THÈSE PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Corto Mascle**

POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

SPÉCIALITÉ INFORMATIQUE

Vérification et synthèse de systèmes distribués à synchronisation faible

Verification and synthesis of distributed systems with weak synchronisation

Sous la direction d'Anca Muscholl et Igor Walukiewicz

Soutenue le 28 novembre 2024

Composition du jury :

Anca Muscholl	Professeure des universités, Université de Bordeaux	Directrice de thèse
Igor Walukiewicz	Directeur de Recherche, Université de Bordeaux	Co-directeur de thèse
Parosh Aziz Abdulla	Professor, Uppsala University	Rapporteur
Rupak Majumdar	Scientific Director, MPI Kaiserslautern	Rapporteur
Nathalie Bertrand	Directrice de Recherche, INRIA Rennes	Examinatrice
Ahmed Bouajjani	Professeur, Université Paris-Diderot	Examinateur
Jérôme Leroux	Directeur de recherche, Université de Bordeaux	Examinateur

À mon grand-père, André Delarc

Acknowledgements

Find a group of people who challenge and inspire you; spend a lot of time with them, and it will change your life.

Amy Poehler

First of all I am very grateful to my two supervisors, Anca and Igor, who gave me time, support, solid advice, enlightening conversations, infinite patience, and excellent book recommendations.

I am also grateful to Parosh Aziz Abdulla and Rupak Majumdar for taking the time to review my thesis. Many thanks to the other members of my jury, Nathalie Bertrand, Ahmed Bouajjani and Jérome Leroux, for accepting to take part in it.

I had the pleasure to work with many wonderful people over the past three years, all with their own styles and personalities. A warm thank you to Aditya, Antonio, Aliaume, Chana (special thanks to you two for the wonderful Autobóz in Estonia), Denis, Gabriel, Hugo (thank you for consistently believing in population games), Nathan, Nathanaël (special thanks as you helped me tremendously to integrate the world of research), Lucie, Nicolas(this thesis would have been very different without you two), Olivier, Patrick, Roland, Sandra, Steffen. You made research incredibly fun!

The list above is without a doubt incomplete. My gratitude goes also to all the many, many people who took the time, throughout these years, to explain their work, discuss problems or share their experience with me.

Impossible de faire des remerciements sans y inclure l'AFoDIB, le cœur battant de la vie du labo. Merci à Aline, Jonathan et Quentin, qui tenaient l'association à mon arrivée, et qui ont fait de ce lieu de travail un endroit convivial pour les nouveaux arrivants. Merci Sarah (sans oublier Azur, célébrité locale), j'ai adoré les nombreuses heures passées à discuter autour d'un jeu, d'un café, d'un délicieux plat, ou sur un quai danois. Merci Rémi, par-delà tes excellents conseils en cuisine et en livres, et diverses facéties, tu sais donner à toutes les occasions un *je ne sais quoi* qui les rends terriblement plus funs. Merci Thibault (avec qui nous nous connaissons depuis petits) pour les délicieux cookies, les décorations à base de petits chats, et pour avoir distillé ta bonne humeur à travers de nombreuses années de cohabitation. Merci Théo, je te souhaite de continuer à grimper jusqu'aux plus hauts sommets. Merci Émile, aventurier infatigable et source intarrissable d'informations sur les champignons, j'ai hâte au prochain concert ou à la prochaine soirée diapos; Edgar, *jack of all trades* et ses talents de magie (ou sorcellerie, je ne suis pas

sûr); Clara, avec qui il est impossible de s'ennuyer; Priscilla, avec qui j'apprécie toujours les balades en fin de journée; Romain, d'une jovialité communicative, grâce à qui j'ai découvert un tas de films chouettes; Roman, puits de connaissances, qui sait rendre n'importe quelle conversation profonde et intéressante; Timothée, amateur de bon mots; Timothé, fin connaisseur de crêpes et pains pitas; Gabriel, grâce à qui j'ai appris mille et une astuces; Aïda, dont les rares visites ne les ont rendues que plus précieuses.

Merci aux nombreuses personnes qui ont animé les soirées pizzas, les GT son et lumière, les repas-campagnard dans l'atrium, les balades en Forêt et les nombreux autres moments : Claire, Joséphine, Vincent M, Colleen, Amaury, Guilhem, Pierre B, Théo D, Augusta, Clément, Zoé, Taïssir, ... j'en oublie sans doute, mais iels se reconnaitront.

Merci également aux autres membres de l'équipe: Géraud, qui partage généreusement ses connaissances et sa bonne humeur, Vincent, d'une bonhommie et bienveillance constante (ça doit être le thé), Gérald pour son précieux engagement dans l'aide aux doctorant.e.s en difficulté, et celles et ceux qui font vivre cette équipe, Diego, Laurent, Marc, ...

Je remercie aussi les amis qui ont égayé mon temps libre pendant ces années, à commencer par Axel et Brendan évidemment, avec qui j'ai hâte de vivre de nouvelles aventures. Merci à Valentin, qui m'a fait découvrir les festivals d'Angoulême qui sont devenus des moments incontournables de mon année, à Manon, Bastien, Yoan, qui m'ont accueilli plusieurs fois, je reviendrai vous voir, promis, et merci aux autres amis de l'ENS, Lucas, Marius,... Aux Cognitivistes, notamment Mathias et Pierre, sans qui mon déménagement aurait été beaucoup plus compliqué. Aux Potirons, menés par Pierre, avec qui nous avons inlassablement écumé les quizzs des pubs de Bordeaux (je révise attentivement mon Nakamura lock, et les playlists de M. Fenouil). À Victor, de qui j'ai toujours apprécié les visites à Bordeaux, merci d'avoir gardé contact.

Antonio, me mola la gramola pasar tiempo contigo. In and outside the lab, you are always supporting me and pushing me forward, with humour and wisdom. Throughout our adventures, (be it on the walls of Saint-Malo, the shores of northern Estonia, the heights of Marseille, the ice cream shops of Valladolid or the middle of the German countryside), I made unforgettable memories. You are a formidable friend, quien gestionas el helecho.

Emma, tu as toujours été là, dans le calme comme dans la tempête. Je suis si heureux de t'avoir rencontré. Toutes ces maquettes, saisons de Charmed, sorties dans les parcs ou dans les rues d'Angoulême seraient sans saveur si tu n'étais pas là. Merci aussi à ta famille, Laurence, Ghislain et Maëlle, de m'avoir accueilli et aidé tant de fois, avec une générosité sans faille.

Enfin, je ne serais jamais arrivé jusqu'ici sans le soutien sans faille de ma mère. Merci de m'avoir offert tant d'amour et de soutien à travers toutes ces années. Cette thèse t'es aussi dédiée.

Merci à toutes et à tous. Trois ans sont une grande part de vie, et je suis heureux de les avoir passés avec vous.

Vérification et synthèse de systèmes distribués à synchronisation faible

Résumé Les programmes distribués sont une source de difficulté pour les méthodes formelles. Les interactions de plusieurs composantes provoquent une explosion combinatoire qui complexifie la vérification. Dans le cadre de la synthèse de contrôleurs, à cela s'ajoute le fait que les stratégies d'une composante ne peuvent tenir compte que d'une vision partielle du système. Ceci a mené à de nombreux résultats d'indécidabilité dans le domaine et des complexités très élevées dans les cas décidables. Dans cette thèse, nous proposons une approche permettant de décomposer les problèmes de vérification et de synthèse en instances locales aux processus. Cette approche mène à des résultats de décidabilité, et même dans certains cas à des algorithmes de complexités suffisamment basses pour envisager des applications. Nous montrons que cette méthode s'applique dans des systèmes distribués avec une communication restreinte entre les composantes: d'une part des systèmes où la communication se fait via la prise et le relâchement de locks, et d'autre part des systèmes où les processus communiquent par broadcast. De plus, nous utilisons cette approche dans des modèles a priori très différents, où le nombre de processus est constant, évolue dynamiquement, ou encore est paramétré.

Verification and synthesis of distributed systems with weak synchronisation

Abstract Distributed programs are a cause of difficulty for formal methods. The interactions of multiple components lead to a combinatorial explosion, which complicates verification. In the context of controller synthesis, this is further compounded by the fact that the strategies of one component can only take into account a partial view of the system. This has led to numerous undecidability results in the field and very high complexities in decidable cases. In this thesis, we propose an approach that allows the decomposition of verification and synthesis problems into local instances for individual processes. This approach leads to decidability results, and in some cases, to algorithms with low enough complexity to consider practical applications. We show that this method applies to distributed systems with restricted communication between components: on the one hand, systems where communication is limited to the acquisition and release of locks, and on the other hand, systems where processes communicate via broadcast. Moreover, we apply this approach to models that are quite different, where the number of processes is constant, evolves dynamically, or is parameterised.

Contents

Résumé étendu en français

Cette thèse se place dans le cadre de l'étude des systèmes distribués, c'est-à-dire des systèmes informatiques constitués de plusieurs composantes travaillant en parallèle. Le problème principal étudié est la *synthèse de contrôleurs*: on considère un système où chaque composante peut faire des choix sur ses actios locales, et on cherche à construire des stratégies qui font ces choix de manière à garantir le bon fonctionnment du système.

Illustrons cette idée avec l'énigme suivante [**PrisonersLightbulb15**; **Deslandes2023**; **Winkler2024**]:

Les prisonniers et l'ampoule

100 prisonniers doivent jouer au jeu suivant : Dans la prison, il y a une pièce vide à l'exception d'un interrupteur. Chaque jour, un prisonnier est choisi au hasard et amené dans la pièce. Il peut voir si la lumière est allumée ou éteinte, et l'allumer ou l'éteindre s'il le souhaite. Il est ensuite renvoyé dans sa cellule. À tout moment, un prisonnier peut décider de mettre fin au jeu en affirmant que tous les prisonniers sont entrés dans la pièce au moins une fois. S'il a raison, tous les prisonniers sont libérés, sinon aucun d'entre eux ne l'est.
Les prisonniers ne peuvent pas communiquer pendant le jeu, mais peuvent se mettre d'accord sur une stratégie à l'avance. Comment peuvent-ils garantir d'être finalement libérés avec une probabilité de 1 ? [a]

[a]Les solutions peuvent varier selon que l'on autorise ou non les prisonniers à compter les jours écoulés depuis le début du jeu. Dans cette thèse, nous considérons des systèmes asynchrones, nous chercherons donc une solution où les prisonniers ne comptent pas le temps.

Cette énigme illustre parfaitement le genre de problème que nous cherchons à résoudre. Ici les prisonniers n'ont qu'une vue très limitée du reste du système. Essentiellement, la seule information qui leur parvient est le fait que l'ampoule soit allumée ou éteinte quand ils la voient. Le fait que cette énigme a une solution indique que même une communication très limitée entre les composantes peut avoir des effets non triviaux.

Motivations et histoire de la synthèse distribuée

Le problème de la synthèse trouve son origine dans l'article fondateur d'Alonzo Church publié en 1957 [**Church57**]. Il pose le problème de la construction d'un circuit qui, lorsqu'il reçoit certaines entrées, produit des sorties satisfaisant une spécification donnée. Quelques années plus tard, il a introduit une version du problème pour les systèmes réactifs, dans lesquels le système conçu doit interagir avec un environnement. Ce problème a été traduit dans un formalisme de jeux et résolu dans ce cadre par Büchi et Landweber [**BuchiL69a**]. Depuis lors, les études sur la synthèse réactive et les jeux se sont multipliées. Cependant, la plupart de ces travaux ne prennent en compte que les systèmes et les spécifications séquentiels. Dans cette thèse, nous nous intéressons aux systèmes *distribués*.

