible sequence \(\rho_i^{1} \cdot \rho_i^{2} \cdot \pi_i \cdots \rho_i^{\ell_i}\) of length \(\ell_i \geq 0\), composed of:
- Contexts of depth at most \(d\), and
- Blocks of contexts of size (at most) \(k\) and of depth \(d\).

**Bounded budget state reachability problem.**

Let \(M = \langle n, Q, \Gamma, \Delta, q_{init} \rangle\) be a MPDS, and \((k,d) \in \mathbb{N}^2\). Given a state \(q_{final} \in Q\), the \((k,d)\) bounded budget reachability problem asks whether there exists a \((k,d)\) bounded budget computation from the initial configuration \(c_{M}^{\text{init}}\) to the configuration \((q_{final}, \varepsilon, \ldots, \varepsilon)\). The input size of this problem is defined as \(n + |Q| + |\Gamma| + |\Delta| + k + d\).

### 6.3 Bounded Budget State Reachability is PSAPCE-Complete

This section is devoted to show the following result:

**Theorem 6.3.1.** The \((k,d)\)-bounded budget state reachability problem for MPDS is PSAPCE complete.

**The lower bound.**

We show the lower bound through a reduction from the non-emptiness problem of the intersection of \(n\) regular languages, known to be PSAPCE-hard [62]. An instance of the non-emptiness problem of the intersection of \(n\) regular languages is given by \(n\) finite-state automata \(A_1, A_2, \ldots, A_n\) with \(A_i := \langle Q_i, \Sigma, \delta_i, I_i, F_i \rangle\) for every \(1 \leq i \leq n\). The non-emptiness problem of the intersection between the regular languages \(L(A_1), L(A_2), \ldots, L(A_n)\) asks whether there exists a word \(w \in \Sigma^*\) such that \(w \in L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)\). We show that this problem reduces to the \((0,1)\)-bounded budget state reachability problem for an MPDS \(M\) that we define.

First, given an automaton \(A\) and a letter \(a\) from its alphabet, we define the automaton \([A]_a\) as the copy of automaton \(A\) in which only transitions reading the letters \(\varepsilon\) or \(a\) are kept. Second, using the set of automata \([A_i]_a\) \((1 \leq i \leq n, a \in \Sigma)\), we build a MPDS \(M\) that iteratively (a) guesses a letter \(a\) from \(\Sigma\), and (b) runs each automaton \(A_i\) with the input letter \(a\). Before each one of these iterations, the MPDS \(M\) can nondeterministically check whether the word composed of the letters guessed so far is accepted by each automaton \(A_i\), \(1 \leq i \leq n\). The MPDS \(M\) uses its \(i^{th}\) stack in order to keep track of the current state of each automaton \(A_i\), \(1 \leq i \leq n\). Finally, we show that the reduction holds.

**Automaton \([A]_a\).** Let \(A = \langle Q, \Sigma, \delta, I, F \rangle\) be a finite-state automaton and \(a \in \Sigma\). Let \(A_{\varepsilon} = \langle Q, \Sigma, \delta_{\varepsilon}, I_{\varepsilon}, F_{\varepsilon} \rangle\) be the result of removing all \(\varepsilon\) transitions in \(A\) by
saturating the set of transitions $\delta$. The finite-state automaton $[A]_a$ is a copy of $A_\emptyset$ such that: (i) For each state $q$ from the set of states $Q$ of $A$, $[A]_a$ contains the pair of states $[q]_a$ and $[q]^a_a$, (ii) Only transitions reading $a$ are kept, (iii) If $[A]_a$ is in a state of the form $[q]_a$, then the word read by $[A]_a$ so far is the empty word $\varepsilon$, (iv) If $[A]_a$ is in a state of the form $[q]^a_a$, then the word read by $[A]_a$ so far is the letter $a$. Intuitively, automaton $[A]_a$ simulates runs of automaton $A$ composed of $\varepsilon$ transitions and of at most one transition labeled with the letter $a$. Formally, $[A]_a$ is defined by the tuple $(Q_a, \Sigma, \delta_a, I_a, F_a)$ where:

1. The set of states is $Q_a = \{[q]_a, [q]^a_a \mid q \in Q\}$,
2. The transition relation is given by $\delta_a = \{([q]_a, a, [q]^a_a) \mid (q, a, q') \in \delta\}$,
3. The set of initial states $I_a = \{[q]_a \mid q \in I\}$,
4. The set of final states $F_a = \{[q]_a, [q]^a_a \mid q \in F\}$.

The size of $[A]_a$ is polynomial in the size of $A$.

**Definition of the MPDS $M$.** We define the MPDS $M = (n, Q, \Gamma, \Delta, q_0)$ such that it includes all the states and transitions of automata $[A_i]_a$ for $i: 1 \leq i \leq n$ and $a \in \Sigma$. $M$ invariantly keeps track of the current state of each automaton $A_i$ by saving it in stack $i$, for $i: 1 \leq i \leq n$. $M$ starts by performing an initialization phase from the initial state $q_0$. In this phase, $M$ goes through the states $q_1, \ldots, q_n$ and saves at each state $q_i$ an initial state of the automaton $A_i$ in stack $i$. After the completion of this phase, $M$ moves to state $q_{\text{sim}}$ in which the simulation phase starts. The simulation consists in iteratively: (i) Guessing a letter $a$ from the alphabet $\Sigma$, and (ii) checking that every automaton $A_i$ reads letter $a$. Moreover, $M$ can nondeterministically check in state $q_{\text{sim}}$ if all automata $A_i$ are in an accepting state. In this case, it moves to the halting states $q_{\text{end}}$. Formally, the MPDS $M$ is defined as follows:

1. The set of states $Q$ of $M$ is the union of the following subsets:
   - $\{q_0, q_1, q_2, \ldots, q_n, q_{\text{sim}}, q_{\text{end}}\}$,
   - $\{[q]_a, [q]^a_a \mid q \in \bigcup_{1 \leq i \leq n} Q_i, \ a \in \Sigma\}$, and
   - $\{q^a_i \mid a \in \Sigma \cup \{\text{acc}\}, i: 1 \leq i \leq n + 1\}$
2. The stack alphabet $\Gamma$ is the set $\bigcup_{1 \leq i \leq n} \{q \mid q \in Q_i\}$.
3. The transition relation is the union of the following subsets:
   a. The transition relations of every automata $[A_i]_a$ for $i: 1 \leq i \leq n$ and $a \in \Sigma$.
   b. The transitions of the initialization phase:
      $\{(q_{i-1}, i, q_i, \langle q_{\text{init}} \rangle) \mid i: 1 \leq i \leq n, q_{\text{init}} \in I_i\} \cup \{(q_n, 1, q_{\text{sim}})\}$
   c. In state $q_{\text{sim}}$, $M$ can either guess which is the next letter $a \in \Sigma$ to be commonly read by each automaton $A_i$, $i: 1 \leq i \leq n$, or start checking the acceptance condition of every $A_i$, $i: 1 \leq i \leq n$. The guessing corresponds to the set of transitions $\{(q_{\text{sim}}, 1, q^a_i) \mid a \in \Sigma\}$ while starting checking the acceptance condition corresponds to the transition $(q_{\text{sim}}, 1, q^1_{\text{acc}})$.
   d. In every state $q^a_i$, $i: 1 \leq i \leq n, a \in \Sigma$, $M$ checks that $A_i$ can read letter $a$ by:
At this point, since automaton accepting states should be the reachability of the state is in an accepting state after having read the succession of letters of this section is dedicated to this reduction.

We show that we can reduce the bounded budget state reachability problem for MPDS to the non-emptiness test of the synchronization of depth-bounded PDA, a PSAPCE-complete problem according to lemma 6.1.2. The remaining of this section is dedicated to this reduction.

Let $M = (\mathcal{A}, Q, \Gamma, \Delta, q_{\text{init}})$ be an MPDS, $q_{\text{final}}$ be one of its states and $(k, d) \in \mathbb{N}^2$ be a budget. We define in the following the set of pushdown automata

\begin{enumerate}
  
  \item Reading the current state $q_{\text{cur}}$ of $A_i$ from stack $i$ and moving to state $[q_{\text{cur}}]_a$: \( \{ (q_a^0, \langle q_{\text{cur}} \rangle, i, [q_{\text{cur}}]_a) \mid q_{\text{cur}} \in Q_i \} \).
  
  \item In state $[q_{\text{cur}}]_a$, the automaton $[A_i]_a$ starts running.
  
  \item If $[A_i]_a$ ends up in a state of the form $[q_{\text{new}}]_a^n$, then it means that automaton $A_i$ can move to a new state $q_{\text{new}}$ after reading the letter $a$. In this case, $M$ saves the new state in stack $i$ and moves to the next step where the same check happens for $A_{i+1}$: \( \{ ([q_{\text{new}}]_a^n, i, q_{\text{new}}^{i+1}, \langle q_{\text{new}} \rangle) \mid q_{\text{new}} \in Q_i \} \).
  
  \item In every state $q_{a}^{i+1}$, $a \in \Sigma$, $M$ has checked that letter $a$ was read by every automaton. $M$ can then start the next guessing and checking iteration by taking the transition $(q_{a}^{i+1}, 1, q_{\text{sim}})$.
  
  \item In every state $q_{\text{acc}}^{i}$, $1 \leq i \leq n$, $M$ checks that $A_i$ is in an accepting state by performing any of the transitions in: \( \{ (q_{\text{acc}}^i, \langle q \rangle, i, q_{\text{acc}}^{i+1}) \mid q \in F_i \} \).
  
  \item In state $q_{\text{acc}}^{n+1}$, $M$ has checked that every automaton $A_i$ is in an accepting state and can therefore stop the simulation: \( \{ (q_{\text{acc}}^{n+1}, 1, q_{\text{end}}) \} \).

\textbf{Correctness of the reduction.} Assume that the state $q_{\text{end}}$ is reachable by the MPDS $M$. If this is the case, $M$ should have guessed a number $m \geq 0$ of letters $a_1, a_2, \ldots, a_m$ from $\Sigma$, checked that every letter $a_j$, $1 \leq j \leq m$, was read by every automaton $A_i$, $i : 1 \leq i \leq n$, and finally checked that every automaton $A_i$ is in an accepting state after having read the succession of letters $a_1, a_2, \ldots, a_m$. This translates into the fact that the word $w = a_1 a_2 \ldots a_m$ is accepted by every automaton $A_i$, $1 \leq i \leq n$, i.e., $L(A_i) \cap L(A_2) \cap \ldots \cap L(A_n) \neq \emptyset$.

Reversely, if there is a word $w \in L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)$, then there should be $m \geq 0$ letters $a_1, a_2, \ldots, a_m \in \Sigma$ such that $w = a_1 a_2 \ldots a_m \in \Sigma^*$. By successively choosing the letters $a_1, a_2, \ldots, a_m$ at each iteration, $M$ can check that each letter $a_j$, $1 \leq j \leq m$, can be read by each automaton $A_i$, $1 \leq i \leq n$. At this point, since $w \in L(A_i)$, each simulated automaton $A_i$ can be in one of its accepting states. $M$ can therefore check and validate the acceptance condition and finally move to the state $q_{\text{end}}$.

Observe that each stack $i$ of $M$ contains at most one stack symbol corresponding to the current state of automaton $A_i$. Therefore, every computation of $M$ is $(k, 1)$ budget bounded for any $k \in \mathbb{N}$. We deduce that the emptiness of the intersection between the regular languages $L(A_1), L(A_2), \ldots, L(A_n)$ reduces to the reachability of the state $q_{\text{end}}$ of the MPDS $M$ under the $(0, 1)$-bounded budget restriction.

\textbf{The upper bound.} We show that we can reduce the bounded budget state reachability problem for MPDS to the non-emptiness test of the synchronization of depth-bounded PDA, a PSAPCE-complete problem according to lemma 6.1.2. The remaining of this section is dedicated to this reduction.