Les systèmes distribués sont constitués de plusieurs composants distincts travaillant dans un but commun et dotés de mécanismes de communication. Ils offrent de grands avantages en termes d'efficacité, car ils permettent de répartir les tâches entre plusieurs composants. Ces gains sont encore meilleurs lorsque les systèmes peuvent effectuer leurs tâches sans se synchroniser souvent. Toutefois, cet avantage s'accompagne d'un inconvénient majeur : ils sont notoirement sujets aux erreurs, car il est facile de ne pas remarquer un mauvais entrelacement d'actions lors de leur construction. C'est ce qui a motivé l'utilisation de méthodes formelles pour vérifier et synthétiser les systèmes distribués dans les années 80. Le problème de la synthèse distribuée s'est depuis avéré très difficile. Notamment, Pnueli et Rosner [**PnuRos89**] ont démontré l'indécidabilité de la synthèse distribuée pour les systèmes à états finis qui communiquent de manière synchrone par messages en respectant les spécifications LTL, même avec seulement trois processus. Des recherches ultérieures ont montré que, pour l'essentiel, les seules architectures décidables sont les pipelines, où chaque processus ne peut envoyer des messages qu'au processus suivant dans le pipeline. messages qu'au processus suivant dans le pipeline [**ScheweF06**]. En outre, la complexité n'est pas élémentaire dans la taille du pipeline. Ces résultats négatifs ont motivé l'étude de la synthèse distribuée pour le modèle des *automates asynchrones* [**Zielonka87**], d'abord avec des stratégies locales [**MadhusudanT01**; **Sznajder2009**] puis avec de la *mémoire causale* [**GastinLZ04**; **MadhusudanTY05**]. Dans ce dernier cadre, les processus communiquent en se synchronisant sur les actions. Lorsqu'ils le font, ils ont accès aux informations des autres sur ce qui s'est passé dans l'exécution jusqu'à présent. Il a été démontré que le problème de la synthèse distribuée est décidable dans certains sous-cas très restreints [**GastinLZ04**; **GenestGMW13**; **MuschollW14**], mais même dans ces cas la complexité est non élémentaire. Pire encore, il a été récemment établi que la synthèse distribuée avec des informations causales est indécidable pour les architectures non contraintes [**Gimbert22**].

La synthèse distribuée pour les réseaux de Petri [**FinkbeinerO17ic**] a connu des progrès limités similaires et, en raison de [**Gimbert22**], est également indécidable dans le cas général, puisque la synthèse distribuée pour les automates asynchrones peut y être réduite [**BeutnerFH19**]. Tout ceci démontre la difficulté inhérente au problème.

Dans cette thèse, nous présentons une nouvelle approche de la synthèse distribuée, avec des stratégies locales. Nous démontrons son efficacité en l'appliquant à trois modèles différents dont le point commun est que la communication entre les processus est très restreinte. Ces modèles sont décrits à la fin de ce chapitre.

Approche de cette thèse

Nous commençons par discuter de notre définition de la synthèse distribuée. Dans cette thèse, nous nous concentrons spécifiquement sur la synthèse de *contrôleurs* : nous considérons des systèmes préexistants qui incluent à la fois des actions contrôlables et incontrôlables, et nous visons à développer des stratégies pour sélectionner des actions contrôlables d'une manière qui assure qu'une spécification donnée est satisfaite.

En outre, nous supposons que les stratégies sont entièrement locales, c'est-à-dire qu'elles reposent uniquement sur la séquence d'opérations effectuées par le processus lui-même, sans accès à l'exécution plus large de l'ensemble du système. Cette hypothèse est justifiée par le fait que nos modèles sont des abstractions de systèmes réels, où l'ensemble des exécutions dans le modèle sur-approxime celui du système réel.

Bien que cette approximation soit en principe saine (une mauvaise exécution dans le système implique une mauvaise exécution dans le modèle), elle n'est pas complète : l'absence d'erreurs dans le système n'implique pas l'absence d'erreurs dans le modèle. Il est donc raisonnable de sous-approximer les informations dont disposent les contrôleurs sur le système. Si nous pouvons trouver un contrôleur qui fonctionne efficacement avec des informations limitées dans le modèle, il devrait également fonctionner dans le système réel.

Nous abordons les défis de la synthèse distribuée avec une nouvelle approche qui passe par l'étude de modèles distribués avec une communication restreinte, où la synthèse globale peut être décomposée en sous-problèmes locaux. Dans ce contexte, le terme « local » fait référence au comportement des processus individuels, tandis que le terme « global » se rapporte au comportement commun de tous les processus. Nous présentons ici une vue d'ensemble de l'approche.

Description générale de notre méthode

Notre approche peut être résumée par le schéma suivant :

1. Tout d'abord, nous définissons une classe de familles d'exécutions locales, qui ont une description de taille bornée. Nous identifions les « bonnes » familles d'exécutions comme étant celles pour lesquelles toutes les exécutions globales obtenues en composant les exécutions locales d'une telle famille satisfont la spécification.

2. Nous montrons ensuite que si une stratégie est gagnante, alors l'ensemble des exécutions locales autorisées par cette stratégie est inclus dans une de ces bonnes familles.

3. Nous montrons que nous pouvons décider, étant donné une telle famille, s'il existe une stratégie qui n'autorise que des exécutions dans cette famille. Pour ce faire, nous codons ce problème sous la forme d'un jeu régulier à deux joueurs.

4. Une fois que nous avons tout cela, nous pouvons procéder comme suit : Enumérer les bons ensembles de runs (qui ont une description bornée), et pour chacun d'entre eux, vérifier s'il existe des stratégies locales permettant d'appliquer cet ensemble d'exécutions locales à chaque processus.

Modèles et problèmes étudiés

Dans cette section nous présentons les trois principaux modèles étudiés dans cette thèse. Ces modèles sont de natures différentes, néanmoins la méthode développée ci-dessus s'applique dans les trois cas.

Broadcasts et automates à registre

Dans le chapitre 3, nous étudions un modèle paramétré, appelé «Broadcast Networks of Register Automata» (BNRA), avec un nombre arbitraire d'agents communiquant par broadcast. Les agents ont des identifiants uniques qu'ils peuvent s'envoyer les uns aux autres et stocker dans des registres locaux. Plus précisément, les processus communiquent au moyen de messages signés de la forme (m, d) avec m une lettre d'un alphabet fini et d une donnée d'un alphabet infini. Nous supposons que la topologie de communication évolue arbitrairement tout au long de l'exécution. En d'autres termes, l'ensemble des processus recevant chaque broadcast est arbitraire.

Chaque processus est modélisé comme un transducteur de registres à état fini qui peut diffuser le contenu d'un registre, stocker les données entrantes dans ses registres et comparer les données reçues au contenu de ses registres.

Le chapitre est centré sur le résultat suivant : étant donné un « réseau de broadcast d'automates de registres » avec certains états contrôlables, il est possible de déterminer s'il existe une stratégie locale garantissant qu'un état d'erreur q_{err} ne peut pas être atteint. Dans un cas particulier, nous obtenons la décidabilité du problème de l'atteignabilité des états pour ce modèle. Nous construisons ce résultat en le prouvant dans deux sous-cas de difficulté croissante. Cela devrait aider le lecteur à acquérir une certaine intuition sur la preuve générale, qui est assez technique.

Le reste du chapitre complète ce résultat de diverses manières : Nous montrons qu'un problème étroitement lié, dans lequel nous demandons si nous pouvons rassembler tous les agents dans un état donné, est indécidable. Nous discutons également de diverses extensions du modèle et donnons des limites de complexité correspondantes en fonction du nombre de registres que possède chaque processus.

Systèmes à locks

Passons au modèle étudié dans le Chapitre 4.

Deadlocks Un *deadlock* est une situation où plusieurs agents se bloquent parce que chacun attend une action des autres pour agir. Un exemple de deadlock est si votre fournisseur d'internet vous envoie un mail vous demandant de confirmer votre abonnement: Vous attendez d'avoir accès à vos mail pour confirmer, tandis qu'il attend votre confirmation pour vous donner l'accès internet. En informatique, ce genre de problème peut notamment arriver dans les systèmes distribués lorsqu'ils partagent des *locks*. Un *lock* est un marqueur utilisé pour contrôler l'accès à une ressource ou une variable partagée. Lorsqu'un agent veut utiliser cette ressource, il *prend le lock* associé, ce qui empêche les autres agents d'y accéder tant qu'il ne l'a pas *rendu*.

Notre premier modèle pour l'étude de ce genre de phénomènes consiste à considérer un nombre fixé de processus prenant et relachant des locks, chacun modélisé par un automate fini. Initialement ce modèle a été introduit avec des processus modélisés par des automates à piles [**KahIvaGup05**]. On se concentrera ici sur les systèmes à états finis, mais la plupart des résultats s'adaptent facilement au cas de processus à piles. Ces systèmes sont appelés LSS (Lock-Sharing Systems).

Les propriétés à vérifier sont, elles, écrites comme des automates sur des runs infinis, ou ω-automates. En effet, on veut pouvoir exprimer des choses comme "le système peut s'exécuter indéfiniment sans blocage".

Nous nous intéressons notamment à deux cas particuliers : celui où chaque processus utilise au plus deux locks (comme dans le célèbre dîner des philosophes), et celui où chaque processus gère les locks comme dans une pile (un processus ne peut relacher que le lock pris le plus récemment). Les LSS respectant ces conditions sont appelés respectivement 2LSS et nested LSS.

Nous montrons dans un premier temps que la vérification de LSS est PSPACE-complète, mais NP-complète si on se restreint à des 2LSS ou nested LSS. De même, la synthèse de contrôleurs est indécidable pour les LSS en général, mais décidable pour ces deux cas particuliers.

Pour analyser ces systèmes, on montre que les projections d'une exécution sur chaque processus peuvent être résumées en de petites descriptions contenant toute l'information nécessaire à la synchronisation, appelées *motifs* (*patterns* en anglais). Nous pouvons encoder les contraintes de locks comme des langages d'automates et résoudre les problèmes considérés par des méthodes classiques sur les langages réguliers. Dans le cas des 2LSS, nous montrons que les problèmes de deadlocks peuvent se traduire en problèmes de graphes, où les sommets sont des locks et les arêtes des processus. Ceci nous permet de caractériser plusieurs types de deadlocks et d'obtenir de meilleures bornes de complexité.

Systèmes dynamiques

Au vu des techniques utilisées dans la partie précédente, nous nous sommes demandé si ces résultats pouvaient être étendus au cas où les processus peuvent créer de nouveaux locks et d'autres processus pendant l'exécution. Ces systèmes sont alors appelés Dynamic LSS, ou DLSS [**LammichMW09**; **Kenter22**]. C'est le sujet du Chapitre 5.

Afin d'analyser les exécutions de ces systèmes, nous les représentons sous forme d'arbres. L'exécution du premier processus est écrite le long de la branche gauche partant de la racine. Si un nouveau processus est créé, nous ajoutons un fils à droite au point de création et nous écrivons l'exécution locale du processus créé le long de la branche gauche à partir de ce fils. L'ensemble des exécutions d'un tel système peut donc être codé sous la forme d'un langage d'arbres. Nous nous concentrons sur les exécutions *équitables* (*fair* en anglais), où les processus ne peuvent pas rester inactifs indéfiniment à moins qu'ils n'aient aucune action disponible. Nous montrons que le langage des exécutions équitables est reconnu par un automate de Büchi de taille exponentielle. Cela nous permet de vérifier les propriétés des arbres réguliers à l'aide d'un simple produit d'automate. Nous explorons ensuite trois extensions de ce modèle. La première concerne le cas où les processus sont des systèmes à piles. Nous montrons que cela n'affecte pas la complexité théorique du problème de vérification. Ensuite, nous combinons notre caractérisation des arbres représentant des exécutions équitables du système avec les *motifs* utilisés dans le chapitre précédent pour obtenir des *motifs étendus*. Nous montrons que le fait qu'une stratégie de contrôle soit gagnante ne dépend que de l'ensemble des motifs étendus qu'elle autorise, ce qui nous permet de décider le problème de la synthèse de contrôleurs. Enfin, nous ouvrons une nouvelle ligne de recherche en ajoutant une variable partagée au modèle. Nous montrons que la vérification devient alors indécidable. Pour récupérer la décidabilité, nous proposons de limiter le nombre de fois où le processus qui écrit sur la variable partagée change.

Chapter 1

Introduction

S'il y a dans le monde trop de sens incontestable, l'homme succombe sous son poids. Si le monde perd tout son sens, on ne peut pas vivre non plus.

Le livre du rire et de l'oubli
Milan Kundera

1.1 Motivations and history of distributed synthesis

The synthesis problem originates in Alonzo Church's seminal 1957 paper [**Church57**]. He poses the problem of constructing a circuit that, when given some inputs, produces outputs satisfying a given specification. A few years later, he introduced a version of the problem for reactive systems, in which the designed system has to interact with an environment. This problem was translated into a games formalism, and solved in this setting, in [**BuchiL69a**]. There has since been a rich history of studies of reactive synthesis and games. However, most of this body of work considers only sequential systems and specifications.