Let $M = (\mathcal{A}, Q, \Gamma, \Delta, q_{\text{init}})$ be an MPDS, $q_{\text{final}}$ be one of its states and $(k, d) \in \mathbb{N}^2$ be a budget. We define in the following the set of pushdown automata
$P'_0, P'_1, \ldots, P'_n$ and the set of depth bounds $l_0, l_1, \ldots, l_n$ such that the state is $q_{\text{final}}$ in the $(k,d)$ bounded budget MPDS $M$ if and only if:

$$L_{(l_0)}(P'_0) \cap \bigcup (L_{(l_1)}(P'_1), \ldots, L_{(l_n)}(P'_n))$$

**Outline of the reduction.** First, we define the notion of a context summary. Second, we build for each stack $i : 1 \leq i \leq n$ a pushdown automaton $P_i$ that iteratively simulates contexts of the MPDS $M$ relative to stack $i$. More precisely, $P_i$ simulates any possible context of stack $i$ that is either 1. a context of depth at most $d$, 2. a context of depth at least $d+1$, 3. a pusher context of depth $d$, or 4. a popper context of depth $d+1$. We define the language of $P_i$ as the set of summaries of every simulated context. Third, we derive from $P_i$ a pushdown automaton $P'_i$ that only allows $k$ consecutive simulations of contexts of depth at least $d+1$. Fourth, we define a pushdown automaton $P_0$ which language contains the summary of every possible run MPDS $M$ that reach $q_{\text{final}}$. Finally, we show that the reduction holds.

**Summary of a context.** Let $\pi$ be a computation of $M$. The computation $\pi$ can be written as a compatible sequence of contexts $\pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m$ such that every context $\pi_j$ is a context of some stack $i : 1 \leq i \leq n$. Since the communication between stacks happens through control states, we can summarize every context $\pi_j$ by a pair of states composed of the state at the beginning and the end of the context. Formally, if $\pi_j = c_0 t_0 c_1 \ldots t_{l-1} c_l$, then we define its summary to be $\text{summary}(\pi_j) := (\text{State}(c_0), \text{State}(c_l))$. The summary of the computation $\pi = \pi_1 \cdot \pi_2 \cdot \ldots \cdot \pi_m$ is defined as the word $\text{summary}(\pi) = \text{summary}(\pi_1) \cdot \text{summary}(\pi_2) \cdot \ldots \cdot \text{summary}(\pi_m)$. We extend this definition in the natural way to compatible sequences of contexts and blocks.

**Pushdown $P_i$: Context simulator.** Let $i : 1 \leq i \leq n$ be the index of a stack. We construct a pushdown automaton $P_i$ that simulates every possible context of the MPDS $M$ relative to stack $i$ of the following sorts: (a) Context of depth at most $d$, (b) context of depth at least $d+1$, (c) pusher context of depth $d$, and (d) popper context of depth $d+1$. In other words, a context simulated by $P_i$ should be of depth at most $d$ or be part of a block of contexts of depth $d$ of some size $m \geq 0$. $P_i$ invariably keeps track in its control state of the depth of the stack $i$ when it is smaller than $d$. $P_i$ also uses a special symbol in the stack to delimit the part of the stack which depth is smaller than $d$ from the part of the stack of depth larger than $d+1$. The exact definition of $P_i$ can be found in Paper IV. We give in the following the main steps used in $P_i$ to simulate each type of context.

$P_i$ simulates a context $\pi$ of depth at $d$ or of depth at least $d+1$ by:

1. Guessing a summary $(q_{\Delta}, q_{\Delta})$ of $\pi$. If the stack depth is less or equal to $d$, then the context being simulated is of depth at most $d$. Otherwise, the simulated context is of depth at least $d+1$.
2. Running $M$ from state $q_{\Delta}$. Runs of $M$ are restricted to transitions $t \in \Delta_i$ relative to stack $i$. 

110
(3) Updating the information about the depth of the stack at each pop (decrementation) and push operation (incrementation),

(4) Allow the simulation of the context to end successfully when \( M \) reaches state \( q_b \). At this point, \( P_i \) reads the letter \((q_a, q_b)\) from the alphabet of summaries \((Q \times Q)\).

(5) Abort the simulation of a context of depth at most \( d \) (respectively of depth at least \( d + 1 \)) when the depth of stack \( i \) is about to become \( d + 1 \) after a push operation (respectively \( d \) after a pop operation).

When the depth of the stack is \( d \), \( P_i \) can simulate a pusher context of depth \( d \) by:

1. Guessing a summary \((q_a, q_b)\) of the pusher.
2. Running \( M \) from state \( q_a \). The run of \( M \), in this case, should not contain any pop operation and should contain one push operation on stack \( i \).
3. If the push transition occurs, push a special symbol in the stack to separate the lower part (depth smaller than \( d \)) from the upper part (depth larger than \( d + 1 \)) of the stack.
4. Allow the simulation of the pusher context to successfully end if \( M \) reaches state \( q_b \). At this point, \( P_i \) reads the letter \((q_a, q_b)\) from the alphabet of summaries \((Q \times Q)\).

When the depth of the stack is \( d + 1 \), \( P_i \) can simulate a popper context of depth \( d + 1 \) in a similar manner.

**Pushdown \( P'_i \): At most \( k \) consecutive contexts of depth at least \( d + 1 \).** \( P_i \) iterates the simulation of contexts relative to stack \( i \) of the MPDS \( M \) by guessing a summary \((q_a, q_b)\), simulating \( M \) and reading the guessed summary when the simulation is successful. The language of \( P_i \) contains the summary of the projection on stack \( i \) of every possible computation of the MPDS \( M \). Since we are only interested in \((k, d)\) bounded budget computations, we only need to consider simulations of sequences of compatible contexts in which at most \( k \) consecutive contexts of depth at least \( d + 1 \) are allowed. The extra symbol we use to delimit the upper (depth larger than \( d + 1 \)) from the lower (depth smaller that \( d \)) parts of the stack increments the depth of stacks that are larger than \( d \). Therefore, we should only focus on sequences of compatible contexts of \( P_i \) in which at most \( k \) consecutive contexts of depth at least \( d + 2 \) are allowed. The set of summaries of these sequences exactly corresponds to the language \( L_{(k,d+1)}(P_i) \). According to lemma 6.1.1, we can build in polynomial time in the size of \( P_i \) a PDA \( P'_i \) such that \( L_{k+d+3}(P'_i) = L_{(k,d+1)}(P_i) \).

**Pushdown \( P_0 \): Consistency check.** Since pushdown automata \( P_i, 1 \leq i \leq n \) are running independently from each other, we need to check that the shuffling of their summaries produces a consistent run of the MPDS. In other words, we should be able to glue the summaries together in order to form a valid run of the MPDS. Moreover, we are interested in all possible consistent runs that can reach state \( q_{final} \). For these two reasons, we would like to only consider summaries of the form \((q_1^1, q_1^b) \cdot (q_2^1, q_2^b) \cdot \ldots \cdot (q_m^1, q_m^b) \in (Q \times Q^*)\) such that:
• The summary must at least indicate a state at the beginning and at the end of the run: \( m \geq 1 \).
• The very first state is the initial state of \( M \): \( q_1^0 = q_{\text{init}} \).
• The last state of a context is identical to the initial state of the next context: \( q_i^b = q_{i+1}^0 \) for every \( i : 1 \leq i < n \).
• The final state is \( q_{\text{final}} \), i.e. \( q_m^b = q_{\text{final}} \).

This set of summaries is a regular language \( L_{\text{cons}} \) which can be generated by a PDA \( P_0 \) with a stack of depth 0 and we have \( L_{\text{cons}} = L(P_0) = L(\lambda)(P_0) \).

**Correctness of the reduction.** Assume that the state \( q_{\text{final}} \) is reachable in \( M \) by a \((k,d)\) bounded budget computation \( \pi \). By definition of a budget bounded computation, we should have that, for every stack \( i : 1 \leq i \leq n \), if \( \downarrow(\pi,i) = \pi_1^i \bullet \pi_2^i \bullet \cdots \bullet \pi_{\ell_i}^i \) is the maximal sub-sequence of contexts in \( \pi \) belonging to the stack \( i \), then \( \downarrow(\pi,i) \) can be written as a compatible sequence \( p_1^1 \bullet p_2^2 \bullet \cdots \bullet p_{\ell_i}^{i+1} \) of length \( \ell_i \geq 0 \) and composed of: (a) Contexts of depth at most \( d \), and (b) Blocks of contexts of size (at most) \( k \) and depth \( d \). If, for some \( j : 1 \leq j \leq \ell_i \), \( p_j^i \) is a block of contexts of size (at most \( k \)) and of depth \( d \), then it can be written as a compatible sequence of at most \( k \) contexts of depth at least \( d + 1 \) delimited by a pusher context of depth \( d \) at its beginning and by a popper context of depth \( d + 1 \) at its end. As a consequence, every projection of the computation \( \pi \) on stack \( i \) can be written as a compatible sequence of contexts \( \downarrow(\pi,i) = \sigma_1^i \bullet \sigma_2^i \bullet \cdots \bullet \sigma_{\ell_i}^i \) such that:

- For every \( j : 1 \leq j \leq \ell_i \), \( \sigma_j^i \) is either a context of depth at most \( d \), a context of depth at least \( d + 1 \), a pusher context of depth \( d \) or a popper context of depth \( d + 1 \).
- Contexts of depth at least \( d + 1 \) can be grouped into blocks of contexts of size at most \( k \).

A summary of the projection of \( \pi \) on stack \( i \) is given by \( \text{summary}(\downarrow(\pi,i)) = \text{summary}(\sigma_1^i) \cdot \text{summary}(\sigma_2^i) \cdot \ldots \cdot \text{summary}(\sigma_{\ell_i}^i) \) a word from \((Q \times Q)^*\), a word composed of letters from the alphabet of summaries \( Q^2 \). A summary of the full computation \( \pi \) is an interleaving of the summaries of the projections of \( \pi \) on each stack \( i \), that is:

\[
\text{summary}(\pi) \in \bigcup \{ \text{summary}(\downarrow(\pi,1)), \ldots, \text{summary}(\downarrow(\pi,n)) \} \quad (6.1)
\]

Let \( \Pi_{(k,d)}(M) \) be the set of all possible \((k,d)\) bounded budget computations of \( M \). The summary of every \((k,d)\) budget bounded computation of \( M \) that reaches \( q \) is in the set of consistent summaries, that is:

\[
\{ \text{summary}(\pi) \mid \pi \in \Pi_{(k,d)}(M) \} \subseteq L_0(P_0) \quad (6.2)
\]

Moreover, for every stack \( i : 1 \leq i \leq n \), \( L(P_i) \) contains the summary of every possible compatible sequence of contexts of \( M \) relative to stack \( i \) that can be decomposed into contexts of depth at most \( d \) and blocks of contexts of depth \( d \) and of arbitrary size. On the other hand, the language \( L_{(k,d+1)}(P_i) \) contains the subset of summaries in \( L(P_i) \) in which blocks of contexts of depth \( d \) are of
size at most \( k \). Thus, \( L_{(k,d+1)}(P_i) \) includes a summary of the projection \( \perp(\pi,i) \) of any \( (k,d) \) budget bounded computation \( \pi \). In other words, we have that:

\[
\{ \text{summary}(\perp(\pi,i)) \mid \pi \in \Pi_{(k,d)}(M) \} \subseteq L_{(k,d+1)}(P_i) \quad (6.3)
\]

We deduce from (6.1), (6.2) and (6.3) that:

\[
L(P_0) \cap \bigcup (L_{(k,d+1)}(P_1), \ldots, L_{(k,d+1)}(P_n)) \neq \emptyset
\]

which is equivalent to

\[
L_0(P_0) \cap \bigcup (L_{(k+d+3)}(P'_1), \ldots, L_{(k+d+3)}(P'_n)) \neq \emptyset
\]

The reverse direction is straightforward.

### 6.4 LTL Model Checking

We consider in this section the linear-time model checking problem for bounded budget MPDS. We consider that we are given an \( \omega \)-regular property \( \phi \) expressed in linear-time propositional temporal logic (LTL) [80] from a finite set atomic properties \( \text{Prop} \). Moreover, we assume that we are given a MPDS \( M = \langle n, Q, \Gamma, \Delta, q_{\text{init}} \rangle \), a budget \( (k,d) \in \mathbb{N}^2 \), and a state labeling function \( l : Q \rightarrow 2^\text{Prop} \) that associates to each state in \( M \) the set of atomic propositions that holds in it. The \( (k,d) \)-budget-bounded model checking of \( M \) against \( \phi \) checks whether all infinite \( (k,d) \)-budget-bounded computations of \( M \) satisfy \( \phi \).

We show in the rest of this section the following theorem:

**Theorem 6.4.1.** LTL model checking of bounded budget MPDS is \( \text{EXPTIME} \)-complete.

Notice that any pushdown automaton \( P = \langle Q, \Sigma, \Gamma, \delta, I, F \rangle \) can be simulated by a \( (0,0) \) bounded budget MPDS \( M = \langle 1, Q, \Gamma, \Delta, q_{\text{init}} \rangle \) that operates on one unique stack. More precisely, the set of transitions \( \Delta \) of \( M \) contains a copy of the transitions \( \delta \) of \( P \) applied to the unique stack 1 together with the set of initial transitions \( \{(q_{\text{init}}, 1, q) \mid q \in I\} \). Based on this remark and on the result [27] showing that LTL model checking of pushdown automata is \( \text{EXPTIME} \)-hard, we deduce the lower bound of Theorem 6.4.1.

To show that model checking the \( (k,d) \) bounded budget MPDS against the LTL formula \( \phi \) is in \( \text{EXPTIME} \), we adopt an automata-based approach similar to the one used in [27] to solve the analogous problem for pushdown automata. It is well-known [91] that we can construct in exponential time a Büchi automaton \( B_{\neg \phi} \) accepting the negation of \( \phi \) such that the size of \( B_{\neg \phi} \) is exponential in the length of the formula \( \phi \). Then, we can compute the product of the MPDS \( M \) and the Büchi automaton \( B_{\neg \phi} \) to obtain
a MPDS $M'$ with a Büchi accepting set of states $F$. The size of $M'$ is the product of the size of $M$ and the size of $B_\phi$. More precisely, we have that $|M'| \in O(|M| \times |B_\phi|) = O(|M| \times c_1^2|\phi|)$ for some constants $c_1$ and $c_2$. The formula $\phi$ is satisfied by very possible $(k,d)$ bounded budget run of the MPDS $M$ if and only if there are no accepting $(k,d)$ bounded budget runs of the Büchi MPDS $M'$. Therefore, checking whether every $(k,d)$ bounded budget run of the MPDS $M$ satisfies $\phi$, boils down to checking for emptiness of the $(k,d)$ budget bounded Büchi MPDS $M'$. In other words, we need to check whether there exists an infinite $(k,d)$ budget bounded computation of $M'$ that visits infinitely often its set of final states. This corresponds to the repeated state reachability problem.

Given a MPDS $M = \langle n, Q, \Gamma, \Delta, q_{init} \rangle$, a budget $(k,d) \in \mathbb{N}^2$ and a set of accepting states $F \subseteq Q$, the budget-bounded repeated states reachability problem asks whether there is an infinite computation $\pi = c_0d_0c_1t_1 \ldots$ such that $c_0 = \langle q_{init}, e \rangle$ and, for every $i \geq 0$, there is a $j \geq i$ such that $State(c_j) \in F$. The following lemma, which proof can be found at the end of this section, is useful to rest of our proof:

**Lemma 6.4.2.** Let $(k,d)$ be two natural numbers, $M = \langle n, Q, \Gamma, \Delta, q_{init} \rangle$ a multi-pushdown system, and $F \subseteq Q$ a set of states. It is possible to construct a Büchi finite-state automaton $A$ such that $M$ has a $(k,d)$ budget-bounded infinite computation that visits a state in $F$ infinitely often if and only if the language $L^\omega(A)$ is not empty. Moreover, the size of $A$ is $O(|M|^{c(n+1)(k+1)(d+1)})$ for some constant $c$.

We saw that the $(k,d)$-budget-bounded model checking of $M$ against $\phi$ amounts to check the repeated reachability of any of the states of some subset $F$ in the MPDS $M'$. According to Lemma 6.4.2, we can build a Büchi automaton $A'$ that reduces the latter problem to checking the emptiness of the language $L^\omega(A')$. Moreover the size of $A'$ is such that (for some constant $c'$):

$$|A'| = O(|M'|^{c(n+1)(k+1)(d+1)}) = O\left(|M| \times c_1^2|\phi|^{c(n+1)(k+1)(d+1)}\right)$$

$$= O\left(|M|^{c'(n+1)(k+1)(d+1)}(|\phi|+1)\right)$$

Checking repeated reachability of a finite-state Büchi automaton is polynomial in the size of the automaton. We deduce that the problem of $(k,d)$-budget-bounded model checking of $M$ against $\phi$ is in EXPTIME.

**Proof of lemma 6.4.2.** Let $(k,d)$ be two natural numbers, $M = \langle n, Q, \Gamma, \Delta, q_{init} \rangle$ a multi-pushdown system, and $F \subseteq Q$ a set of states.

Similar to the upper-bound proof of the reachability problem (Section 6.3), we build a pushdown automaton $P_i$ for every stack $i: 1 \leq i \leq n$. The pushdown automaton $P_i$, $i: 1 \leq i \leq n$, simulates the projection $\perp (\pi,i)$ on stack $i$ of every possible infinite computation $\pi$ of the MPDS $M$. The simulated projection

114
is an infinite compatible sequence of contexts such that each context in the compatible sequence can be either (a) a context of depth at most \(d\), (b) a context of depth at least \(d+1\), (c) a pusher of depth \(d\), or (d) a popper of depth \(d+1\).

The pushdown \(P_i\) simulates the compatible sequence of contexts by guessing, depending on the current depth of the stack, the summary of the next context. The summary of a context \(\pi = c_0l_0c_1t_1 \ldots t_{m-1}c_m\) is defined as the pair \((\text{State}(c_0), \text{State}(c_m))\) composed of the state of the pushdown at the beginning and at the end of the context. For the purpose of this proof, we extend the notion of summary by incorporating the information on whether \(P_i\) had visited any of the states from the set \(F\) during its simulation. We call the extended summary of a context \(\pi = c_0l_0c_1t_1 \ldots t_{m-1}c_m\) its interface and we define it as \(\text{interface}(\pi) := (\text{State}(c_0), f, \text{State}(c_m))\) where \(f\) is equal to 1 if there is a \(j : 1 \leq j \leq m\) such that \(\text{State}(c_j) \in F\) and \(f\) is equal to 0 otherwise. The interface of an infinite sequence of contexts is defined as the infinite word composed of the interfaces of each one of its contexts. The interface of an infinite compatible sequence of contexts and blocks is defined in a similar manner.

Each pushdown \(P_i, i : 1 \leq i \leq n\), simulates a possible projection \(\bot(\pi, i)\) on stack \(i\) of a run \(\pi\) of the MPDS \(M\). The simulation proceeds by (i) guessing an interface \((q_1,f,q_2)\) and guessing the type of context to be simulated, (ii) running the MPDS \(M\) from state \(q_1\) and restricted to the transitions relative to stack \(i\), (iii) making sure that the type of context being run corresponds to the guessed type, and (iv) validating the simulation by checking that state \(q_2\) has been reached and that some state from \(F\) has been visited if \(f\) is equal to 1. The pushdown iterates steps (i), (ii), (iii) and (iv) for every context that composes the projection \(\bot(\pi, i)\). Moreover, at each iteration, if the guess is validated, then the pushdown \(P_i\) reads the letter \((q_1,f,q_2)\) from the finite alphabet of interfaces \(\{(q_1,f,q_2) \mid q_1,q_2 \in Q, f \in \{0,1\}\}\).

Since we are only interested in infinite computations of \(M\) that are \((k,d)\) budget bounded, at most \(k\) consecutive contexts should be allowed when a stack goes over the depth \(d\). Therefore, and similar to the upper-bound proof of the reachability problem in Section 6.3, we are only interested in simulations by \(P_i\) that produce at most \(k\) consecutive contexts of depth of at least \(d+2\) (an extra stack symbol is used to separate the part of the stack that is below depth \(d\) from the upper part). Interfaces generated by this subset of simulations corresponds exactly to the language \(L_{(k,d+1)}^\omega(P_i)\).

Consider a \((k,d)\) budget bounded infinite computation \(\pi\) of the MPDS \(M\). The computation \(\pi\) is an interleaving of its projections \(\bot(\pi, i), i : 1 \leq i \leq n\). As a consequence, the interface of a run \(\pi\) is a possible shuffle of the interfaces of the projections of \(\pi\) on every stack \(i : 1 \leq i \leq n\), i.e. \(\text{interface}(\pi) \in \sqcup_{i=1}^n(\text{interface}(\bot(\pi, 1)), \ldots, \text{interface}(\bot(\pi, n)))\). We can assume that \(\pi\) is written as a sequence of contexts of types a, b, c and d (listed earlier) and so are
its projections $\perp(\pi, i)$ for every stack $i : 1 \leq i \leq n$. The interface of the projection $\perp(\pi, i)$ is then an infinite word and a member of the language $L_{(k,d+1)}^{\omega}(P_i)$ generated by the constructed pushdown $P_i$. We deduce that the word-interface $\text{interface}(\pi)$ of the $(k,d)$ budget-bounded computation $\pi$ is a member of the language $\bigsqcup(L_{(k,d+1)}^{\omega}(P_1), \ldots, L_{(k,d+1)}^{\omega}(P_n))$.

Let $w = (q_1^0, f_1, q_1^1)(q_2^0, f_2, q_2^1) \ldots$ be an infinite word from $\bigsqcup(L_{(k,d+1)}^{\omega}(P_1), \ldots, L_{(k,d+1)}^{\omega}(P_n))$. In order for $w$ to be the interface of some run $\pi$ of the MPDS $M$, it has to be consistent, i.e. it has to meet the following two conditions: (i) $q_i^0 = q_{init}$, and (ii) $q_i^b = q_i^{b+1}$ for every $i \geq 0$. Moreover, an infinite computation $\pi$ of $M$ visits infinitely often states in the set of accepting states $F$ if and only if its interface $\text{interface}(\pi) = (q_1^0, f_1, q_1^1)(q_2^0, f_2, q_2^1) \ldots$ is consistent and is such that, for every $j \geq 0$, there is an $l \geq j$ with $f_l = 1$. It is possible to build a Büchi automaton $A_0$ that precisely accepts this set of interfaces. Moreover, the size of $A_0$ is quadratic in the size of $M$.

Thus, $M$ has a $(k,d)$ bounded budget infinite computation that visits infinitely often some states in $F$ if and only if there is a word in $L(A_0) \cap \bigsqcup(L_{(k,d+1)}^{\omega}(P_1), \ldots, L_{(k,d+1)}^{\omega}(P_n))$.

According to lemma 6.1.1, we can build in polynomial time a pushdown automaton $P'_i$ which size is polynomial in the size of $P_i$ and such that $L_{k+d+3}^{\omega}(P'_i) = L_{(k,d+1)}^{\omega}(P_i)$. By encoding the content of the stack in the control state, we can build an automaton $A_i$ such that $L(A_i) = L_{k+d+3}^{\omega}(P'_i)$. The alphabet of the automaton $A_i$ is identical to the alphabet of $P'_i$, that is the finite alphabet of interfaces $\{((q_1, f, q_2) | q_1, q_2 \in Q, f \in \{0, 1\}\}$ which size is polynomial in the size of $M$. On the other hand, the set of states of $A_i$ is defined as the set $Q_{A_i} := \{(q, w) | q \in Q_{P'_i}, w \in \Gamma^{\leq k+d+3}\}$ which size is in $O(|Q_{P'}| \times |\Gamma|^{k+d+3}) = O(|M|^{k+d+3})$.