Distributed systems are made of several separate components working towards a common goal and equipped with communication mechanisms. They come with great gains in efficiency, as we can distribute tasks among several components. Those gains are even better when the systems can conduct their tasks without synchronising often. However, this advantage comes with a major drawback: they are notoriously prone to errors as it is easy to miss a bad interleaving of actions when building them. This has motivated the use of formal methods to verify and synthesize distributed systems in the 80s. For instance, the CTL logic was introduced with this motivation in mind [**ClaEme81**; **EmersonC82**]. The distributed synthesis problem has since proven to be highly challenging. Notably, Pnueli and Rosner [**PnuRos89**] demonstrated the undecidability of distributed synthesis for finite-state systems that communicate synchronously by messages with respect to LTL specifications, even with as few as three processes. Subsequent research showed that, essentially, the only decidable architectures are pipelines, where each process can send messages only to the next process in the pipeline [**ScheweF06**]. In addition, the complexity is non-elementary in the size of the pipeline. These negative results motivated the study of distributed synthesis for the model of *asynchronous automata* [**Zielonka87**], first with local strategies [**MadhusudanT01**; **Sznajder2009**] and then with so called *causal memory* [**GastinLZ04**; **MadhusudanTY05**]. In this setting, processes communicate by synchronising over actions. When they do, they have access to each other's information about what happened in the execution so far. It was shown that the distributed synthesis problem is decidable for co-graph action alphabets [**GastinLZ04**], and for tree architectures of processes [**GenestGMW13**; **MuschollW14**]. Yet the complexity is again non-elementary, this time with respect to the depth of the tree. Worse, it has been recently established that distributed synthesis with causal information is undecidable for unconstrained architectures [**Gimbert22**]. [1] Distributed synthesis for (safe) Petri nets [**FinkbeinerO17ic**] has encountered a similar line of limited advances, and due to [**Gimbert22**], is also undecidable in the general case, since distributed synthesis for asynchronous automata can be reduced to it [**BeutnerFH19**]. This history demonstrates the inherent difficulty of distributed synthesis.

In this thesis we present a new approach to distributed synthesis, with local strategies. We demonstrate its effectiveness by applying it to three different models with the common point that the communication between processes is very restricted. Let us briefly describe them. We will detail the history that led to these models in the related work section.

In Chapter 3 we consider networks of identical processes communicating by broadcast. We make the additional assumption that the communication network may evolve arbitrarily during runtime, meaning that the set of processes receiving each broadcast is chosen non-deterministically. Since this basic model is already well-understood, we consider its extension with data, called broadcast networks of register automata. At the start each process is given a unique datum d, its identifier. Processes now broadcast messages of the form (m, d) with m a letter from a finite alphabet and d a datum. They use local registers to store the received data, compare them for equality. and broadcast them. This is a parameterised setting: we consider runs with an arbitrary number of processes. We mostly consider the simple specification that a given error state q_{err} should never be reached.

In Chapter 4 we consider systems with a finite set of finite-state processes communicating by taking and releasing locks. We call them lock-sharing systems. This is a static setting: the set of processes is given, and stays the same throughout the executions. There we consider more diverse specifications: we define a class of objectives that lets us express several types of deadlocks, local regular conditions, and boolean combinations of those.

In Chapter 5 we extend the systems from Chapter 4 by allowing processes to spawn other processes and create new locks. We call the resulting model dynamic lock-sharing systems. As the name says, this is a dynamic setting: we start with a single process, but arbitrarily many new processes may be created in a run. As runs of these systems will be represented as binary trees, we will use regular tree languages as specifications.

[1] At the beginning of this thesis, the goal was to examine the distributed synthesis problem on asynchronous automata. However, shortly after I started my thesis, Hugo Gimbert came up with an undecidability proof [**Gimbert22**]. Hugo's proof uses only 6 processes. The problem is decidable with up to 4 processes, and the case of 5 processes is open.

1.2 Approach of this thesis

We begin by discussing our definition of distributed synthesis. In this thesis, we focus specifically on the synthesis of *controllers*: we consider pre-existing systems that include both controllable and uncontrollable actions, and aim to develop strategies for selecting controllable actions in a way that ensures a given specification is satisfied.

Furthermore, we assume that strategies are entirely local, meaning they rely solely on the sequence of operations performed by the process itself, with no access to the broader execution of the entire system. This assumption is justified by the fact that our models are abstractions of real systems, where the set of runs in the model over-approximates those of the actual system.

While this approximation is sound, it is not complete: an absence of errors in the model should imply an absence of errors in the system, but the reverse is not necessarily true. Thus, it is reasonable to under-approximate the information available to controllers about the system. If we can find a controller that operates effectively with limited information in the model, it should also work in the actual system.

We address the challenges of distributed synthesis with a new approach that involves identifying distributed models with restricted communication, where global synthesis can be decomposed into local sub-problems. In this context, “local” refers to the behaviour of individual processes, while “global” pertains to the joint behaviour of all processes. We start by illustrating the approach on an example, in which we apply this method to a simple model. Following this, we provide a high-level overview of the approach.

1.2.1 An illustrating example

Consider the model of Asynchronous Shared-Memory Systems (ASMS), where identical agents communicate by writing and reading from a shared variable [**Durand-Gasselin15**]. It is described by a finite-state transition system where transitions have two kinds of operations: reading and writing values from a finite alphabet Σ on a shared variable.

A run of an ASMS starts with an arbitrary number of agents in the initial state. At each step, one agent moves by either writing a new value in the variable, or reading the current value of the variable. Importantly, note that agents cannot read and write in one atomic step.

In addition to previous literature on this model, here we have controllable (rounds) and uncontrollable states (squares). A (local) strategy is a function $\sigma : \Delta^* \to \Delta$ that, given a sequence of transitions from Δ, chooses the next one. A local run $q_0 \xrightarrow{\delta_1} q_1 \xrightarrow{\delta_2} \cdots$ respects σ if for all i such that q_i is controllable we have $\delta_{i+1} = \sigma(\delta_1 \cdots \delta_i)$. A global run respects σ if the local run of every agent does. Suppose we have some controllable states and we are looking for a strategy guaranteeing that an error message m_{err} is never written. We call such a strategy *winning*.

We now go through the steps of our method to check whether a winning strategy exists.

Step 1: Define invariants We consider *invariants* that are sets of messages $I \subseteq \Sigma$. We say that a strategy satisfies an invariant if all it local runs respecting it are such that if they only read letters of I then they only write letters of I. A *good* invariant is one that does not contain m_{err}. Clearly a strategy satisfying a good invariant is winning: suppose m_{err} is written at some point in a run respecting σ. Since $m_{err} \notin I$, there is a

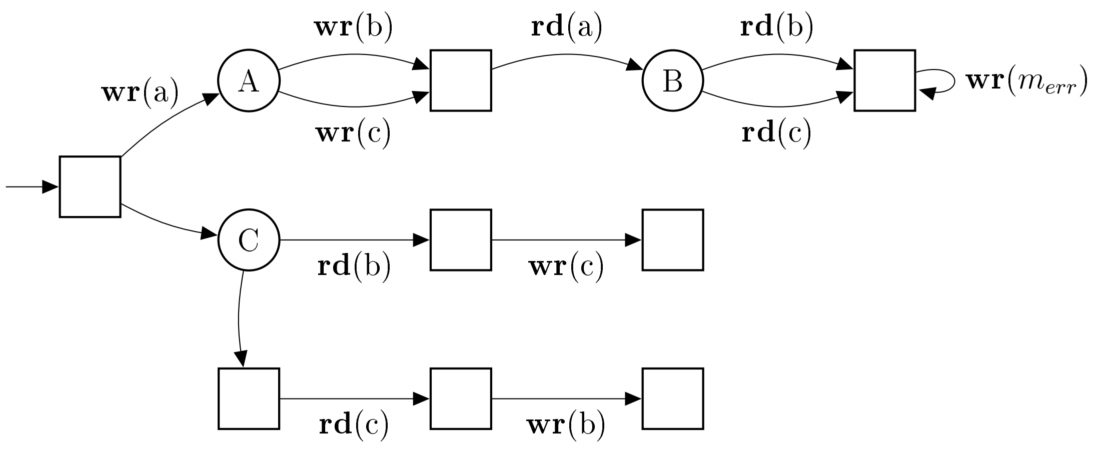


Figure 1.1: Example of an ASMS with controllable states. A winning strategy is to always pick the upper transition from A, the lower one from B and the lower one from C. This satisfies the invariant $\{a, b\}$: no local run writes c, and a local run writing m_{err} would need to read c first.

first agent who writes a letter $m \notin I$. Then that agent has written a letter outside of I without reading one before, a contradiction.

Step 2: Show that winning strategies induce invariants (of bounded size) Let us show that a winning strategy σ always satisfies a good invariant. Suppose σ is winning. Let I be the set of letters that are written in some σ-run. As σ is winning, $m_{err} \notin I$. We now show that σ respects I.

Consider a local run u respecting σ and only reading letters of I. We build a global run respecting σ and in which some agent follows the local run u. For each letter m read in u, by definition of I there is a run ϱ_m respecting σ in which m is written. We start with a sufficiently large number of agents and make an agent a execute u. Whenever agent a needs to read a letter m, we make some other agents execute ϱ_m until m is written. Then we make a read m and continue. We stop when u has been fully executed. This defines a run respecting σ such in which every letter written in u is written at some point. As a consequence, by definition of I, every letter written by u is in I.

Step 4: Show that one can check if there is a strategy satisfying an invariant Given an invariant I, checking if there is a strategy that satisfies it comes down to a simple reachability game: two players play on the graph of the transition system of the ASMS. If at some point a letter $m \notin I$ is read the first player (Controller) wins. If at some point a letter $m \notin I$ is written the second player (Environment) wins. If those things never happen then Controller wins.

Step 5: Algorithm We can non-deterministically guess an invariant $I \subseteq \Sigma$ and check if there is a strategy satisfying it (see Step 3). This yields an NP algorithm.

$\boxed{\textbf{As a result, the controller synthesis problem is decidable for ASMS.}}$

We will illustrate this method a second time (more formally) on a closely-related model in Section 3.3, by proving a similar result on networks of processes communicating by unreliable broadcasts. We will use the same invariants. See [**BalaC2021**] for the connection between the two models.

1.2.2 General description of our approach

More concretely, our approach can be summarised as the following scheme:

1. First we define a class of families of local runs, that have a bounded description. We identify "good" families of local runs as the ones such that no global, bad run can be obtained by composing local runs from such a family.

2. Then we show that if a strategy is winning then the set of local runs allowed by this strategy is included in a good family.

3. We show that we can decide, given such a family, whether there is a strategy that only allows runs in it. We do this by encoding this problem as a two-player game with a regular winning condition.

4. Once we have all of this, we can proceed as follows: Enumerate good sets of runs up to the bound, and for each one of them, check whether there are local strategies to enforce this set of local runs over every process.

Note that the good families of local runs described above are not only useful for distributed synthesis. As they characterise the satisfaction of a specification, they could be used to produce contracts that processes should satisfy, and build correct-by-construction systems.

The proofs will not always follow those steps exactly. In particular, in Chapters 4 and 5, a large part of the chapter will be on the verification problem. There, parts of this approach will be handled in the verification part.

On the way to study synthesis, it turns out that the verification problem is an interesting first step to understand the model and the communication mechanisms. We therefore include in this thesis results on verification as a first step towards the synthesis problem.

As a final touch, let us conclude this section by illustrating the distributed synthesis problem with the following riddle. It is commonly found in riddle books (with some variations), see for instance [**PrisonersLightbulb15**; **Deslandes2023**; **Winkler2024**]. An interesting aspect of this enigma is that it is easily modelled as a distributed synthesis problem with a lock and a shared boolean variable. We will discuss this kind of systems at the end of Chapter 5.

The prisoners and the lightbulb

100 prisoners have to play the following game: In the prison there is a room, empty except for a light switch. Every day, a prisoner is picked at random and brought to the room. They can see whether the light is on or off, and turn it on/off if they want. They are then returned to their cell. At all times a prisoner may decide to end the game by claiming that every prisoner has been in the room at least once. If they are right, all prisoners are released, otherwise, none of them is.
Prisoners cannot communicate during the game, but can agree on a strategy beforehand. How can they guarantee to be eventually released with probability 1?[a]

[a]Solutions may vary depending on whether you allow prisoners to count the days since the start of the game or not. In this thesis, we consider asynchronous systems, so we would look for a solution where prisoners do not keep track of time.

1.3 Overview of models

The choice of model is crucial in the analysis of distributed systems.

A first aspect is the means of communication. Among the common choices we have messages, pairwise interactions (*rendezvous*), synchronisation over actions, broadcasts, locks and shared variables.

Then, we also need to define the set of interacting processes. Some models, which we call *static*, operate over a fixed architecture, that is, the number of processes is known, and we also know who can communicate with whom. For instance, in models with message channels between processes, this would be given by a set of processes and a set of channels between pairs of processes. We will study such a model in Chapter 4.

Other models, which we refer to as *dynamic*, allow the creation of new processes during the execution, in the style of Petri nets. We can also mention for instance Dynamic Pushdown Networks or Regular model-checking (a general framework which allows one to represent systems made of many finite-state processes). The model presented in Chapter 5 fits in this class.