We build the cartesian product $A$ of the automata $A_0, A_1, \ldots, A_n$ such that $L^{\omega}(A) = L^{\omega}(A_0) \cap \bigsqcup(L^{\omega}(A_1), \ldots, L^{\omega}(A_n))$. Automaton $A$ exactly accepts the interface of every $(k,d)$ bounded budget and accepting run of $M$. The size of $A$ is in $O(|A_0| \times |A_1| \times \ldots \times |A_n|) = O(|M|^2 \times |M|^{(k+d+3)} \times \ldots \times |M|^{(k+d+3)}) = O(|M|^2 \times |M|^{n(k+d+3)}) = O(|M|^{c(n+1)(k+1)(d+1)})$ for some constant $c$. □

6.5 Code-to-Code Translation

We present in this section an automatic code-to-code translation that implements our verification approach. We propose a translation scheme takes as input a concurrent program $P$ and a budget $(k,d)$ and builds a sequential program $P'$ that simulates the behaviors of the concurrent program under the budget restriction.

The idea of transforming a concurrent program into a sequential one, or sequentialization, has already been used in the past to analyze concurrent pro-
grams. The motivation behind sequentialization approaches is to leverage the power of existing verification tools for sequential programs and use them to verify concurrent programs. In a sequential program, the unique thread has a single stack. On the other hand, each thread has a stack in a concurrent program. A sequentialization scheme should be able to produce a sequential program that simulates, using only one stack, a large subset of the interleavings of a concurrent program.

The code-to-code translation we propose is inspired by three sequentialization reductions [83, 65, 63] that aim at simulating the behaviors of a concurrent program up to a bounded number of context switches. More specifically, the three approaches that we mention are used to check the state reachability problem of concurrent programs within a bounded number of context switches. Qadeer and Wu have defined in [83] an explicit reduction which outputs a sequential program that keeps track of the local state of all the concurrent threads. Later, Lal and Reps proposed in [65] an eager approach that does not keep track of the local state of all the concurrent threads. Instead, the eager approach runs threads independently from each other and guesses valuations of the shared variables at context switches. Finally, La Torre et al. defined in [63] a lazy reduction which adopts the same idea as [65] of keeping track of the local store of only one thread. However, instead of guessing valuations of the shared store at context switches, the lazy approach iteratively computes them.

Outline of the section. We start by describing for each of the explicit, eager and lazy approaches how the reduction from a concurrent program to a sequential one works. Then we present our automatic code-to-code translation and show how we can use it to check state reachability under the budget restriction. Finally, we show how we can extend our code-to-code reduction to verify termination of budget bounded concurrent programs.

6.5.1 The explicit approach

Qadeer and Wu propose in [83] the Keep It Simple and Sequential (KISS) approach. The KISS approach transforms a concurrent program $P$ composed of a number of threads into a sequential program $P'$. The stack of the sequential program $P'$ is divided into contiguous blocks. Each contiguous block is the stack of one of the concurrent threads in $P$. In particular, the topmost block is the stack of the current running thread. In order to resume the execution of a thread which stack is in a lower block, $P'$ needs to pop all the upper blocks and consequently terminate the execution of the corresponding threads.

The reduction to a sequential program is explicit because it keeps track of the local and global variables of every thread of the concurrent program $P$. Moreover, the output sequential program $P'$ maintains a copy of the set of shared variables and uses two instrumentation variables $\text{Raise}$ and $ts$. The
variable \textit{Raise} is used to force return every pending procedure of the current
running thread when the sequential program decides to terminate the thread.
In other words, this variable allows to pop up the topmost block in the stack.
The variable \textit{ts} is a multiset with a fixed size. It contains threads that were
forked but not yet scheduled.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure6.1}
\caption{Following the KISS approach, this figure illustrates the simulation of
the run of a concurrent program by a sequential program using a single stack. \textbf{The upper part of the figure} shows the content of the stack of the sequential program
during the simulation of the run. In the initial configuration, thread \textit{T}_1 is the current
running thread, and its stack is the topmost block. In (a), the sequential program let \textit{T}_1
continue its execution. In (b), the simulation decides to pause the execution of \textit{T}_1 and
to schedule the execution of another thread, here \textit{T}_2. In (c), the execution of the current
running thread \textit{T}_2 is terminated, and all its pending calls are popped out from the stack.
In (d), the execution of \textit{T}_1 is resumed. In (e), \textit{T}_2 execution is again put on hold and
a new thread (\textit{T}_3) is scheduled. \textbf{The lower part of the figure} shows the sequence
of contexts per thread and the context switches that occurred during the run. We use
the crossed box \ding{51} to denote that a thread (e.g. \textit{T}_2) has been terminated and cannot
be resumed (its contiguous box has been popped out from the stack). Thread \textit{main}
and \textit{T}_1 are put on hold and are waiting to be resumed, pictured in the figure using the
black triangle \textbf{▼}. The stacks of both threads are contained in the two lower contiguous
boxes in the sequential stack. Finally, we use the symbol \textbf{▽} to distinguish the current
running thread \textit{T}_3, which stack content is in the topmost box of the sequential stack.}
\end{figure}

At every point in time during the run of the sequential program \(P', P'\) may
nondeterministically decide to perform one of the following actions:
\begin{itemize}
\item Continue the execution of the current thread (i.e., the thread which stack
  is in the topmost block),
\end{itemize}
• Terminate the execution of the current thread and pop all the procedure calls pending in its contiguous topmost block, or
• Pause the execution of the current thread and start the execution of a thread forked but not yet scheduled.

The KISS approach produces a sequential program $P'$ out of a concurrent program $P$ such that $P'$ simulates a subset of the behaviors of $P$. Consider for instance the example presented in Figure 6.1. In this example, the concurrent program $P$ is composed of 4 different threads, and the number of context switches that occurred during the run depicted in the figure is equal to 4. In transition $(c)$, the running thread $T_2$ had to be terminated for thread $T_1$ to be resumed. A thread that has been terminated cannot be resumed again because its stack was lost when it was terminated. This constraint limits the number of behaviors that the KISS sequentialization approach can cover. In particular, the KISS approach would be able to analyze a concurrent program composed of two threads up to a maximal number of only two context switches. We present in the next two paragraphs two sequentialization approaches that aim to cover all possible behaviors of the concurrent program up to a given and arbitrary context bound $k$.

6.5.2 The eager approach

Lal and Reps propose in [65] a sequentialization scheme that applies to concurrent recursive Boolean programs. Given a context bound $k$ (per thread) and a concurrent program $P$ composed of $n$ threads, the reduction gives as output a sequential program $P'$ that simulates every possible behavior of $P$ up to $k$ context switches. The main idea of this approach is to run every thread of the program only once and separately from the other threads. To be able to do so, the sequential program initially guesses $k$ valuations of the shared store. These valuations are used when a thread resumes its execution after a context switch happens. Intuitively, the guessed value of the shared store should match the value computed by the execution of another thread. After the simulation of all the threads has terminated, the sequential program $P'$ has to check that the guessed values match the computed ones.

This approach has the advantage of keeping track of the valuation of the local variables of only one thread at a time. Thus, this approach avoids the exponential blow up that results from an explicit approach in which a cross-product of the states of the local variables of each thread has to be considered. Since the shared store is in general smaller than the local store, exploring a state space composed of $k$ copies of shared variables can be more efficient than exploring a state space induced by $n$ local stores. Finally, in the case where the concurrent program $P$ is composed of recursive Boolean programs, Lal and Reps approach allows using one stack to simulate the behaviors of $n$ stacks.
For ease of presentation, consider the run of the concurrent program $P$ depicted in Figure 6.2. $P$ is composed of two threads $T_1$ and $T_2$. Each thread execution is interrupted twice by a context switch. The execution of thread $T_1$ (respectively thread $T_2$) is composed of three contexts (respectively two). The scheduling of the threads in the run can be informally written as the sequence $\langle T_1, T_2, T_1, T_2, T_1 \rangle$. Thread $T_1$ is scheduled first and starts with the initial valuation $s_0$ of the shared store and the initial valuation $l_1$ of its local store. At the end of the first context, $T_1$ has the valuation $l_2$ for its local store and computes the valuation $s_1$ of the shared store. A context switch then happens and thread $T_2$ is scheduled. Thread $T_2$ starts its execution from the valuation $s_1$ of the shared store. A context switch happens for $T_2$ when the valuation of the shared store becomes $s_2$. At this point, $T_1$ resumes its execution with the (unchanged) valuation $l_2$ of its local store and with the new valuation $s_2$ of the shared store produced by the execution of the first context of thread $T_2$. A second context switch for $T_1$ happens when the shared store has the valuation $s_3$. Thread $T_2$ then resumes its execution with the new valuation $s_3$ of the shared store. The rest of the run is depicted in the figure.

Given the concurrent program $P$ of Figure 6.2 and the number of context switch (per thread) $k = 2$, Lal and Reps approach produces a sequential program $P'$ which can simulate every possible context sequence of the concurrent program for this number of context switches. For clarity, let us assume that the sequence of contexts simulated by $P'$ is $\langle T_1, T_2, T_1, T_2, T_1 \rangle$. The sequential program $P'$ starts by guessing $k = 2$ valuations $v_1$ and $v_2$ of the global store. Then, $P'$ proceeds by running $T_1$ from the initial valuation of the shared store $s_0$. The run of $T_1$ is performed independently from $T_2$. Thus, $P'$ only works on one local store at a time and requires the use of only one stack. Since the analysis is done for two context switches per thread and more precisely for the context sequence $\langle T_1, T_2, T_1, T_2, T_1 \rangle$, we can assume that the execution of $T_1$ has the shape depicted in Figure 6.3. In particular, the run of $T_1$ is composed of three contexts. During the run of $T_1$, the thread nonde-deterministically decides at two different points in time to perform a context switch. Instead of being performed, context switches are simulated by performing a shared store jump in which the shared store is set to one of the guessed valuations. For instance, when $T_1$ decides to perform its first context switch, $T_1$ first saves a copy of the global store in $s_1$ and then sets it to
the first guessed valuation $v_1$. The guessed valuation $v_1$ allows us to continue
the execution of $T_1$ without waiting for $T_2$ to compute the new shared store.

The first context switch is depicted in the
figure by the jump $j_1$. $T_1$ resumes its ex-
ecution till it nondeterministically decides
to perform the second context switch. The
second guessed valuation $v_2$ is then used
to resume the execution of $T_1$ (jump $j_2$ in
the figure). Now that $T_1$ has been run,
we can proceed with the execution of $T_2$.
Since the context sequence simulated here
is $\langle T_1, T_2, T_1, T_2, T_1 \rangle$, $T_2$ starts with the valua-
tion of the shared store produced by $T_1$ at the
end of its first context and saved in $s_1$. When
$T_2$ decides nondeterministically to perform
the first context switch, it saves the current
shared store in $s'_2$, and sets the shared store
to the valuation $s_3$ saved by $T_1$ at the end of
its second context. Then $T_2$ continues its ex-
ecution till it decides nondeterministically to
make the second context switch. At which
point it saves the shared store in $s'_4$ and stops
its execution. For this run of $P'$ to be a valid
run of $P$, the guessed valuations $v_1$ and $v_2$
should respectively match the computed valuations $s'_2$ and $s'_4$. If the computed
valuations do not match the guessed ones, the run of the sequential program
$P'$ is simply ignored. The eager approach, consists in considering all possible
combinations of the of the $k$ guessed valuations of the shared store, including
valuations that might not be reachable by the concurrent program. In order to
avoid these particular valuations, another approach called the lazy approach,
consists in iteratively computing the set of valuations of the shared store that
a thread can compute after executing one single context from a given initial
shared store. While Lal and Reps detail the eager approach for both non-
recursive and recursive Boolean programs, they describe the lazy approach
only for the case of non-recursive Boolean programs.

6.5.3 The lazy approach

Given a concurrent program $P$ and a context bound $k$, La Torre et al. propose
in [63] a reduction from $P$ to a sequential program $P'$ such that the set of states
reachable by $P'$ is identical to the set of states reachable by $P$ within $k$ context
switches.
The reduction of La Torre et al. follows the same idea as of the eager approach of Lal and Reps of keeping track of the local state (local store valuation and stack content) of only one thread. This allows the analysis to be compositional, i.e., the analysis of the concurrent program is reduced to the separate analysis of the threads composing the program.

The eager approach of Lal and Reps guesses the $k$ valuations of the shared store needed to run the threads independently. As a consequence, their approach considers simulation-runs of the sequential program $P'$ with valuations of the shared store that are not necessarily reachable in the concurrent program $P$. Naturally, these particular runs are discarded by the simulation since they do not simulate a real run of the concurrent program $P$. This might be problematic in the case where the set of possible valuations of the shared store is much bigger than its reachable subset. Therefore, an approach that only considers reachable valuations of the shared store is highly desirable. Following this idea, La Torre et al. propose the lazy approach. Instead of being guessed, the valuations of shared stores at context switches are computed. The main idea behind the reduction is to iteratively compute valuations of the shared store from one thread to another. Since only one local store is tracked at a time, individual threads have to reconstruct their local stores when they are again scheduled.

To illustrate this approach, consider again the run of the concurrent program $P$ in Figure 6.2. Instead of describing the transformation from the concurrent program $P$ to the sequential program $P'$, we take the path in the concurrent program depicted in the figure, and describe how the sequential program $P'$ will discover it (Figure 6.4). Recall that the sequence of contexts of this example is $\langle T_1; T_2; T_1; T_2; T_1 \rangle$. The sequential program $P'$ has $k = 4$ (the number of context switches in the run) extra copies $s_1, s_2, s_3$ and $s_4$ of the shared store $s$. These copies are used to store the valuations of the shared store computed at the end of each context before a context switch happens. The simulation is composed of five runs, each corresponds to a run in which $T_1$ or $T_2$ simulates a particular context from the sequence of contexts $\langle T_1; T_2; T_1; T_2; T_1 \rangle$.

The first run ((1) in Figure 6.4) consists in simulating the first context of thread $T_1$. We run $T_1$ from the initial valuation of the shared store $s_0$. We instrument the code of every thread of the concurrent program in such a way that a context switch can happen non-deterministically before the execution of any statement. Thus, $T_1$ can perform a context switch at any point during its execution. If $T_1$ decides to make a context switch, then the valuation of the shared store computed by $T_1$ is saved into $s_1$ (i.e., $s_1 := s$).

The second run ((2) in the figure) is a simulation of the first context of $T_2$. Since we can keep track of the local state of only one thread, we need to terminate $T_1$ before running $T_2$. After terminating $T_1$, we run $T_2$ from the valuation $s_1$ computed and saved by $T_1$. At any point during its execution, thread $T_2$ can non-deterministically decide to perform a context switch. If so,
the shared store computed by $T_2$ is saved into $s_2$ (i.e. $s_2 := s$) and thread $T_2$ is terminated.

The third run ((3) in the figure) consists of simulating the second context of $T_1$. At this point, we managed to compute (and not guess) a valuation of the shared store $s_2$ that can be used to resume the execution of $T_1$ after its first context switch. Unfortunately, since $T_1$ has been terminated in order to run $T_2$ (and compute $s_2$), we have lost the local store $l_2$ that $T_1$ had built at the end of its first context. Instead of trying to rebuild the same local store, we proceed by computing any local store that thread $T_1$ can build when it computes the valuation $s_1$ of the shared store. More precisely, we run $T_1$ from scratch by (i) setting the shared store to its initial value $s_0$ ($s := s_0$), (ii) let $T_1$ run (the first context), and (iii) non-deterministically check during the run if the shared store matches the valuation $s_1$. If the condition $s \equiv s_1$ is met, then we can perform the jump ($j_1$ in the figure) from the reconstructed valuation of the shared store $s_1$ to the valuation $s_2$ computed by $T_2$. After setting $s := s_2$, we let $T_1$ run its second context. Again, $T_1$ can non-deterministically and at any point perform a context switch by terminating its execution and saving the computed shared store in $s_3$ ($s_3 := s$).

Observe that the local store $l'_2$ computed by $T_1$ before the jump might be different from the local store $l_2$ that was computed in the first run of $T_1$. However, this does not alter the correctness of the simulation. In fact, the correctness of the run simulated so far only depends on whether the valuation of the shared store computed at the end of each context matches the valuation used at the beginning of the next context.
Thread $T_2$ can then be switched on to run its second context (run (4) in the figure). Since we lost the local store $l_3$ that $T_2$ had at the end of its first context, we need to recompute another compatible local store. Therefore, we run $T_2$ from the valuation $s_1$ of the shared store till it matches the precomputed valuation $s_2$ (i.e. till $s \equiv s_2$). The shared store jump ($j_3$) can then be performed by setting $s$ to $s_3$, the valuation computed by $T_1$ at the end of its second context. Thread $T_2$ can then execute its second context till it decides at some point to perform its second context switch by terminating its execution and saving the computed shared store in $s_4$.

The last run of the simulation ((5) in the figure) consists of running the third context of $T_1$. Again, we need to rebuild the local store of $T_1$ when it was switched out before executing the third context. To that purpose, it is not sufficient to run $T_1$ from scratch till it computes a shared store identical to the valuation $s_3$ it had at the end of its second context. Instead, $T_1$ needs to (i) First recompute the valuation $s_1$ it had before its first context switch, (ii) perform the jump ($j'_1$ in the figure) from $s_1$ to $s_2$, (iii) run its second context till the shared store becomes $s_3$, (iv) perform the jump ($j'_2$ in the figure) from $s_3$ to $s_4$, and (v) finally start running the third context.

Besides the $k$ copies of the shared store, the sequential program produced by the lazy transformation makes use of other instrumentation variables: A Boolean variable terminate, two integer variables $cx$ and $ix$ that range over the interval $1, \ldots, k$ and $k$ integer variables $t_1, \ldots, t_k$ that take values in the interval $1, \ldots, n$. Similar to the variable Raise in the explicit (KISS) approach, the variable terminate is used to exit every pending procedure of the running thread and to terminate the execution of the thread. The variable $cx$ indicates the context to be run in the current part of the simulation, while $ix$ indicates the current running context. Finally, each variable $t_i$, for $i : 1 \leq i \leq k$, is the index of the thread scheduled in the $i^{th}$ context.

### 6.5.4 Budget bounded code-to-code translation

Given a budget $(k, d) \in \mathbb{N}^2$ and a concurrent program $P$, we propose an automatic reduction to a sequential program $P'$ that simulates all $(k, d)$ budget bounded behaviors of the concurrent program $P$. For each thread $T$ of the concurrent program $P$, the reduction that we propose allows (i) an unbounded number of context switches when the depth of the stack of $T$ is smaller than $d$, and (ii) at most $k$ context switches when the depth of the stack of $T$ is strictly larger than $d$. In order to combine both features, our reduction borrows ideas from both the explicit approach of [83] and the lazy approach of [63].

In order to properly describe our code-to-code translation, we start by giving the syntax of the input concurrent program. Then, we show how we can distinguish in the code of the concurrent program between the case where the depth of the stack of a thread is smaller than $d$ and the case where it is larger.
than \(d + 1\). After defining the global structure of the sequential program output \(P'\), we show how \(P'\) is built in order to allow an unlimited number of context switches for each thread whose stack depth is smaller than \(d\). Later, we describe how we adopt the ideas of the lazy approach of [63] to allow at most \(k\) context switches when the depth of the stack of a thread is larger than \(d + 1\). Finally, we briefly present the features of the prototype implementation of our approach and discuss the experimental results.

**Syntax.**

We use a minimal C-like syntax in order to define a sequential program. More precisely, we assume that a sequential program (or simply thread) \(T\) is defined as the tuple \(\langle G, F_1, \ldots, F_m \rangle\) composed of a finite set of *global* variables \(G\) and a number \(m \geq 1\) of procedures \(F_i\). Exactly one of the procedures \(F_i\), called \(\text{main}\), constitutes the entry point to the execution of the sequential program \(T\). Each procedure is itself defined as a tuple \(\langle L, \text{statements} \rangle\) where \(L\) is a finite set of variables *local* to \(F\) and where \(\text{statements}\) is a sequence of statements. Statements that we consider cover the basic features of an imperative programming language such as variable assignments, arithmetic and Boolean expressions, conditional and iterative control structures and procedure calls. Furthermore, we enrich this set with \text{assume}, \text{assert}, \text{nop} (no-operation) and \text{atomic} statements. The detailed syntax of the statements can be found in [9].

We define a concurrent program \(P\) as a tuple \(\langle S, T_1, \ldots, T_n \rangle\) composed of a finite set of *shared* variables \(S\) and a finite number \(n \geq 1\) of threads \(T_1, \ldots, T_n\). An example of a simple concurrent program is shown in Figure 6.5.

```
1    int x = 0;
2
3    process p1:
4        void main(){
5            x = 1;
6            x = 2;
7    }

8    process p2:
9        void main(){
10           x = x + 2;
11           assert(x != 1);
12    }
```

*Figure 6.5. A simple concurrent program*

Observe that all the variables that we consider are either Boolean or integer variables. In the examples considered in our experiments, we have only considered bounded integers, although our code-to-code translation can also be carried over for programs with variables of (unbounded) integer type.

**Inlining.**

Depending on the depth of the stack of a thread, there are two constraints on the number of allowed context switches. Therefore, we need to be able to distinguish, in the code of every thread \(T\) composing the concurrent program \(P\), the case where the depth of the stack of the thread is smaller than \(d\) and the case where it is larger than \(d + 1\). In order to be able to distinguish between these two cases, we iteratively *inline* \(d\) times the code of the main procedure of each thread. The inlining of a procedure consists in replacing every procedure call that appears in the code of the calling procedure by the instantiated code.
of the called procedure. Naturally, the inlining procedure will add the local variables of the called procedure to the set of local variables of the calling procedure. Given a thread $T$, we use $I(T)$ to denote the thread that we obtain after inlining of the main procedure of $T$ and we use $I^d(T) := (I \circ \ldots \circ I)(T)$ to denote the effect of inlining $T$ $d$ times. Thus, the result of inlining $d$ times the main procedure of each thread composing the concurrent program $P = \langle S, T_1, \ldots, T_n \rangle$ gives the concurrent program $P^{\text{inlined}} = \langle S, I^d(T_1), \ldots, I^d(T_n) \rangle$. Based on the concurrent program that we obtain, our code to code translation should allow:

- An unbounded number of context switches when a thread is executing its main procedure,
- At most $k$ context switches when a thread is executing a procedure different from its main procedure.

**Structure of the output program.**

We define the output program of the code-to-code translation as the sequential program $P' := \langle G', \text{main}, F'_1, \ldots, F'_m \rangle$ defined by a set of global variables $G'$, a main procedure and a set of procedures $F'_1, \ldots, F'_m$.

The set of global variables $G'$ is defined as the union of (a) the set $S$ of shared variables of the concurrent program $P$, (b) the set of global variables of every thread $I^d(T_i), i : 1 \leq i \leq n$, and (c) the set of local variables of the main procedure of every thread $I^d(T_i), i : 1 \leq i \leq n$.

Moreover, for instrumentation purposes, we add to the set of global variables $G'$: (a) an integer variables running which range over the set of indices $1, \ldots, n$ of threads, and (b) a Boolean variable ret (similar to the variable Raise in the KISS approach) used to force-exit the execution of the thread currently simulated, (c) a Boolean variable progress set to true whenever a statement from the main procedure of any of the threads is executed.

Furthermore, for each thread $I^d(T_i), i : 1 \leq i \leq n$, we add to $G'$ a program counter $pc_i$ that keeps track of the next statement to be executed in the main procedure of thread $I^d(T_i)$. In the case where $I^d(T_i)$ contains procedure calls, we also add to $G'$ (a) $2k + 1$ copies $S'_j, j : 0 \leq j \leq 2k$, of the shared variables $S$, and (b) two integer variables $\text{phase}_i$ and $\text{phase}'_i$ that range over the interval $0, \ldots, k$. The copies of the shared variables are used to save the valuation of the shared store at context switches, while the phase variables are used to keep track of which current context is being run or reconstructed.

The main procedure of the sequential program $S'$ is designed in such a way that it can simulate the execution of the main procedure of every thread $I^d(T_i), i : 1 \leq i \leq n$ and allows every thread to perform an unbounded number of context switches. The exact encoding of the main procedure is explained in the next paragraph.