A third kind of model, which we refer to as *parameterised*, does not allow process creation, but the number of processes in the system is unknown. Typically, we consider a transition system describing a single process and try to check that the specification is satisfied by a system made of a composition of N processes with this same transition system, for all N. The model considered in Chapter 3 is of this kind. This class also contain population protocols and asynchronous shared-memory systems.

1.3.1 Static models

A long-studied family of models is the communicating finite-state systems, in which processes communicate by sending messages to each other. Those messages are not received immediately: they are stored in an intermediate data structure (often FIFO channels) and received by the other process after an arbitrary amount of time. The use of FIFO channels makes verification undecidable for these models, as they can simulate Turing machines. Several ways have been tried to recover decidability, wee for example the survey [**KuskeM21**]. One was to bound various parameters, for instance the maximum content of channels [**GenestKM07**] or the size of so-called *exchanges*. The idea is to focus on runs which can be cut into a bounded number of pieces with a particular form [**BouajjaniEJQ18**; **GiustoLL23**; **DelpyMS24**]. Another example is to bound some graph parameters on the so-called message communication chart, which describes the messages exchanged in an execution: see, for instance, [**Cyriac14**].

Finally, one can use abstractions to compute an approximation of the set of reachable configurations. A popular abstraction are lossy channel systems (LCS) [**AbdullaJ1996verif**]. In LCS, a finite set of processes communicate by pushing and popping messages from FIFO channels. However, at all times messages may get lost and disappear from the channel. It can be shown that the resulting system has decidable reachability, using the fact that the subword ordering is a well quasi-order over the set of configurations. This will be explained in Section 3.2.3. Moreover, it has been shown that LCS state reachability is complete for the complexity class $\mathbf{F}_{\omega^\omega}$ [**ChambartS08ordinal**; **Schnoebelen2002verifying**]. In Chapter 3 we show that the same is true for state reachability for the central model of the chapter, and prove the lower bound by simulating LCS, see Proposition 3.44.

Another approach to the modelisation of distributed systems is through distributed automata (also known as Zielonka automata) [**Zielonka87**]. They are an elegant model to recognise languages of traces, i.e., words up to permutation of independent actions. As mentioned before, there is a long line of study on distributed synthesis with causal memory, from its formal introduction in [**GastinLZ04**] to the recent undecidability result [**Gimbert22**]. Interestingly, while our approach consists in considering models with very restricted communication, an opposite trend exists. Sound negotiations (which can be seen as a special case of distributed automata) are a class of systems in which processes know at all times on which actions they will synchronise with each other process. They thus maintain a lot of information on the current configuration of the system, and this makes those systems much easier to verify [**EsparzaD13**; **EsparzaKMW18**].

As mentioned before, distributed synthesis in the Petri net model, called Petri games, has been proposed in [**FinkbeinerO17ic**]. The idea is that some tokens are controlled by the system and some by the environment. Note that while the number of tokens may change during runs, it is commonly assumed that the system has a bounded number of tokens in each place. This is why we place this model in the static section. For instance, the distributed synthesis problem is EXPTIME-complete for one environment token and arbitrary many system tokens [**FinkbeinerO17ic**], over a bounded Petri net.

1.3.2 Dynamic models

As we do not use Petri nets in this work, and as they come with a completely different set of techniques from ours, we will not dive into their history here. We can briefly mention some of the most celebrated results, with many applications in verification of infinite-state systems. On the one hand, the combined works of Lipton [**Lipton1976**] and Rackoff [**Rackoff78**] yield EXPSPACE-completeness of the coverability (or state reachability) problem. Those bounds were recently further refined (under some common complexity assumptions) [**KunnemannMSSW23**]. On the other hand, the reachability problem was recently proven to be $\mathbf{F}_\omega$-complete (also called Ackerman-complete) [**LerouxS19**; **CzerwinskiO21**].

Another way to model dynamic systems is through *regular model-checking*. In this framework, a configuration is a word over a finite alphabet Σ; it can represent the current states of finite-state processes ordered in a line [**BouajjaniJNT00**]. The transition relation is given by a finite-state transducer, which, given a configuration, outputs the next one. The question is then, given two regular languages $I, F \subseteq \Sigma^*$, whether we can reach F from a configuration of I in a finite number of steps.

While this problem is undecidable in general, several abstraction techniques exist to obtain sound (incomplete) algorithms for it, see for instance [**EsparzaRW22**; **CzernerEKW24**]. Typically, a way to prove that F is not reachable is to find an invariant, i.e., a language containing I, disjoint from F and stable by application of the transducer. More results on the regular model-checking problem can be found in the following surveys [**AbdullaJNS04**; **Abdulla21**]. In Chapter 3, a lot of decision procedures will rely on computing similar invariants. In our case, we will be able to restrict the analysis to downward-closed sets of words, that is, languages closed under the subword relation. We can rely on the theory of well quasi-orders to obtain sound and complete algorithms.

The first paper about a model similar to the one in Chapter 5 is [**BouajjaniMT05**]. It introduces Dynamic Pushdown Networks (DPNs). These consist of pushdown processes with spawn but no locks. The main idea is to represent a configuration as a sequence of process identifiers, each identifier followed by a stack content. Computing Pre^* of a regular set of configurations is decidable by extending the saturation technique from [**BouajjaniEM97**].

A conceptual step is made in [**LammichMW09**] where the authors introduce a tree representation of configurations. This is essentially the same representation as we use here. They extend DPNs by a fixed set of locks, and show how to adapt the saturation technique to compute Pre^* in this case. Their result is an EXPTIME decision procedure for verifying reachability of a regular set of configurations. This work has been extended to incorporate join operations [**GawlitzaLMSW11**], or priorities on processes [**DiazT17**]. Our work extends [**LammichMW09**] in two directions: it adds lock creation, and considers liveness properties. The saturation method has been adapted to DPNs with lock creation in the recent thesis [**Kenter22**]. The approach relies on hyperedge replacement grammars, and gives decidability without complexity bounds. Our liveness conditions can express this kind of reachability conditions.

In his thesis, Kenter adds to DPNs dynamic lock creation, as we do [**Kenter22**]. Configurations in [**Kenter22**] are represented as words, and the verification problem is to check if a system can reach a configuration in a given regular language. The proof still extends the saturation method and the approach relies on hyperedge replacement grammars. The main result is the decidability of configuration reachability, without complexity bounds. Compared to [**Kenter22**] we deal with liveness properties and give verification algorithms with matching complexity bounds.

Actually, the first related paper to deal with lock creation is probably [**YasukataT016**]. The authors consider a model of higher-order programs with spawn, joins, and lock creation. Apart from nested locking, a new restriction of scope safety is imposed. Under these conditions, reachability of pairs of states is shown to be decidable. The works above have been followed by implementations [**Lammich11**; **YasukataT016**; **DiazT17**]. In particular [**DiazT17**] reports on verification of several substantial size programs and detecting an error in xvisor.

Let us comment on shared state and global variables. These are not present in the above models because reachability for two pushdown processes with one lock and one global variable is already undecidable. There is an active line of study of multi-pushdown systems where shared state is modeled as global control. In this model decidability is recovered by imposing restrictions on stack usage such as bounded context switching and variations thereof [**QadeerR05**; **TorreMP07**; **TorreNP20**; **AkshayGKR20**]. Observe that these are restrictions on global runs, and not on local runs of processes, as we consider here. Finally, in [**BaumannMTZ20**] a model of threaded pools, without locks, is introduced. There, verification is decidable, once again assuming bounded context switching. But the complexity of this model is as high as Petri net coverability [**BaumannMTZ22**].

Note that the study of those models focuses on verification. We are not aware of existing approaches to controller synthesis that resemble the one of Section 5.6. Some games have been defined on arenas defined by systems with dynamic process creations, for instance in [**BolligLS18**], but they are sequential two-player games and require completely different techniques to be analysed.

1.3.3 Parameterised models

Parameterised verification studies models equipped with a parameter (typically the number of processes) and questions of the form "Does this specification hold for any value of the parameter?".

One of the first frameworks in which parameterized verification has been used is *token-passing systems*, in which a network of identical processes communicate by passing a token around from neighbour to neighbour. The first result on this kind of system was negative: In [**Suzuki88**] it is shown that verification of systems where an arbitrary number of processes organised in a ring communicate by passing a token carrying a value is undecidable. Decidability can be recovered by bounding the number of times the token changes value [**EmersonK04**]. Emerson and Namjoshi showed several cut-off properties when the token does not carry a value. They consider families of specifications which build upon the CTL* logic (without the X operator). In order to check that a property from one of those families is satisfied on all rings, it suffices to check it on rings with up to 5 agents [**EmersonN95**]. Cut-offs were also exhibited for general token graphs, this time with specifications based on LTL without X [**ClarkeTTV04**]. The authors also show that this cannot be extended to CTL specifications. The existence of cut-offs for general graphs with CTL*-like specifications was further investigated in [**AminofJKR14**].

In parallel of this line of research, another type of parameterised model emerged, with rendezvous communication: the processes interact by pairs. The analysis of parameterised systems with rendezvous communications was initiated by the seminal work of German and Sistla [**GermanS92**]. The problem of deciding the existence of cut-offs for this model was investigated by Horn and Sangnier [**HornS20**], and then by Balasubramanian, Esparza and Raskin, who improve the complexity bounds [**BalasubramanianER23**].

In 2004, population protocols were introduced, where the rendezvous communication is used to compute predicates by consensus [**AngluinADFP04**]. Most of the time, they are seen as a model of computation studied from the point of view of expressivity and succinctness. However, there is a line of research that aims to verify various properties on them, such as whether they correctly compute some predicate. As those verification problems are as hard as Petri net reachability [**EsparzaGLM17**], some subclasses are considered, on which parameterised verification techniques can be used [**EsparzaJRW21**; **Weil-Kennedy23**].

In parallel of rendezvous communications, another line of research was developed around broadcasts. In 2010, Delzanno, Sangnier and Zavattaro introduced a formal verification approach to Ad Hoc networks, which are networks in which the connection topology is built up dynamically during runtime [**DelzannoSZ2010Adhoc**]. Processes communicate by broadcasting messages to their neighbours. In their model, all processes are identical and represented by a finite-state system with two operations, broadcasts and receptions. Their initial question is whether a given protocol will avoid a designated error state no matter the network topology. This problem turns out to be undecidable, justifying an additional assumption: the topology may change during the execution. The resulting model is called Reconfigurable broadcast networks, and has been the subject of many studies since its creation. Other attempts exist to verify Ad Hoc networks: for instance in [**DelzannoSZ11cliques**] the underlying graph of the network is assumed to belong to some restricted classes (on which the induced subgraph ordering is a well quasi-order). In [**DelzannoSZ12**] the authors look at the model when nodes in the communication graph may sometimes fail to receive messages. In [**AbdullaAR13**] they are restricted to DAGs of bounded depth. In [**Balasubramanian18**] the number of reconfigurations between broadcasts is bounded. Guillou, Sangnier and Snajder recently showed decidability when each process can alternate between broadcasts and receptions at most once [**GuillouSS24arxiv**].

A general model integrating both broadcasts and rendezvous communications was proposed by Emerson and Namjoshi [**EmersonN98**]. They present an algorithm computing increasing under-approximations of the set of reachable configurations, but do not show termination in general. In fact, it was proven in [**EsparzaFM1999verification**] that this algorithm may not terminate. They establish that liveness is undecidable, but safety is decidable thanks to a well quasi-order argument.

Let us now switch to a model closely related to reconfigurable broadcast networks. In 2011, Hague proposed a model where processes communicate by writing and reading from a shared variable[2] [**Hague11**]. Crucially, processes are only allowed to read the current value of the memory or write a new one, but never both at the same time. Hague shows that state reachability is decidable for systems made of one leader and arbitrarily many users, each one being modelled as a pushdown system. Note that in the non-parameterised case, when we have a fixed set of processes, this non-atomicity restriction does not help [**AtigBKS14**]. Decidability for parameterised systems was later generalised to a large class of models, which have effectively computable downward-closures [**MuschollSW17**]. The complexity of safety and liveness for this model was further investigated in [**Durand-Gasselin15**] and [**EsparzaGM16**], both for finite-state and pushdown processes. It was later shown in [**FortinMW17**] that verification of stutter-free LTL properties is NExpTime-complete on this model, with pushdown processes. All those results consider adversarial scheduler; in [**BouyerMRSS16**] the authors initiate a study of the model with randomised schedulers. The basic model was extended to allow processes to spawn new processes during the execution, with the restriction that they can only spawn an arbitrary number of new processes [**MuschollSW17**]. This allows parameterised verification arguments to be applied. Some more general criteria on the transition systems of individual processes yielding decidability are given in [**LaTorreMW15**]. Another extension is with *rounds*, in which processes increase their rounds asynchronously, and can access variables in a bounded window centred on their current round. The authors show that this can be used to model and verify procedures such as Aspnes' algorithm [**BertrandMSW22**; **Waldburger23**].