Finally, the procedures $F'_i, i : 1 \leq i \leq m$, are composed of all the non-main procedures of all the threads $I^d(T_i), i : 1 \leq i \leq n$, modulo some instrumentation code that we explicit in the remaining of this section.
process Pprime:

```c
int pc1 = 1;
int pc2 = 1;
int running;
x = 0;
bool progress;
```

```c
void scheduler()

while (progress) {
    progress = false;
    // schedule a process
    if (? & & pc1 != 3) {
        running = 1;
        if (pc1 == 1 && ?) {
            x = 1;
            progress = true;
            pc1 = 2;
        }
        if (pc1 == 2 && ?) {
            x = 2;
            progress = true;
            pc1 = 3;
        }
    } else {
        if (? & & pc2 != 3) {
            running = 2;
            if (pc2 == 1 && ?) {
                x = x + 2;
                progress = true;
                pc2 = 2;
            }
            if (pc2 == 2 && ?) {
                assert (x != 1);
                progress = true;
                pc2 = 3;
            }
        }
    }
    // process 1
    if (running == 1) {
            if (pc1==1 & & ?){
                    x = 1;
                    progress = true;
            }
    }
    // process 2
    if (running == 2) {
            if (pc2==1 & & ?){
                    x = x + 2;
                    progress = true;
                    pc2 = 2;
            }
            if (pc2==2 & & ?){
                    assert (x != 1);
                    progress = true;
                    pc2 = 3;
            }
    }
```

Figure 6.6. Transformation of the Program of Figure 6.5

Unbounded number of context switches in the main procedure.

In order to allow an unbounded number of context switches when a thread is executing its main procedure, we use in the sequential program $P'$ a scheduling loop that contains every statement of the main procedure of every thread $I^d(T_i)$, $1 \leq i \leq n$. At each iteration of the loop, at most one statement is executed. The choice of the statement to be executed is determined by the choice of the thread to be run and by the current program counter of the chosen thread. To choose which thread should be run next, we use the integer variable `running` which is non-deterministically set at the beginning of every loop to any element in the set $1, \ldots, n$ of thread indices. Moreover, for each thread, $I^d(T_i)$, $1 \leq i \leq n$, we use the variable $pc_i$ which keeps track of the next statement to be executed from the main procedure of thread $I^d(T_i)$.

Figure 6.6 shows the code-to-code translation of the concurrent program shown in Figure 6.5. The concurrent program of Figure 6.5 is composed of two threads, each of them contains only one (main) procedure. In the scheduling loop of the output sequential program, the execution of a statement of one of the two threads is dependent on three parameters: a) the choice of the running thread, b) The value of the program counter, and c) the symbol `?` which encodes a non-deterministic choice. The choice of the running thread is made in lines 14-19 at the beginning of every loop iteration. The program counter
pc\_i of every thread \( I^d(T_i) \), initialized before the simulation starts, is updated each time a statement of thread \( I^d(T_i) \) is executed. Thus, at every loop iteration, a thread is chosen to be run and a number of consecutive statements of the chosen thread, starting from the current program counter, are executed. The non-deterministic choice of executing a statement allows for the execution of a subset of consecutive statements belonging to the same thread, and also to skip the execution of the remaining statements. As a consequence, the non-deterministic choice of executing a statement allows simulating context switches between the threads. Finally, the variable progress is used to enforce that at least one statement is executed at each iteration.

Allowing at most \( k \) context switches outside the main procedure.
Each time a procedure call is made from the main procedure of some thread \( I^d(T_i) \), the stack of thread \( T_i \) becomes larger than \( d + 1 \) and only \( k \) context switches shall be allowed. Therefore, each time a procedure call is made from the main procedure of \( P' \), we activate the lazy approach. When we execute a called procedure, it is meaningless to keep track of the program counter of the thread. Instead, using the variable phase\_i (respectively phase'\_i), we keep track of the context thread \( T_i \) is running (respectively rebuilding). The variable phase\_i (respectively phase'\_i) has the same role as the variable cx (respectively ix) in the reduction of [63].

In the original lazy approach of La Torre et al., each time a thread gets the control back, i.e. each time its execution is resumed, it runs from scratch, i.e. from the time it was first scheduled. More precisely, copies of the shared store are saved in order for the thread to run from scratch and rebuild all the previous contexts it has been through before. We use in our code-to-code translation the same idea, but instead of rebuilding the contexts from scratch, we do it from the point in time where the procedure call in the main has been made. To do so, before each procedure call of a thread \( I^d(T_i) \) from the main procedure, we save the current valuation of the shared variables \( S \) in \( S^2\_\text{phase} \) and set it to the valuation \( S^0 \) it had the first time the thread made this particular procedure call. This corresponds to the code in Figure 6.7.

In the case where the thread is rebuilding its contexts, i.e. when phase'\_i \( \leq \) phase\_i, we allow the called procedures to check non-deterministically and at any point if the current valuation of the shared variables matches the valuation of the shared variables saved at the previous context switches. Namely, we non-deterministically check if \( S \vdash S^2\_\text{phase} \_i + 1 \). If this is the case, we perform the jump from context phase'\_i to context phase'\_i + 1 by incrementing

128
if (!ret && ?) {
    if (phase' < phase_t) {
        if (phase' == 0 && S == S^{1}) {
            phase' = 1;
            S = S^{2};
        }
        ...
        if (phase' == k-1 && S == S^{2k-1}) {
            phase' = k;
            S = S^{2k};
        }
    }
    ...
    if (phase_t == k-1 && S == S^{2k-1}) {
        phase_t = k;
        S = S^{2k};
    } else {
        if (phase_t == k-1) {
            S^{2k-1} = S;
            phase_t = k;
            S = S^{2k};
        } // ret = true;
    }
}

Figure 6.8. **Left:** Reconstruction of local state. **Right:** Context Switches inside a procedure.

phase', and setting S to $S^{2\cdot \text{phase'}}$. The left side of Figure 6.8 shows the code of the context reconstruction check and jump.

If the thread is not reconstructing a context but is simply running a context $\text{phase}_t < k$, i.e., when $\text{phase'} = \text{phase}_t$, we allow the code of the procedure to be non-deterministically interrupted in order to perform a context switch. The valuation of the shared variables is then saved in $S^{2\cdot \text{phase}_t + 1}$ and $\text{phase}_t$ is incremented. Moreover, variable $\text{ret}$ is set to true which will allow to skip any further statement in the called procedure and to return to the scheduling loop of the main procedure. The right side of Figure 6.8 shows the code allowing for a context switch within a procedure.

**Experimental results.**

We implemented the code-to-code translation in a prototype. Both the concurrent program input and the sequential program output are written in the C-like pseudo-code syntax mentioned earlier. The prototype translates the output sequential program from the pseudo-code syntax to the input syntax of three sequential program verification tools we considered, namely CBMC [34], ES-BMC [35] and MOPED [44]. We tested our approach on a number of examples taken from the literature and manually translated to the pseudo-code syntax we mentioned. The time taken by the code-to-code translation was negligible compared to the verification time taken by the sequential program verification tools. We compared the verification time and the results produced by each sequential program verification backend tool with the ones produced by POIROT [64] and ESBMC, two concurrent program verification tools. In half of the examples, the experimental results show that using our code-to-code translation coupled with one of the sequential tools is faster than using the concurrent tools. This was particularly evident when considering examples
where a relatively large number of context switches was needed in order to violate the assertion.

More details concerning the examples we used and the experiments we did can be found in Paper IV while user instructions, binaries and the source code of the examples can be found on the tool webpage [9].

6.5.5 Termination of bounded budget concurrent programs under fairness

In the context bounded model-checking of concurrent programs [82] and related sequentialization schemes [83, 65, 63], at most one of the threads can have an infinite run. This makes context-bounded analysis inappropriate to the class of programs which contain coordinated nontermination, i.e., where the interaction of several processes leads to a nonterminating execution. On the other hand, the bounded budget approach allows executions to have an infinite number of context switches as long as the stack bounds are respected.

We extend the code-to-code translation proposed in the previous section to check for termination under two different notions of fairness, impartiality and strong fairness. Impartiality states that any thread that has not ended should eventually be scheduled and executed. On the other hand, strong fairness states that if a process is infinitely often enabled, then it will infinitely often make some progress.

To express and precisely define the two notions of fairness, we add for each thread $T_i$ of the concurrent program a Boolean flag $en_i$. The variable $en_i$ is set to true unless thread $T_i$ is busy waiting for a lock or unless it ended. Furthermore, we use a Boolean variable $end$. Initialized to false at the beginning of the simulation, the variable $end$ is set to true when all the threads have ended. Using the program counters of the concurrent threads and the Boolean variable $progress$, we can express the fairness conditions in LTL as follows:

**Impartiality:**
\[
\bigwedge_{i:1 \leq i \leq n} \Box((pc_i \neq lastpc_i) \implies \Diamond (running = i \land progress))
\]

**Strong fairness:**
\[
\bigwedge_{i:1 \leq i \leq n} \Box\Diamond (en_i) \implies \Box\Diamond (running = i \land progress)
\]

The bounded budget concurrent program terminates under a fairness condition $\phi$ if the formula $\phi \implies \Diamond (end)$ is satisfied. We use the code-to-code translation proposed earlier, extended with the new instrumentation variables, to output sequential programs in PROMELA that we feed to the SPIN model checker [58] to check for termination under three fairness conditions: No fairness, impartiality and strong fairness. We considered three small examples taken from [19], each presenting nonterminating runs. The first example,
shown in the top of Figure 6.9, has a non-fair nonterminating run, while all its fair runs are terminating. The second example we considered, shown in the bottom of Figure 6.9, has a nonterminating run which requires a synchronization between the two concurrent threads. Finally, we consider a third example that implements a retrying mechanism often used in concurrent data structures. A validation phase before the effect of an operation ensures that there has been no interference from other threads. This might lead to nonterminating executions in which another thread always interferes with the operation. We checked the three examples for termination with the three fairness conditions and found correct results, which provides an experimental validation of our approach.

```c
bool g = false;

process thread1:
void main()
{
    while (!g) {
        skip;
    }
    g = false;
}

process thread2:
void main()
{
    g = true;
    while (g) {
        skip;
    }
    g = false;
}
```

---

```c
bool g = false;

process thread1:
void main()
{
    while (g) {
        g = false;
    }
}

process thread2:
void main()
{
    while (!g) {
        g = true;
    }
}
```

Figure 6.9. **Top**: The program Ex1 which does not have any fair nonterminating runs. **Bottom**: The program Ex2, with fair nonterminating runs.
6.A Appendix

This section contains the proofs of lemmas 6.1.1 and 6.1.2 stated in the preliminaries (Section 6.1).

In order to prove lemma 6.1.1, we need to prove the following lemma first:

**Lemma 6.A.1.** Let \( k \in \mathbb{N} \) and \( P \) be a pushdown automata. It is possible to build in polynomial time a pushdown automaton \( P' \) such that \( L_{k+1}(P') = L(P) \cap \Sigma^{\leq k} \).

In order to prove lemma 6.A.1, let us first recall the notions of context-free grammar and Chomsky normal form.

**Context-Free Grammar.**
A context-free grammar (CFG for short) \( G \) is a tuple \( \langle \mathcal{X}, \Sigma, R, S \rangle \) where \( \mathcal{X} \) is a finite set of non-terminals, \( \Sigma \) is a finite alphabet of terminals, \( R \subseteq \mathcal{X} \times (\mathcal{X} \cup \Sigma)^{\ast} \) is a finite set of production rules, and \( S \in \mathcal{X} \) is the start symbol. We write \( X \to w \) to denote that \( (X, w) \in R \). Given two strings \( u, v \in (\Sigma \cup \mathcal{X})^{\ast} \), we say that \( u \Rightarrow_{G} v \) if there exist a production rule \( (X, w) \in R \) and two words \( y, z \in (\Sigma \cup \mathcal{X})^{\ast} \) such that \( u = yXz \) and \( v = ywz \). A derivation sequence \( \alpha \) of \( G \) is given by \( \alpha = u_{0} \Rightarrow_{G} u_{1} \Rightarrow_{G} \ldots u_{n} \) with \( u_{0} = S, u_{n} \in \Sigma^{\ast} \) and \( u_{i} \in \Sigma \cup \mathcal{X}^{\ast} \). We use \( \Rightarrow_{G}^{*} \) to denote the reflexive transitive closure of \( \Rightarrow_{G} \). The context-free language generated by \( G \) is defined by \( L(G) := \{ w \in \Sigma^{\ast} \mid S \Rightarrow_{G}^{*} w \} \). The size of \( G \) is defined by \( |G| = |\Sigma| + |\mathcal{X}| \).

**Chomsky Normal Form.**
A context-free grammar \( G = \langle \mathcal{X}, \Sigma, R, S \rangle \) is in Chomsky Normal Form (CNF for short) if any production rule is in one of the following forms:

(i) \( X \to YZ \) where \( X \in \mathcal{X} \) and \( Y, Z \in \mathcal{X} \setminus \{S\} \).

(ii) \( X \to a \) where \( X \in \mathcal{X} \) and \( a \in \Sigma \), or

(iii) \( S \to \varepsilon \).

Let \( G = \langle \mathcal{X}, \Sigma, R, S \rangle \) be a grammar in Chomsky normal form and \( \alpha = u_{0} \Rightarrow_{G} u_{1} \Rightarrow_{G} \ldots u_{n} \) be a derivation sequence of \( G \). If the production rule \( S \to \varepsilon \) is used in \( \alpha \), then it has to be the only rule used in the whole derivation sequence. In this case, the derivation sequence simply becomes \( \alpha = u_{0} \Rightarrow_{G} u_{n} \) with \( u_{0} = S, n = 1 \) and \( u_{1} = \varepsilon \). Otherwise, the production rule \( S \to \varepsilon \) is not used and only the two first types of productions rules \( (X \to YZ \) and \( X \to a) \) can be used in the derivation sequence. Since these two rules replace one symbol with either one or two symbols that are not \( \varepsilon \), we have that \( |u_{1}| \leq |u_{2}| \leq \ldots \leq |u_{n}| \).

**Proof of lemma 6.A.1.** Let \( P \) be a PDA and \( k \in \mathbb{N} \). By keeping track of the number of input letters read so far, we can construct in polynomial time in the size of \( P \), a PDA \( P_{\leq k} \) such that \( L(P_{\leq k}) = L(P) \cap \Sigma^{\leq k} \). Moreover, it is possible to build in polynomial time in the size of \( P_{\leq k} \) a CFG \( G \) such that \( L(G_{\leq k}) = L(P_{\leq k}) \) [59]. Furthermore, it has
been shown in [67] that we can construct, in polynomial time in the size of $G_{\leq k}$, a CFG $G_{\text{CNF}}$ in Chomsky normal form such that $L(G_{\text{CNF}}) = L(G_{\leq k})$. Assuming that $G_{\text{CNF}} = \langle \chi_{G_{\text{CNF}}}, \Sigma, R_{G_{\text{CNF}}}, S \rangle$, we can build in polynomial time in the size of $G_{\text{CNF}}$ a pushdown automaton $P_{\text{CNF}} = \langle \{q_0, q\} \cup \{q^1, q^2 \mid r \in R_{G_{\text{CNF}}} \}, \Sigma, \chi_{G_{\text{CNF}}}, \delta_{P_{\text{CNF}}}, \{q_0\}, \{q\} \rangle$ such that $L(P_{\text{CNF}}) = L(G_{\text{CNF}})$. The transition relation $\delta_{P_{\text{CNF}}}$ of $P_{\text{CNF}}$ is built as follows:

(i) From the initial state $q_0$, $P_{\text{CNF}}$ starts by pushing to the stack the non-terminal start symbol $S$ using the transition $(q_0, \varepsilon) \xrightarrow{\varepsilon} (q, S)$.

(ii) For every production rule $r \in R_{G_{\text{CNF}}}$ of the form $A \rightarrow B, C$ where $A \in \chi_{G_{\text{CNF}}}$ and $B, C \in \chi_{G_{\text{CNF}}} \setminus \{S\}$, we associate in $P_{\text{CNF}}$ two states $q^1_r$ and $q^2_r$ and the following three transitions:

(a) A pop of the non-terminal symbol $A$ from the stack $(q, A) \xrightarrow{\varepsilon} (q^1_r, \varepsilon)$,

(b) A push of the non-terminal $C$ to the stack $(q^1_r, \varepsilon) \xrightarrow{\varepsilon} (q^2_r, C)$, followed by

(c) A push of the non-terminal symbol $B$ to the stack $(q^2_r, \varepsilon) \xrightarrow{\varepsilon} (q, B)$.

(iii) For every production rule $r$ of the form $A \rightarrow a$, we associate in $P_{\text{CNF}}$ the pop transition $(q, A) \xrightarrow{a} (q, \varepsilon)$.

(iv) If the production rule $S \rightarrow \varepsilon$ is in $R_{G_{\text{CNF}}}$, then we add to $\delta_{P_{\text{CNF}}}$ the pop transition $(q, S) \xrightarrow{\varepsilon} (q, \varepsilon)$.

Observe that $P_{\text{CNF}}$ simulates every possible derivation sequence of the grammar $G_{\text{CNF}}$ in a leftmost manner.

Let $w \in L(P_{\text{CNF}})$ and $c_0, c_1, \ldots, c_n$ be sequence of configurations of the run of $P_{\text{CNF}}$ that produces $w$, where $c_0 = \langle q_0, \varepsilon \rangle$ and $c_n = \langle q, \varepsilon \rangle$. This run simulates a derivation sequence of the grammar $G_{\text{CNF}}$ of the form $\alpha = u_1 \Rightarrow G_{\text{CNF}} \ u_2 \Rightarrow G_{\text{CNF}} \ \cdots \Rightarrow G_{\text{CNF}} \ u_m$ where $u_1 = S$, $u_m = w$ and $u_i \in (\Sigma \cup R_{G_{\text{CNF}}})^*$. Since the stack of $P_{\text{CNF}}$ contains, at every moment of the derivation, the remaining non-terminal symbols of some derivation word $u_i$, $i : 1 \leq i \leq m$, we can assert that for every configuration $c_j$, $j : 1 \leq j \leq n$, there is a derivation word $u_l$ with $l : 1 \leq l \leq m$ such that $|\text{Stack}(c_j)| \leq |u_j|$. Moreover, Since $G_{\text{CNF}}$ is in CNF, there could be only two possible cases. In the first case, the derivation tree is of the form $\alpha = S \Rightarrow G_{\text{CNF}} \ \varepsilon$ and $w = \varepsilon$. In this case, the stack of $P_{\text{CNF}}$ contains at most the start symbol. In the second case, the derivation tree is such that $|u_1| \leq |u_2| \leq \ldots |u_m| = |w|$. Since $w \in L(P_{\text{CNF}}) = L(G_{\text{CNF}}) = L(P_{\leq k})$, we have that $|w| \leq k$. We conclude that the stack of every run of the PDA $P_{\text{CNF}}$ is either bounded by $k$ or by 1, i.e. bounded by $\max(k, 1) \leq k + 1$. As a corollary, we have that $L_{k+1}(P_{\text{CNF}}) = L(P_{\text{CNF}})$. Thus, from the PDA $P$ and $k \in \mathbb{N}$, we could build in polynomial time in the size of $P$ a PDA $P' := P_{\text{CNF}}$ such that $L_{k+1}(P') = L(P) \cap \Sigma^k$.

Now we can prove lemma 6.1.1 for the case of finite words. The proof can be extended in a straightforward manner for the case of infinite words.

**Proof of lemma 6.1.1.** Let $P$ be a PDA and $k, d \in \mathbb{N}$. To prove lemma 6.1.1, we need to build, in polynomial time in the size of $P$, a PDA $P'$ such that
We obtain the upper bound by reducing the problem to the **PSAPCE-complete emptiness problem for a Turing machine** \( M_{\text{lst}} \) operating on \((n+1)\) tapes \( T_i \), \( i : 0 \leq i \leq n \), where each tape has \((d_i + 3)\) cells.

To demonstrate this reduction, we first show how to encode a \( d \)-bounded-depth PDA \( P \) into a Turing machine \( M(P, d) \). In particular, we show how to encode the bounded-depth stack into a tape and how to encode pushdown transitions into Turing machine transitions. Then, given a pushdown automaton \( P \) and a letter \( a \) from its alphabet, we show how to derive a pushdown automaton \( [P]_a \) that mimics \( P \) by restricting its transitions to the ones where either \( \varepsilon \) or the particular letter \( a \) is read. Finally, we show how the Turing machine \( M_{\text{lst}} \) is built and explain the reduction.

**Encoding of the stack.** Let \( P = \langle Q, \Sigma, \Gamma, \delta, I, F \rangle \) be a PDA and \( d \) its bounded-depth. We encode the \( d \)-bounded depth stack of \( P \) into the tape of the Turing machine \( M(P, d) \) as follows. The tape has the following set of symbols \( \Gamma \cup Q \cup \{\#, \bot\} \). The first and the last elements of the tape are initialized to the symbol \( \# \) to delimit the beginning and the end of the tape. The second element of every tape \( T_i \), next to the first delimiter, is initialized to one of the initial states \( q_0 \in I \) of \( P \). The remaining \( d \) elements in the tape are used to encode the content of the bounded depth stack of \( P \). At the beginning of the simulation, these \( d \) elements are initialized to the symbol \( \bot \), denoting the empty part of the tape.

**Encoding of pushdown transitions.** The head of the tape of the Turing machine \( M(P, d) \) should always point to an element in the tape that contains the
current state of $P$. If the transition performed by $P$ does not involve the stack and is of the form $(q_1, \varepsilon) \xrightarrow{a} (q_2, \varepsilon)$, then the Turing machine performs the following actions: (i) Read $q_1$ from the tape, (ii) Write $q_2$. If the transition performed by $P$ is a push transition of the form $(q_1, \varepsilon) \xrightarrow{a} (q_2, b)$, then the Turing machine performs the following successive actions: (i) Read $q_1$ from the tape, (ii) Write $b$ on the tape, (iii) Move towards the empty part of the tape, (iv) Read $\bot$, (v) Write $q_2$. If the transition performed by $P$ is a pop transition of the form $(q_1, b) \xrightarrow{a} (q_2, \varepsilon)$, then the Turing machine performs the following successive transitions: (i) Read $q_1$ from tape $T$, (ii) Write $\bot$, (iii) Move towards the beginning of the tape, (iv) Read $b$, (v) Write $q_2$.

**Pushdown** $[P]_a$. Let $P = \langle Q, \Sigma, \Gamma, \delta, I, F \rangle$ be a PDA and $a \in \Sigma$. The pushdown automaton $[P]_a$ is a copy of $P$ such that: (i) For each state $q$ from the set of states $Q$ of $P$, $[P]_a$ contains the pair of states $[q]_a$ and $[q]^a_a$, (ii) Only transitions reading $\varepsilon$ and $a$ are kept, (iii) If $[P]_a$ is in a state of the form $[q]_a$, then the word read by $[P]_a$ so far is the empty word $\varepsilon$, (iv) If $[P]_a$ is in a state of the form $[q]^a_a$, then the word read by $[P]_a$ so far is the letter $a$. Formally, $[P]_a$ is defined by the tuple $\langle Q_a, \Sigma, \Gamma, \delta_a, I_a, F_a \rangle$ where:

(i) The set of states is $Q_a = \{ [q]_a, [q]^a_a \mid q \in Q, a \in \Sigma \}$.

(ii) The transition relation is:

$$\delta_a = \left\{ ([q]_a, u) \xrightarrow{a} ([q']_a, u') \mid (q, u) \xrightarrow{a} (q', u') \in \delta \right\}$$

$$\cup \left\{ ([q]_a, u) \xrightarrow{e} ([q']_a, u') \mid (q, u) \xrightarrow{a} (q', u') \in \delta, a \in \Sigma \right\}$$

(iii) The set of initial states $I_a = \{ [q]_a \mid q \in I \}$.

(iv) The set of final states $F_a = \{ [q]_a, [q]^a_a \mid q \in F \}$.