As we will study parameterised systems with data, let us review a few related mod-

[2]This is the model used earlier in Section 1.2.1

els. Many approaches exist to define parameterised models with registers. In dynamic register automata, processes are allowed to spawn other processes with new identifiers and communicate integers values [**AbdullaAKR14**]. While basic problems on these models are in general undecidable, some restrictions on communications allow to obtain decidability [**AbdullaAKR15**; **Rezine17**]. We can also mention distributed memory automata, where each process accesses a local register and a shared register, and can query the number of other processes sharing its register value. Those can be seen as a model of register automata with a parameterised number of registers.

Such parameterised verification problems often relate to the theory of well quasi-orders and the associated high complexities obtained from bounds on "bad sequences" in ordered sets. In particular, our model is linked to two classical models from this field. The first one is data nets, which are Petri nets in which tokens are labelled with natural numbers and can exchange and compare their labels [**LazicNORW08**]. In general, inequality tests are allowed, but data nets with only equality tests have also been studied [**Rosa-Velardo17**]. They do not subsume BNRA as, in data nets, each process can only carry one integer at a time (problems on models of data nets where tokens have tuples of integers as labels are typically undecidable).

There are many more formal models for parameterised verification. Let us give a couple pointers. For more global views of the field, see the book [**BloemBook2015**], as well as the chapter on parameterised model-checking from this other book [**Abdulla:2018aa**]. See also Esparza's survey on parameterised systems which compares the computational power of such systems depending on the presence of leaders and the means of communication (broadcast or rendezvous) [**Esparza14**]. In [**AminofRZ15**], the expressivities of broadcast, rendezvous, and other primitives are compared.

Beyond verification, going to the parameterised version of the strategy synthesis problem can help regain decidability. Specifically, we will see an example of synthesis problems that are undecidable for a fixed number of agents, but decidable when the number of agents is arbitrary.

This approach to parameterised distributed synthesis was applied by Jacobs and Bloem in the case where processes are organised in a ring and communication is made by passing a token along the ring [**JacobsB14**].

In [**BertrandFS15**; **Fournier15these**], a decidability frontier is drawn for a problem that can be seen as an instance of controller synthesis for broadcast networks. The question is the existence of a bad run, but runs in which agents react differently upon receiving the same sequence of messages are ignored. This essentially comes down to checking the existence of a losing local control strategy. One of their results is that this problem is NP-complete for reconfigurable broadcast networks with respect to state reachability and synchronisation in a state. Another approach was initiated by Stan during his PhD thesis [**Stan17**], over broadcast networks with randomised schedulers and some randomised transitions. He shows that controller synthesis is decidable and even tractable against several natural properties, for instance reaching a state with positive probability.

We can also mention some parameterised games, which do not fit in the framework of distributed synthesis but may call for similar techniques. Concurrent parameterised games were introduced in [**BertrandBM20**]. Agents operate synchronously, and choose actions concurrently. A major point is that agents can see the global state of the system after each action. Population games [**BertrandDGGG19**] also involve an arbitrary amount of agents, but the games are two-player sequential games. A short section is dedicated to them in Appendix B.

1.4 Structure and content of the thesis

The content of this thesis is divided in three chapters. Each chapter deals with one model of distributed systems. In the introduction of each chapter we present the history of the model and related results from the literature. The chapters can be read independently, although some aspects of Chapter 5 may be easier to understand in light of Chapter 4.

- In Chapter 3 we study a parameterised model, called broadcast networks of register automata with arbitrarily many agents communicating using broadcasts. The agents have unique identifiers which they can send to each other and store in local registers. More precisely, processes communicate through signed messages of the form (m, d) with m a letter from a finite alphabet and d a datum from an infinite alphabet. We assume that the communication topology evolves arbitrarily throughout the execution. In other words, the set of processes receiving each broadcast is arbitrary.

  Each process is modelled as a finite-state register transducer that can broadcast the contents of a register, store incoming data in its registers, and compare received data against the contents of its registers.

  The chapter is centred on the following result: given a broadcast networks of register automata with some controllable states, it is decidable whether there is a local strategy guaranteeing that an error state q_{err} cannot be reached. As a particular case, we obtain decidability of state reachability problem for this model. We build up to this result by proving it in two subcases of increasing difficulty. This should help the reader gain some intuition on the general proof, which is quite technical.

  The rest of the chapter complements this result in various ways: We show that a closely-related problem, in which we ask whether we can gather all agents in a given state, is undecidable. We also discuss various extensions of the model, and give matching complexity bounds depending on the number of registers each process has. The decidability of the state reachability problem for those systems was presented in a 2024 paper [**GuillouMW24**], but most of the chapter is new: the extension to the synthesis problem requires a very different approach.

- In Chapter 4 we consider a simple model with processes communicating only by taking and releasing locks. First we focus on systems in which each process uses at most two locks. We show that we can associate with each run a pattern from a finite set, and that we can decide whether a family of local runs can be combined into a global run based solely on their patterns. We use this characterisation to translate the verification and controller synthesis problems into graph properties.

  Then we consider systems where processes may use arbitrarily many locks, but manage them according to a stack discipline: a process can only release the lock taken latest among the ones it holds. Once again, we come up with a finite set of patterns that let us characterise schedulable families of local runs. Most of the content of this chapter comes from the combination of a 2022 paper [**GimMMW22**] and an unpublished preprint [**Mascle23**], with some minor additions.

- In Chapter 5 we extend this model with operations spawning new processes. In order to analyse the possible executions of such systems, we represent them as trees. The run of the first process is written along the left branch from the root. If a new process is spawned, we add a right child at the point of the spawn and write the local execution of the spawned process along the left branch from that child. The set of executions of such a system can therefore be encoded as a tree language. We focus on *fair* executions, where processes cannot stay idle forever unless they have no available action. We show that the language of fair executions is recognised by a Büchi tree automaton of exponential size. This lets us verify regular tree properties through a simple automaton product. We then explore three extensions of this model. The first one is when processes are pushdown systems. We show that this does not worsen the theoretical complexity of the verification problem. Then, we combine our characterisation of trees representing fair executions of the system with the patterns used in Chapter 4 to obtain extended patterns. We show that whether a control strategy is winning depends only on the set of extended patterns it allows, which lets us decide the controller synthesis problem. Finally, we open a new research direction by adding a shared variable to the model. We show that the state reachability problem then becomes undecidable. To mitigate this result, we propose to bound the number of times the process writing on the shared variable changes. The part about verification of (pushdown) dynamic systems is adapted from a 2023 publication [**MascleMW23**]. The two last sections, on synthesis and addition of variables, are completely new.

- In addition to the main content, in Appendix A we show a run reduction technique for register automata that yields a simplified proof of decidability for one of the problems considered in Chapter 3 (where it is derived as a corollary of a more general theorem).

  Furthermore, Appendix B offers a brief survey of open problems on population games. It does not directly relate to the rest of the thesis, which is why it is in the appendix. It should be of interest to people studying parameterised verification, in the style of Chapter 3.

Reading tips

This thesis uses the knowledge package. When we define an *important term* it is coloured in red. Occurrences of that important term are coloured in blue. The reader can click on those (or just hover over them on some PDF readers) to see the definition.

Although chapters follow a common method, they can be read independently. As we consider different models in each part, the objects defined in a chapter are rarely used in others. In addition to the three main chapters, we add some appendices which are not directly relevant to the main results, but may be interesting for the reader.

Wishing you a pleasant read!

Chapter 2

Preliminaries

Basique, simple, simple, basique

ORELSAN

2.1 Terminology for classic objects

Sets. Given a set S, we write $|S|$ for its cardinality and 2^S for the set of its subsets. The intersection of two sets A, B is written $A \cap B$. Their union is written $A \cup B$, or sometimes $A \sqcup B$ when A and B are disjoint. The complement of a subset S of A is written S^{c} (A will always be clear from the context). Given two integers $i, j \in \mathbb{N}$, we write $[i, j]$ for $\{k \in \mathbb{N} \mid i \leq k \leq j\}$.

Words. A *finite word* $w = a_1 \cdots a_n$ over A is a finite sequence of elements of A. Its *length* $|w|$ is its number of elements n. The set of finite words over A is written A^*. We denote the empty word by ε. Analogously, an *infinite word* $w = a_1 a_2 \cdots$ over A is an infinite sequence of elements of A. The set of infinite words over A is written A^ω. The union of A^* and A^ω is written A^∞.

Given two words $u \in A^*$ and $v \in A^\infty$, we write $u \cdot v$ or simply uv for the concatenation of u and v. We say that $u \in A^*$ is a *prefix* of v if there exists $u' \in A^\infty$ such that $v = uu'$. Similarly, $u \in A^\infty$ is a *suffix* of $v \in A^\infty$ if there exists $u' \in A^*$ such that $v = u'u$.

Trees. An *ordered tree* τ is a subset of $\mathbb{N}^*$ such that for all $\nu \in \tau$:

- If $\nu \in \tau$ then all its prefixes are in τ.
- For all $i \in \mathbb{N}$, if $\nu \cdot i \in \tau$ then $\nu \cdot j \in \tau$ for all $j \leq i$.

An element of τ is called a *node*. If a node ν' is a prefix of a node ν, we say that ν' is an *ancestor* of ν, and ν a *descendant* of ν'. If furthermore $|\nu| = |\nu'| + 1$ then we say that ν' is the *parent* of ν, and ν a *child* of ν'. If $\nu' 0^k = \nu$ for some $k \in \mathbb{N}$ we say that ν is a *left descendant* of ν'.

A *branch* is a subset of nodes closed by prefix, totally ordered by prefix and such that if a node of the branch has a child, then it has a child in that branch. The *root* of τ is the empty word ε. A *leaf* is a node with no children. The *subtree rooted in* ν is the set of descendants of ν.

Graphs. A *graph* is a pair $G = (V, E)$ with V a set of *vertices* and $E \subseteq V^2$ a set of *edges*. It is *finite* if V is. It is *undirected* if for all $(u, v) \in E$, we have $(v, u) \in E$. A *path* in G is a sequence of edges $v_0 \to v_1 \cdots \to v_n$. It is a *cycle* if $v_n = v_0$, and a *simple cycle* if furthermore all $(v_i)_{0 \leq i \leq n-1}$ are distinct. We say that t is *reachable* from s if there is a path from s to t. We say that G is *strongly connected* if every vertex is reachable from every other.

A *subgraph* of G is a graph (V', E') with $V' \subseteq V$ and $E' = E \cap V'^2$. A *strongly connected component* (SCC) of G is a maximal strongly connected subgraph of G.

Functions. Given a function $f : \mathbb{N} \to \mathbb{N}$, we call it *polynomial* if there exist $K, d \in \mathbb{N}$ such that $f(n) \leq n^d + K$ for all $n \in \mathbb{N}$. We call it *exponential* if there exist $K, d \in \mathbb{N}$ such that $f(n) \leq 2^{n^d} + K$ for all $n \in \mathbb{N}$. We call it *doubly-exponential* if there exist $K, d \in \mathbb{N}$ such that $f(n) \leq 2^{2^{n^d}} + K$ for all $n \in \mathbb{N}$.

Define $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as $g(n, 0) = n$ and $g(n, d+1) = 2^{g(n,d)}$ for all $n, d \in \mathbb{N}$. We call f *elementary* if there exist K, d such that $f(n) \leq g(n, d) + K$ for all $n \in \mathbb{N}$.

2.2 Automata

Finite automata. A *finite automaton* is defined by a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ where Q is a finite set of *states*, Σ a finite *alphabet*, $\Delta \subseteq Q \times \Sigma \times Q$ a set of *transitions*, which are written $q \xrightarrow{a} q'$, $I \subseteq Q$ a set of *initial states* and $F \subseteq Q$ a set of *final states*. We will sometimes view Δ as a function $Q \times \Sigma \to 2^Q$. We say that $\mathcal{A}$ is *deterministic* if $|I| \leq 1$ and $|\Delta(q, a)| \leq 1$ for all $q \in Q, a \in \Sigma$. We use the acronyms NFA for non-deterministic finite automaton and DFA for deterministic finite automaton.

A *run* of $\mathcal{A}$ over a word $w = a_1 \cdots a_k$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k$ such that $q_0 \in I$. It is accepting if $q_n \in F$. The *language* of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$ is the set of words which have accepting runs in $\mathcal{A}$.

Pushdown automata and context-free grammars. A *pushdown automaton* is defined by a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where Q is a finite set of *states*, Σ and Γ finite *alphabets*, $\Delta \subseteq Q \times \Sigma \times \{\mathtt{push}(z), \mathtt{pop}(z), \mathtt{nop} \mid z \in \Gamma\} \times Q$ a set of *transitions*, $I \subseteq Q$ a set of *initial states* and $F \subseteq Q$ a set of *final states*.

A *configuration* is an element of $Q\Gamma^*$. A step $qy \xrightarrow{a} q'y'$ with $q, q' \in Q$ and $y, y' \in \Gamma^*$ is defined when either:

- $(q, a, \mathtt{push}(z), q') \in \Delta$ and $y' = zy$, or
- $(q, a, \mathtt{pop}(z), q') \in \Delta$ and $zy' = y$, or
- $(q, a, \mathtt{nop}, q') \in \Delta$ and $y' = y$

A *run* of $\mathcal{A}$ over a word $w = a_1 \cdots a_k$ is a sequence of steps $(q_0, w_0) \xrightarrow{a_1} (q_1, w_1) \xrightarrow{a_2} \cdots \xrightarrow{a_k} (q_k, w_k)$ such that $q_0 \in I$ and $w_0 = \varepsilon$. It is accepting if $q_n \in F$. The *language* of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$ is the set of words which have accepting runs in $\mathcal{A}$.

A *context-free grammar* is defined by a tuple $\mathbf{G} = (NT, \Sigma, R, S)$ with NT a finite alphabet of non-terminals, Σ a finite alphabet of terminals, S an initial non-terminal and R a set of rules of the form $X \to w$ with $X \in NT$ and $w \in (NT \cup \Sigma)^*$. Given two words $u, v \in (NT \cup \Sigma)^*$, we say that u *directly yields* v, written $u \Rightarrow v$, if there is a rule $X \to w \in R$ and two words $u_1, u_2 \in (NT \cup \Sigma)^*$ such that $u = u_1 X u_2$ and $v = u_1 w u_2$.

We write $\overset{*}{\Rightarrow}$ for the reflexive transitive closure of $\Rightarrow$. The language of $\mathbf{G}$ is the set $\mathcal{L}(\mathbf{G}) = \{v \in \Sigma^* \mid S \overset{*}{\Rightarrow} v\}$.

Proposition 2.1 ▶ Folklore

Pushdown automata and context-free grammars recognise the same languages.

2.3 Automata on infinite words

When we deal with potentially infinite runs, we will need to use suitable automata. There are many ways to define finite automata reading infinite words. Here, we will focus on languages defined by a finite automaton whose acceptance condition is based on the set of states visited infinitely often.

A *Muller automaton* is defined by a tuple $\mathcal{A} = (Q, \Sigma, \Delta, \mathit{init}, \Gamma, \mathbf{col}, \mathcal{F})$ with

- Q a finite set of states,
- Σ a finite alphabet of letters,
- $\Delta \subseteq S \times \Sigma \times S$ a transition relation,
- $\mathit{init} \in S$ the initial state,
- Γ a finite alphabet of colours,
- $\mathbf{col} : S \to 2^\Gamma$ a colouring of the states[1].
- $\mathcal{F} \subseteq 2^\Gamma$

A *run* of $\mathcal{A}$ over a word $w = a_1 a_2 \cdots \in \Sigma^\omega$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots$ such that $q_0 \in I$. It is accepting if $\{c \mid \forall i, \exists j \geq i, c \in \mathbf{col}(q_j)\} \in \mathcal{F}$, i.e., if the set of colours seen infinitely often during the run is in $\mathcal{F}$. The *language* of $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$ is the set of words which have accepting runs in $\mathcal{A}$.

We will sometimes deal with languages that contain both finite and infinite words. In this case, we will add an extra component to this tuple, a set of states F. The language of the automaton is then the language of the Muller automaton plus all finite words that have a run ending in F.

An *Emerson-Lei automaton* (ELA for short) is a Muller automaton where the acceptance condition $\mathcal{F}$ is described by a Boolean formula φ over variables $\{\inf_\gamma \mid \gamma \in \Gamma\}$. The formula defines the accepting condition $\mathcal{F}(\varphi) = \{G \subseteq \Gamma \mid \nu_G \text{ satisfies } \varphi\}$ where ν_G is the valuation over $\mathcal{G}$ mapping $\inf_\gamma$ to true if $\gamma \in G$ and to false otherwise.

[1]Apologies to Antonio Casares, who spent a significant amount of time demonstrating to me that transition-based automata are by far preferable to state-based automata. While there are indeed good arguments supporting this [**CasaresThesis23**], in this thesis we will sometimes need to deal with languages of both finite and infinite words, making state-based automata more convenient.

Parity automata are the subclass of Muller automata where the acceptance condition $(\Gamma, \mathbf{col}, \mathcal{F})$ is replaced by a function $\texttt{pr} : S \to \mathbb{N}$ mapping states to priorities. A run is accepting if the highest priority seen infinitely often is even, which is easily encoded as a Muller acceptance condition.

Büchi automata are the subclass of parity automata that only use priorities 1 and 2. Historically, this was the first type of automata on infinite words to be introduced [**Buchi62**].

These automata all recognise the same class of languages. While Muller and parity automata can be made deterministic, this is not the case for Büchi automata.

Proposition 2.2 ▶ [McNaughton66; Mostowski84]

The following types of automata recognise the same languages.

- Non-deterministic Muller automata
- Deterministic parity automata
- Non-deterministic Büchi automata

There has been intensive study on the cost of translating from each of those models to the others. We refer to Udi Boker's webpage for a survey of state-of-the art automata translations. In this thesis we will only use the following propositions. The first one follows from the *Latest appearance record* of Gurevich and Harrington.

Proposition 2.3 ▶ [GurevichH82]

Every Muller automaton with n states and using k colours can be converted into a parity automaton with $k \cdot k! \cdot n$ states and k priorities. If the initial automaton is deterministic then so is the resulting one.

The second one gives a determinisation of parity automata with an exponential state blow-up.

Proposition 2.4 ▶ [Safra88; Piterman07]

Every non-deterministic parity automaton with n states and using k priorities can be converted into a deterministic parity automaton with $2n^n(k+1)^{n(k+1)}(n(k+1))!$ states and using $2n(k+1)$ priorities.

Finally, let us mention that the complexity of checking emptiness of one of those automata depends on the class considered. The two results below will be used several times in what follows.

Proposition 2.5 ▶ [EmersonL1987]

Given a parity automaton $\mathcal{A}$, checking if $\mathcal{L}(\mathcal{A}) = \emptyset$ is decidable in polynomial time.

Proposition 2.6 ▶ [EmersonL1987; BaierBDKMS19]

Given an Emerson-Lei automaton $\mathcal{A}$, checking if $\mathcal{L}(\mathcal{A}) = \emptyset$ is NP-complete.

2.4 Automata on infinite trees

Given an alphabet A, an A-labelled binary tree is a tree τ along with a labelling function $\lambda : \tau \to A$. When the alphabet is clear from context, we will simply call (τ, λ) a tree, since we only consider binary trees in this thesis.

A *parity tree automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, q_{init}, F, \mathtt{pr})$ with Q a finite set of states, Σ a finite alphabet, $\Delta \subseteq Q \times \Sigma \times (Q \cup Q \times Q)$ a transition function, $q_{init} \in Q$ the initial state, $F \subseteq Q$ final states and $\mathtt{pr} : Q \to \mathbb{N}$ a function mapping states to priorities.

Let τ be an A-labelled binary tree, a run of $\mathcal{A}$ over τ is a Q-labelling $\varrho : \tau \to Q$ of the same tree such that the root is labelled q_{init} for all $\nu \in \tau$, if $\nu 1 \in \tau$ then $(\rho(\nu), \tau(\nu), \rho(\nu 0), \rho(\nu 1)) \in \Delta$, and if only $\nu 0 \in \tau$ and $\nu 1 \notin \tau$ then $(\rho(\nu), \tau(\nu), \rho(\nu 0)) \in \Delta$.

A run is accepting if every leaf is labelled with a state of F and for all infinite branch, the highest priority seen infinitely often along that branch is even. The language of $\mathcal{A}$ is the set of trees for which there exists an accepting run.

2.5 Games

2.5.1 Regular games

Definition 2.7

An *ω-regular game* $\mathcal{G}$ is given by a directed graph $G = (V, E)$ called the *arena*, along with a partition of V in two, $V = V_0 \sqcup V_1$, an initial vertex v_{init}, a colouring function $c : V \to C$ mapping vertices to a finite set of colours C, and an ω-regular language $\mathcal{L} \subseteq C^\omega$, called the *objective*.

The game goes like this: we place a token on the initial vertex v_{init}. Then two players, P_0 and P_1, move the token as follows. When the token is on a vertex v of V_0, P_0 selects a transition from v to a vertex v' and we move the token to v'. P_1 does the same from vertices of V_1. Player P_0 wins if the resulting sequence of vertices $v_0 v_1 v_2 \cdots$ is so that $c(v_0)c(v_1) \cdots \in \mathcal{L}$. In the literature those two players are often called Eve and Adam, or sometimes Controller and Environment in the context of reactive synthesis. In this thesis we will use the latter names.

A (finite or infinite path) in G starting in v_{init} is called a *play*. A play $v_0 \to v_1 \to \cdots$ is *winning* for P_0 if $c(v_0)c(v_1) \cdots \in$, and *losing* for P_0 otherwise.

A *strategy* for player P_i is a function $\sigma_{\mathcal{G}} : V^* V_i \to V$. A *$\sigma_{\mathcal{G}}$-play* is a path $v_0 \to v_1 \to \cdots$ in G such that for all $j \geq 1$, if $v_{j-1} \in V_i$ then $v_j = \sigma_{\mathcal{G}}(v_0 \cdots v_{j-1})$. A strategy for P_0 (resp. P_1) is *winning* if all infinite $\sigma_{\mathcal{G}}$-plays are winning (resp. losing) for P_0.

When the objective is described by a deterministic parity (resp. Büchi) automaton we call $\mathcal{G}$ a *parity game* (resp. *Büchi game*).

As particular cases, we call $\mathcal{G}$ a *reachability game* (resp. *safety game*), when the objective is of the form LV^ω (resp. $V^\omega \setminus LV^\omega$) with L a regular language of finite words. In both of those cases, we assume that the objective is given by a DFA $\mathcal{A}$ recognising L.

It is well-known that all those games are *determined*, i.e., in every game one of the two players has a winning strategy.

Proposition 2.8 ▶ Folklore

One can compute the winner of a finite reachability game in polynomial time. Furthermore if P_0 has a winning strategy, then she has one that guarantees that she wins in at most $|V| \cdot |\mathcal{A}|$ steps.

A strategy is *positional* if its output only depends on the current state, that is, for all $w, w' \in V^*$ and $v \in V$ we have $\sigma(wv) = \sigma(w'v)$.

A celebrated result in the field is the following quasi-polynomial algorithm for solving parity games.

Proposition 2.9 ▶ [CaludeJKLS17; CaludeJKLS22]

One can compute the winner of a finite parity game in time $|V|^{O(log(d))}$. Furthermore we obtain a positional winning strategy for the winner.

See [**GamesOnGraphsarxiv**] for an overview of results concerning ω-regular games.

2.5.2 Pushdown games

A *pushdown game* is a two-player ω-regular game over an infinite arena defined as the configuration space of a pushdown automaton.

Formally, it is described by a pushdown transition system $(Q, Q_0, Q_1, \Delta, \Gamma, q_0, c, \mathcal{L})$ with

- Q a finite set of states, partitioned into $Q = Q_0 \sqcup Q_1$
- Γ is a finite alphabet
- $\Delta \subseteq Q \times \{\mathtt{push}(z), \mathtt{pop}(z) \mid z \in \Gamma\} \times Q$.
- $q_0 \in Q$ the initial state
- $c : Q \to C$ a colouring function
- $\mathcal{L} \subseteq C^\omega$ an ω-regular language over C

The arena is formed by the set of vertices $Q\Gamma^*$, and the edges $E = \{(qyz, q'y) \mid (q, \mathtt{pop}(z), q') \in \Delta\} \cup \{(qy, q'yz) \mid (q, \mathtt{push}(z), q') \in \Delta\}$. The initial vertex is q_0. The vertices of player P_0 (resp. P_1) are the elements of $Q_0\Gamma^*$ (resp. $Q_1\Gamma^*$). Each vertex qy is coloured with $c(q)$.

Lemma 2.10 ▶ [Walukiewicz01]

Pushdown games with an objective given by a deterministic parity automaton can be solved in exponential time.
Furthermore, in the particular case of reachability games, if the first player wins, he can do so within an exponential number of steps.