**The reduction.** Let $M([P]_a, d_i)$ be the Turing machine that simulates the $d_i$ depth-bounded pushdown automaton $[P]_a$ for $i : 0 \leq i \leq n$ and $a \in \Sigma$. We use $Q_{M([P]_a, d_i)}$ to denote the set of states of the machine $M([P]_a, d_i)$. The Turing machine $M_{\text{lst}}$ is set to have the following set of states

$$\{ q_0, q_1 \} \cup \{ q_a \mid a \in \Sigma \} \cup \left( \bigcup_{a \in \Sigma} \left( \bigcup_{i:1 \leq i \leq n} Q_{M([P]_a, d_i)} \right) \right)$$

The initial state of the Turing machine is $q_0$. The Turing machines has $(n + 1)$ tapes, denoted by $T_i$. Each tape $T_i$ is used to simulate the stack of the PDA $P_i$. The initialization phase of the Turing machine $M_{\text{lst}}$ consists in initializing the tape that encodes the stack of every pushdown automaton $P_i$. After initializing each tape, the machine moves to state $q_1$, where the simulation starts. The machine first guesses a letter $a$. If the letter can be read by both $P_0$ and one of the PDAs $P_i$, $i : 1 \leq i \leq n$, then the simulation succeeds. Otherwise, the simulation fails and the machine $M_{\text{lst}}$ blocks. When guessing letter $a$, the machine reads the current state $q_{\text{cur}}$ of $P_0$ from tape $T_0$ and moves to state $[q_{\text{cur}}]_a$. It simulates $P_0$ till a letter $a$ has been read. Letter $a$ is read when the machine reaches a state of the form $[q_{\text{new}}]_a^a$ with $q_{\text{new}} \in Q_i$. At this point, the machine writes the new state $q_{\text{new}}$ of $P_0$ in $T_0$ and moves to state $q_a$. In state
$q_a$, the machine makes a non-deterministic choice among the set of shuffled pushdown automata $P_i$, $i: 1 \leq i \leq n$. If $P_j$ is chosen for some $j: 1 \leq j \leq n$, then the machine reads from the tape $T_j$ the current state $q_{\text{cur}}$ of $P_j$ and moves to state $[q_{\text{cur}}]_a$. The machine simulates $P_j$ till letter $a$ is read. This occurs when the machine reaches a state of the form $[q_{\text{new}}]_a$ with $q_{\text{new}} \in Q_j$. If this happens, the machine saves the new state $q_{\text{new}}$ of $P_j$ in the tape $T_j$ and moves back to state $q_1$ for another simulation. At state $q_1$, the Turing machine $M_{\text{fst}}$ can non deterministically check whether every PDA $P_i$ is in one of its final states. If so, $M_{\text{fst}}$ halts. By construction, $M_{\text{fst}}$ halts if and only if $L_{d_0}(P_0) \cap \uplus(L_{d_1}(P_1), L_{d_2}(P_2), \ldots, L_{d_n}(P_n))$ is not empty.

**Lower bound.**

We show that the non-emptiness test of the synchronization of depth bounded PDA is PSAPCE-hard. We get this result through a reduction from the non-emptiness of the intersection of $n$ regular languages [62].

Let $A_1, A_2, \ldots, A_n$ be $n$ finite-state automata operating on the same alphabet $\Sigma$, and let $L(A_1), L(A_2), \ldots, L(A_n)$ be their respective regular languages. The non-emptiness problem of the intersection between the $n$ regular languages $L(A_1), L(A_2), \ldots, L(A_n)$ asks whether there exists a word $w \in \Sigma^*$ such that $w \in L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)$. In the following, we show that this problem reduces to the non-emptiness test of the synchronization of depth bounded $n+1$ pushdown automata $P_0, P_1, \ldots, P_n$ that we define.

The idea of the reduction consists in marking the alphabet of every automaton $A_i$ with the number $i$, $i: 1 \leq i \leq n$. By doing so, we will be able to tell which letter comes from which automaton in a word from the language $L(A_1) \uplus L(A_2) \uplus \ldots \uplus L(A_n)$. Then we define the automaton $A_0$ which accepts any word of marked letters, such that each letter in the word appears successively $n$ times marked with 1, 2, \ldots, $n$. The intersection between the languages $L(A_0)$ and $L(A_1) \uplus L(A_2) \uplus \ldots \uplus L(A_n)$ will tell us about the emptiness of $L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)$. The last piece consists in observing that a finite-state automaton can be seen as a pushdown automaton which stack depth is zero.

**Marking the letters.** For $i: 1 \leq i \leq n$, we define the marked alphabet $\Sigma_i := \{(a,i) \mid a \in \Sigma\}$. We use $\Sigma_{1,\ldots,n} := \uplus_{1 \leq i \leq n} \Sigma_i$ to denote the union of these alphabets.

Let $A = \langle Q, \Sigma, \delta, q_{\text{init}}, Q_{\text{final}} \rangle$ be a finite-state automaton. We define the $i^{th}$ marked finite-state automaton $\langle A, i \rangle$ as the copy of automaton $A$, in which each transition reads the marked letter $(a,i)$ instead of $a$. Formally, automaton $\langle A, i \rangle$ is defined as follows: $\langle A, i \rangle := \langle Q, \Sigma_i, \langle \delta, i \rangle, q_{\text{init}}, Q_{\text{final}} \rangle$ with $\langle \delta, i \rangle := \{\langle q_1, (a,i), q_2 \rangle \mid \langle q_1, a, q_2 \rangle \in \delta\}$.

Let $w \in \Sigma$. We use $\text{mark} : \Sigma^* \to \Sigma_{1,\ldots,n}$ to denote the word mapping function defined as follows: (i) $\text{mark}(\varepsilon) = (\varepsilon, 1)(\varepsilon, 2)\ldots(\varepsilon, n)$, and (ii) $\text{mark}(a,w) = \text{mark}(a).\text{mark}(w)$ for every $a \in \Sigma$ and $w \in \Sigma^*$. We define the language $L_{\text{mark}} :=$
\{\text{mark}(w) \mid w \in \Sigma^*\} \subseteq \Sigma_1, \ldots, \Sigma_n^*$. Let $A_0$ be the finite-state automaton that accepts the language $L_{\text{mark}}$.

The reduction. Assume that there is a letter $a \in \Sigma$ such that $a \in L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)$. This is equivalent to say that there exists a letter $a \in \Sigma$ such that the word $\text{mark}(a) = (a,1)(a,2)\ldots(a,n)$ is in the language $L(\langle A_1,1 \rangle) \sqcup L(\langle A_2,2 \rangle) \sqcup \ldots \sqcup L(\langle A_n,n \rangle)$. We extend this idea as follows. Assume now that

$$L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n) \neq \emptyset$$

That is, assume that there is a word $w \in L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n)$. This is equivalent to say that there exists a word $w \in \Sigma^*$ such that

$$\text{mark}(w) \in L(\langle A_1,1 \rangle) \sqcup L(\langle A_2,2 \rangle) \sqcup \ldots \sqcup L(\langle A_n,n \rangle)$$

Which is equivalent to say that:

$$L(A_0) \cap (L(\langle A_1,1 \rangle) \sqcup L(\langle A_2,2 \rangle) \sqcup \ldots \sqcup L(\langle A_n,n \rangle)) \neq \emptyset$$

From finite-state to pushdown automaton. Let $A = \langle Q, \Sigma, \delta, q_{\text{init}}, Q_{\text{final}} \rangle$ be a finite-state automaton. We associate to $A$ the pushdown automaton $P(A) := \langle Q, \Sigma, \emptyset, \delta', I, F \rangle$. where the transition relation $\delta'$ reproduces the transitions $\delta$ from automaton $A$, i.e. $\delta' := \{(q_1, \varepsilon) \xrightarrow{a} (q_2, \varepsilon) \mid (q_1, a, q_2) \in \delta\}$. Observe that none of the transitions in $\delta'$ is a push or a pop. Moreover, the languages accepted by $A$ and $P(A)$ are identical. Therefore, the PDA $P(A)$ is 0-depth bounded and we have that $L_0(P(A)) = L(P(A)) = L(A)$.

We define $P_0 := P(A_0)$ and, for every $i : 1 \leq i \leq n$, we define $P_i := P(\langle A_i, i \rangle)$. The last statement from the previous paragraph can now be reformulated as follows:

$$L(A_1) \cap L(A_2) \cap \ldots \cap L(A_n) \neq \emptyset$$

is equivalent to

$$L(A_0) \cap (L(\langle A_1,1 \rangle) \sqcup L(\langle A_2,2 \rangle) \sqcup \ldots \sqcup L(\langle A_n,n \rangle)) \neq \emptyset$$

which is now equivalent to

$$L_0(P_0) \cap (L_0(P_1) \sqcup L_0(P_2) \sqcup \ldots \sqcup L_0(P_n)) \neq \emptyset$$

\qed
7. Conclusion and Future Work

In this thesis, we have extended the applicability of Model-Checking to three classes of infinite-state systems that model networks of communicating processes.

First, we have considered the coverability problem for the class of directed acyclic Ad-Hoc Networks (AHN). A directed acyclic AHN is a network composed of an unbounded number of finite-state processes that communicate by broadcasting messages and where the topology is a static directed acyclic graph. We have shown that the coverability problem for this class, parametrized by the number of processes in the network, is undecidable. Therefore, we have considered the class of bounded-depth directed acyclic AHN, in which the graph topology of the network has a depth bounded by some predefined value. This class of AHN models the directed and acyclic flow of information in a Wireless Sensor Network. We have proved that the state reachability problem becomes decidable for bounded-depth directed acyclic AHN through an instantiation of the Well-Structured Transition System framework. As future work, it would be interesting to study the impact of richer broadcast mechanisms such as those that allow processes to have local (unbounded) mailboxes, and to consider models augmented by timed and probabilistic transitions in order to allow quantitative reasoning about network behaviors.

Second, we have considered networks of Dynamic Register Automata (DRA). We have shown that the reachability problem for this class of networks is undecidable in the general case. We have considered two variants of this model, a synchronous variant in which messages are exchanged via rendezvous communication, and a buffered variant in which processes are equipped with a queue which allows to save received messages for later reading. In the case of synchronous DRA, we have defined the class of strongly bounded degenerative DRA, in which the graph topology of the network is strongly bounded and where every process can reset any of its registers at any moment in time. In the case of buffered DRA, we have defined the class of bounded-buffer and strongly-bounded lossy DRA, in which the graph topology of the network is strongly bounded, the capacity of the buffer of every process is bounded and every process in the network can non-deterministically disconnect itself. We show that the state reachability problem becomes decidable for each one of these classes. Among possible directions for future work is to see whether the decidability can be obtained for more general classes by considering other channel semantics, such as the
unordered and the lossy ones. We also think that an interesting line of work would be to study the link between dynamic register automata and the \( \pi \)-calculus and to investigate the relation between our results using the DRA formalism and the work of Meyer [73] on the \( \pi \)-calculus with the bounded depth restriction. In fact, in both the DRA and the \( \pi \)-calculus, processes have in common the capability of creating processes and sending and receiving process IDs, allowing the topology of the network to evolve.

Finally, we have considered the problems of state reachability and LTL Model-Checking for concurrent programs modeled as Multi-Pushdown Systems (MPDS). We have analyzed MPDS under the bounded budget restriction, which allows threads to perform an unbounded number of context switches when their stack is below some given depth \( d \) and limits the number of contexts they can perform to some given value \( k \) when their stack goes above \( d \). We have shown that the state reachability problem for bounded budget multipushdown systems is PSAPCE-complete and that the LTL Model-Checking problem for the same class is EXPTIME-complete. We have also proposed a code-to-code translation that, given a budget \((k, d)\), transforms a concurrent program into a sequential program that simulates every behavior of the concurrent program under the \((k, d)\) budget restriction. We have demonstrated the utility of this sequentialization scheme by using sequential program verification tools to verify the safety and the termination of concurrent programs. An interesting line of work in the future would be, first, to consider the CTL Model-Checking of bounded-budget and, second, to analyze termination of bounded-budget MPDS under probabilistic scheduling.
References


Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1605

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-334788

ACTA UNIVERSITATIS UPSALIENSIS
UPPSALA
2017