2.6 Classic decision problems

In this section we present a few decision problems that we will use to show undecidability and complexity lower bounds in this thesis. The problems are sorted by complexity.

3-SAT A Boolean formula φ is in *3-CNF* if it is made of a conjunction of *clauses* $\varphi = \bigwedge_{i=1}^{m} C_i$, where each clause C_i is a disjunction of 3 *literals* $C_i = \ell_i^1 \vee \ell_i^2 \vee \ell_i^3$, and each literal is either a variable or its negation.

The *3SAT problem* takes as input a Boolean formula in 3-CNF over a set of variables $x_1, \ldots, x_n$, and asks if there exists a valuation $\nu : \{x_1, \ldots, x_n\} \to \{\bot, \top\}$ satisfying the formula. This problem is NP-complete [**Karp72**].

Independent set The *independent set problem* asks, given a finite undirected graph $G = (V, E)$ and $k \in \mathbb{N}$, if there exists a set of vertices $I \subseteq V$ such that $|I| = k$ for all $(u, v) \in E$, either $u \notin I$ or $v \notin I$. This problem is NP-complete [**GareyJS1976**].

3-colouring The *graph 3-colouring problem* asks, given a finite undirected graph $G = (V, E)$, if there exists a function $\mathbf{c} : V \to \{1, 2, 3\}$ such that for all $(u, v) \in E$, $(u) = (v)$. This problem is NP-complete [**Stockmeyer73**].

$\exists\forall$ SAT The *$\exists\forall$ SAT problem* asks, given a boolean formula over variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, if there exists a valuation of the x_i $\nu : \{x_1, \ldots, x_n\} \to \{\bot, \top\}$ such that for all valuation $\mu : \{y_1, \ldots, y_m\} \to \{\bot, \top\}$, the valuation mapping each x_i to $\nu(x_i)$ and each y_j to $\mu(y_j)$ satisfies φ. This problem is Σ_2^P-complete, even if the formula is in 3DNF [**SchaeferU02**].

Intersection of DFAs The *intersection emptiness of DFAs problem* asks, given DFAs $\mathcal{A}_1, \ldots, \mathcal{A}_n$, whether $\bigcap_{i=1}^{n} \mathcal{L}(\mathcal{A}_i)$ is empty. This problem is PSPACE-complete [**Kozen77**].

Exponential tiling The *exponential grid tiling problem* asks, given a set of *colours* C, a number N *in unary* and a set of *tiles* $T \subseteq C^{\{\mathtt{up},\mathtt{down},\mathtt{left},\mathtt{right}\}}$, whether there is a *tiling* of the $2^N \times 2^N$ grid, i.e., a function $\tau : [0, 2^N - 1] \times [0, 2^N - 1] \to T$ such that for all $x, y, x', y' \in [0, 2^N - 1]$,

- if $x = x'$ and $y = y' + 1$ then $\tau(x, y).\mathtt{down} = \tau(x, y).\mathtt{up}$
- if $x = x' + 1$ and $y = y'$ then $\tau(x, y).\mathtt{left} = \tau(x, y).\mathtt{right}$
- if $x = 0$ (resp. $x = 2^N - 1$, $y = 0$, $y = 2^N - 1$) then $\tau(x, y).\mathtt{left} = c_{border}$ (resp. `right`,`down`, `up`)

This problem is NEXPTIME-complete [**Boas19**].

Infinite Post correspondence problem The *Infinite Post correspondence problem* (IPCP) asks, given a family of pairs of words $(u_i, v_i)_{i \in I}$ over an alphabet Σ, whether there exists an infinite sequence of indices $i_0 i_1 \cdots \in I^\omega$ such that $u_{i_0} u_{i_1} \cdots = v_{i_0} v_{i_1} \cdots$

This problem is undecidable [**Ruohonen85**].

Emptiness of the intersection of two context-free grammars This problem asks, given two context-free grammars $\mathbf{G}_1$ and $\mathbf{G}_2$, if $\mathcal{L}(\mathbf{G}_1) \cap \mathcal{L}(\mathbf{G}_2) = \emptyset$. This problem is well-known to be undecidable [**HopcroftU07**].

Halting problem for Minsky machines A Minsky Machine with two counters is a tuple $M = (\text{Loc}, \Delta, \mathtt{X}, \ell_0, \ell_f)$ where

- Loc is a finite set of locations,
- $\mathtt{X} = \{\mathtt{x}_1, \mathtt{x}_2\}$ is a set of two counters,
- $\Delta \subseteq \text{Loc} \times \{\mathtt{x}{-}{-}, \mathtt{x}{+}{+}, \mathtt{x} = 0? \mid \mathtt{x} \in \mathtt{X}\} \times \text{Loc}$ is a finite set of transitions,
- $\ell_0 \in \text{Loc}$ is an initial location and $\ell_f \in \text{Loc}$ is a final location.

A *configuration* of a Minsky machine is a tuple $(\ell, v_1, v_2) \in \text{Loc} \times \mathbb{N} \times \mathbb{N}$ where v_1 (resp. v_2) stands for the value of the counter $\mathtt{x}_1$ (resp. $\mathtt{x}_2$). We write $(\ell, v_1, v_2) \to (\ell', v_1', v_2')$ if there is $\delta \in \Delta$ such that:

- $\delta = (\ell, \mathtt{x_i}{+}{+}, \ell')$ and $v_i' = v_i + 1$, $v_{3-i} = v_{3-i}'$;
- $\delta = (\ell, \mathtt{x_i}{-}{-}, \ell')$ and $v_i' = v_i - 1$, $v_{3-i} = v_{3-i}'$;
- $\delta = (\ell, \mathtt{x_i} = 0?, \ell')$ and $v_i' = v_i = 0$, $v_{3-i} = v_{3-i}'$.

An execution of the machine is a sequence $(\ell_1, v_1^{(1)}, v_2^{(1)}) \to \ldots \to (\ell_k, v_1^{(k)}, v_2^{(k)})$. The halting problem asks whether there is an initial execution of the machine ending in location ℓ_f. This problem is well-known to be undecidable [**Minsky**].

Chapter 3

Broadcast Networks of Register Automata

Il y a dans toute foule des hommes que l'on ne distingue pas, et qui sont de prodigieux messagers.

Vol de nuit
ANTOINE DE SAINT-EXUPÉRY

Contents

3.1 Introduction

3.1.1 Context and motivation

The chapter dives into the study of the computational model known as Broadcast Networks of Register Automata. These networks comprise an arbitrary number of agents, each equipped with identical transition systems. Agents communicate by broadcasting messages, which are pairs (m, d), where m is a letter from a finite alphabet and d denotes a datum drawn from an infinite set $\mathbb{D}$. Data of $\mathbb{D}$ should be thought of as identifiers. Each agent possesses local registers, all initialized with its unique identifier. Those registers are used to store and compare the data contained in messages. Consequently, each agent possesses two primary operations: broadcasting a symbol along with the content of one of its registers or receiving a message, and comparing its datum with its registers and/or storing it in them.

We aim to model networks where the communication graph may evolve through time; the set of processes receiving the messages of a given process is not fixed. In our model, this is interpreted as non-determinism: when an agent sends a message, each other agent may or may not receive it; the set of agents receiving the broadcast is non-deterministic. The most fundamental problem on such models is the coverability problem, which asks if a system has a run from an initial configuration to one where at least one agent is in a designated state.

This model was introduced in [**DelzannoST13**], as a natural extension of Reconfigurable Broadcast Networks (RBN) [**DelzannoSZ2010Adhoc**]. The key difference between the two is that in RBN the processes are anonymous. By contrast, in BNRA each process is given a unique identifier at the start. Those identifiers are manipulated by processes using registers and equality tests. That first paper claimed that the coverability problem was decidable and even PSPACE-complete, but the proof turned out to be incorrect [**ArnaudErratum**]. As we will see, the complexity of that problem is in fact much higher.

In this chapter we will establish the decidability of the coverability problem, and prove its completeness for the hyper-Ackermannian complexity class $\mathbf{F}_{\omega^\omega}$, thereby showing that the problem requires a non-primitive recursive (even non multiply-recursive) amount of time.

Some crucial aspects of the model for decidability are:

- The set of configurations we try to avoid (those in which an agent is in the error state) is upward-closed. By contrast, we show that it is undecidable to check whether we can reach a configuration where all processes are gathered in a given state 3.45.
- The messages can only contain one datum. It was shown in the paper introducing

this model that if we allow messages to carry more than one datum, the state reachability problem is undecidable [**DelzannoST13**].

- The lossy broadcast communication: We choose which processes receive each broadcast. This gives us a common property called the copycat principle, which is described below. Even without data, models which do not allow reconfigurations have been shown undecidable [**DelzannoSZ2010Adhoc**].

We contrast this result with the undecidability of the synchronisation problem, sometimes referred to as the *target reachability problem* in the literature. This problem asks whether there is a run at the end of which all agents are in a given state. It is also a classic problem on this type of models. Its undecidability highlights the crucial property of coverability, which is that the set of configurations we are trying to reach is upwards-closed. We contrast these results with the NP-completeness of the coverability problem when each agent has only one register. The aforementioned results were published in [**GuillouMW24**].

This chapter presents these results, with several improvements:

- We characterize (non-)coverability using invariants, which improve our understanding of the problem.
- We use this characterisation to prove decidability of the more general problem of safe strategy synthesis. We also characterise the complexity of those problems in terms of the number of registers per process.
- We discuss several extensions of the results, such as pushdown processes or a leader.

3.1.2 Structure of the chapter and how to read it

Section 3.2 describes our model and the problems we are interested in. We also define the decision problems that we will study in this chapter.

The central result of this chapter is the decidability of controller synthesis for broadcast networks of register automata. We present this in an incremental way:

- First, in Section 3.3 we use the simpler case of broadcast networks without data to illustrate our proof structure for decidability of coverability and strategy synthesis. The results obtained there already exist in the literature: we reprove them here in a way that illustrates our technique.
- In Section 3.4 we extend this proof structure to the subclass of systems called signature BNRA where processes can only send messages with their own identifier. This allows us to illustrate the proof technique in more details, with some idea which could not be shown in the too simple previous section. We also prove complexity lower bounds in this section, as our hardness results for the general case can already be shown for signature BNRA.
- Finally, we present the decidability proofs in the general case in Section 3.5. This part is quite technical, but we hope that the basis provided by the previous section mitigates this difficulty.

In Sections 3.3, 3.4 and 3.5 we tried to highlight the common structure by using similar sequences of definitions and lemma (although the number of intermediate steps increases).

For instance, Definition 3.11, Lemma 3.12, and Theorem 3.15 have counterparts in Section 3.4: Definition 3.19, Lemma 3.20, and Theorem 3.26.

Section 3.7 is dedicated to extensions of this model: for instance, we discuss how to adapt previous proofs to systems where processes have access to local stacks, or can test inequalities between data, with $\mathbb{D} = \mathbb{N}$.

To complete the picture, in Section 3.8 we show tight complexity bounds for the Cover and SafeStrat problems in the case where each process has a single register.

Additionally, in Appendix A we present the proof of a theorem on register automata and transducers, of independent interest. It shows a run reduction method for register automata that produces a "cheaper" run, in the sense that for each datum d the sequence of letters received with datum d is a subword of the one of the original run. This result can be used to give an alternative decidability proof of the coverability problem for BNRA.

Nb of registers / **Problem**	$r = 0$	$r = 1$	$r \geq 2$
Coverability	P [**DelzannoSZ2010Adhoc**]	NP (Thm 3.53)	$\mathbf{F}_{\omega^\omega}$ (Thm 3.43)
Synchronisation	P [**Fournier15these**]	???	Undec. (Thm 3.4
Safe strategy synthesis	NP (Thm 3.15)	NExpTime (Thm 3.55)	$\mathbf{F}_{\omega^\omega}$ (Thm 3.43)

Table 3.1: Complexity of the three considered problems depending on the number of registers. All decidable problems are complete for the indicated class. The results of the last column hold for any fixed $r \geq 2$ and when r is part of the input.

Table 3.1 contains the main decidability and complexity results of this chapter[1].

3.2 Preliminaries

In this section, we describe the transition system of individual processes, which are register transducers, a finite-state model which inputs and outputs messages using registers to store the data and equality tests to compare them. Then, in Section 3.2.2 we define our broadcast network model as the composition of a finite but arbitrary number of processes, which have identical states and transitions, but with distinct initial data in their registers. A step of computation consists in one process broadcasting a message and some subset of other processes receiving it.

3.2.1 Register transducers

In all that follows we fix an infinite set of *data* $\mathbb{D}$. We define a notion of register transducer that is well-suited for the definition of our distributed systems.

[1]It is always a little frustrating to leave a gap in a table like this. However, I convinced myself that an algorithm for the synchronisation problem could only be very long and painful. Since this chapter already contains some pretty technical proofs, we will skip this case and leave it as an open problem.

A register transducer describes the behaviour of an agent in a broadcast network with data. There are two types of transitions: reception transitions receive messages, test equality of the datum against the ones in the register, and put the received datum in some registers. Broadcast transitions send a message with the content of a register as datum.

Definition 3.1 ▶ Register transducer

A *register transducer* with r registers over domain $\mathbb{D}$ is given by a tuple $\mathcal{R} = (Q, \mathcal{M}, q_{init}, \Delta)$ with

- Q a finite set of states, with q_{init} the initial state,
- $\mathcal{M}$ a finite alphabet, and
- Δ a set of transitions which are of four kinds:
  - $q \xrightarrow{\mathbf{br}(m,i)} q'$ *broadcast transitions* that output a message (m, d) with d the content of register i,
  - $q \xrightarrow{\mathbf{rec}(m,=i)} q'$ *equality transitions* that read a message (m, d) and check that d is in register i,
  - $q \xrightarrow{\mathbf{rec}(m,\neq)} q'$ *disequality transitions* that read a message (m, d) and check that d is not in any register,
  - $q \xrightarrow{\mathbf{rec}(m,\downarrow i)} q'$ *record transitions* that read a message (m, d) where d is not in any register and put d in register i.

  Transitions of the three last kinds are called *reception transitions*.

The *size* of $\mathcal{R}$, written $|\mathcal{R}|$, is $|Q| + |\Delta| + r$

Formally, a *local configuration* of $\mathcal{R}$ is an element of $Q \times \mathbb{D}^r$, describing the current state and the content of each register. A local configuration (q, c) is *initial* if $q = q_{init}$ and all registers have the same content, i.e., there exists $d \in \mathbb{D}$ such that $c(i) = d$ for all $i \in [1, r]$.

Let (q, c) and (q', c') be two local configurations, we formalise the conditions required to go from one to the other.

Given a record transition $q \xrightarrow{\mathbf{rec}(m,\downarrow i)} q'$ and a datum d , we can apply δ to go from (q, c) to (q', c') by reading (m, d) if

- for all $j \in [1, r]$, $c(j) \neq d$,
- $c'(i) = d$
- for all $j \neq i$, $c'(j) = c(j)$.

Given an equality transition $q \xrightarrow{\mathbf{rec}(m,=i)} q'$ and a datum d , we can apply δ to go from (q, c) to (q', c') by reading (m, d) if $c(i) = d$ and $c' = c$.

Given a disequality transition $q \xrightarrow{\mathbf{rec}(m,\neq)} q'$ and a datum d , we can apply δ to go from (q, c) to (q', c') by reading (m, d) if $c(j) \neq d$ for all $j \in [1, r]$ and $c' = c$.

If one of the three previous cases applies, we write $(q, c) \xrightarrow{\mathbf{rec}(m,d)}_{\delta} (q', c')$ and call it a *reception step*.

Given a broadcast transition $\delta = q \xrightarrow{\mathbf{br}(m,i)} q'$ and a datum d, we can apply δ to go from (q, c) to (q', c') by broadcasting (m, d) if $c(i) = d$ and $c' = c$. If those conditions are met we write $(q, c) \xrightarrow{\mathbf{br}(m,d)}_{\delta} (q', c')$ and call it a *broadcast step*[2].

A *local step* $(q, c) \xrightarrow{\mathbf{op}(m,d)}_{\delta} (q', c')$ between two local configurations is either a reception step or a broadcast step. A *local run* u of $\mathcal{R}$ is a sequence of local steps $u = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} (q_1, c_1) \xrightarrow{\mathbf{op}_2(m_2,d_2)}_{\delta_2} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$. Its *length* is its number of local steps n. It is *initial* if (q_0, c_0) is an initial configuration. In that case the common datum d of registers in c_0 is called is called the *initial datum* of u.

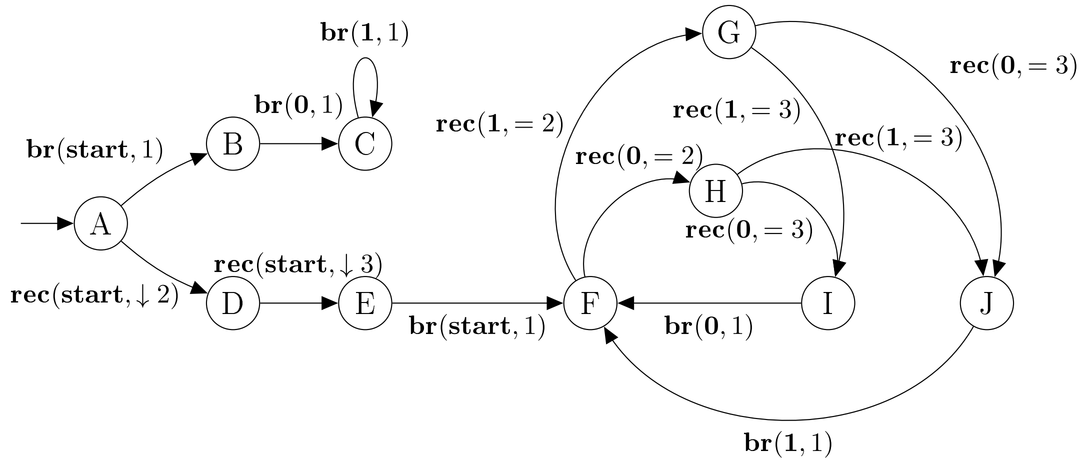


Figure 3.1: An example of a register transducer.

Example 3.2.1. *Have a look at the register transducer in Figure 3.1. In the upper branch it broadcasts a word of* **start01*** *with the initial datum. In the lower branch it receives* **start** *with two different data that it stores in registers 2 and 3, and broadcasts* **start** *with its initial datum. It then receives alternately one bit with the first datum and one bit with the second, and broadcasts their xor, still with its initial datum.*

Its *input* $\mathbf{In}(u) \in (\mathcal{M} \times \mathbb{D})^*$ is the sequence of messages received by input transitions $(q_{i-1}, c_{i-1}) \xrightarrow{\mathbf{rec}(m_i,d_i)}_{\delta_i} (q_i, c_i)$ in u. Similarly, its *output* $\mathbf{Out}(u) \in (\mathcal{M} \times \mathbb{D})^*$ is the sequence of messages sent by output transitions $(q_{i-1}, c_{i-1}) \xrightarrow{\mathbf{br}(m_i,d_i)}_{\delta_i} (q_i, c_i)$ in u.

Its *d-input* $\mathbf{In}_d(u) \in \mathcal{M}^*$ is the sequence of letters associated to datum d in $\mathbf{In}(u)$, and its *d-output* $\mathbf{Out}_d(u) \in \mathcal{M}^*$ is the sequence of letters associated to datum d in $\mathbf{Out}(u)$.

Remark 3.2.1. *Note that we require that record transitions can only be taken if the received value is not already in the registers. This is not a restriction on the expressivity as we can simulate a system with weak record transitions (where we can store values that are already in another register) with those operations: Instead of storing the same datum in several registers, the simulating system uses a function* $[1, r] \to [1, r]$*, stored in*

[2]The **br** notation may look odd for now, but it will be justified later when those automata are used to model processes broadcasting and receiving messages.

the states, that maps registers of the simulated systems to the register holding the same datum in the simulating system.

On the other hand, this requirement will facilitate some proofs: this guarantees that each datum appears in at most one register (except the initial datum).

3.2.2 Broadcast Networks with Data

Let $r \in \mathbb{N}$ and let $\mathcal{R} = (Q, \mathcal{M}, q_{init}, \Delta)$ be a register transducer with r registers. The *broadcast network of register automata* (BNRA for short) described by $\mathcal{R}$ is the infinite transition system described below. We call register transducer *protocols* when we use them to define BNRA. We often identify a BNRA and the protocol describing it.

A *configuration* is a function $\gamma : \mathbb{A} \to Q \times \mathbb{D}^{\mathsf{r}}$ with $\mathbb{A}$ a finite set of *agents*. It maps each agent to a local configuration.

We write $\mathsf{st}(\gamma)$ for the state component of γ and $\mathsf{data}(\gamma)$ for its register component. A configuration γ is *initial* if for all $a \in \mathbb{A}$, $\mathsf{st}(\gamma)(a) = q_{init}$, $\mathsf{data}(\gamma)(a,i) = \mathsf{data}(\gamma)(a,i')$ for all i, i' and $\mathsf{data}(\gamma)(a,i) \neq \mathsf{data}(\gamma)(a',i')$ for all $a \neq a'$ and i, i'. Intuitively, each agents starts with a unique identifier that is contained in all of its registers.

Given a finite set of agents $\mathbb{A}$ and two configurations γ, γ' over $\mathbb{A}$, a *step* $\gamma \to \gamma'$ is defined when there exist $a_0 \in \mathbb{A}$, $m \in \mathcal{M}$, $d \in \mathbb{D}$ and a transition $\delta_0 \in \Delta_O$ such that $\gamma(a_0) \xrightarrow{\mathbf{br}(m,d)}_\delta \gamma'(a_0)$, and for all $a \neq a_0$,

- either $\gamma'(a) = \gamma(a)$,
- or there is a transition $\delta \in \Delta_I$ such that $\gamma(a) \xrightarrow{\mathbf{rec}(m,d)}_\delta \gamma'(a)$.

A (global) *run* ϱ consists of a sequence of steps $\gamma_0 \to \gamma_1 \to \gamma_2 \cdots \gamma_{n-1} \to \gamma_n$. It is *initial* if γ_0 is an initial configuration. The *projection* of ϱ on an agent $a \in \mathbb{A}$ is the local run $\pi_a(\varrho)$ made of all transitions taken by a in ϱ. We write $\varrho : \gamma \xrightarrow{*} \gamma'$ when ϱ is a run starting in γ and ending in γ'.

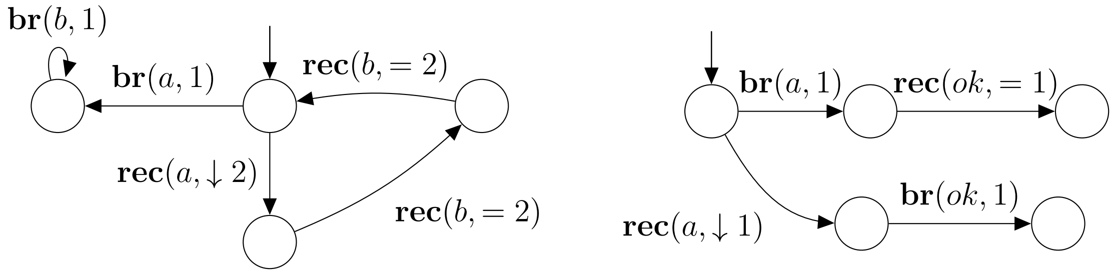


Figure 3.2: Two representative behaviours: on the left, a protocol where an agent can receive a sequence of letters *abb* while checking that a the messages all come from the same agent. On the right, a protocol which can receive a message from another agent and acknowledge reception by sending a message with the other agent's identifier.

We will study two verification problems on these systems. One asks whether there is a run in which some given message is broadcast (or, equivalently, some given state is reached). The other asks for a run leading to a configuration with all agents in a given target state.

Definition 3.2 ▶ Verification problems

The *coverability problem* COVER asks, given a protocol $\mathcal{R}$ and a state q_{err}, whether there is an initial run of $\mathcal{R}$ in which at least one agent reaches q_{err}.
The *synchronisation problem*[a] SYNCHRO asks, given a protocol $\mathcal{R}$ and a state q_f, whether there is an initial run of $\mathcal{R}$ such that all agents are on q_f in the last configuration.

[a]sometimes called the target reachability problem or target problem

If there is an initial run ϱ in which an agent reaches q_{err}, then we say that ϱ *covers* q_{err}, and that q_{err} is *coverable*. Similarly, if there is an initial run ϱ in which a message with letter m_{err} is broadcast, then we say that ϱ covers m_{err}, and that m_{err} is coverable.

Remark 3.2.2. *It will often be convenient to consider a version of the coverability problem in which a letter m_{err} is given instead of q_{err}. The question is then whether there is an initial run in which m_{err} is broadcast.*

The two problems are easily reducible to each other: in one direction it suffices to add a loop broadcasting a message m_{err} on q_{err}, in the other it suffices to redirect every transition broadcasting m_{err} towards a new state q_{err}.

We will also examine the controller synthesis problem on this model. We assume that We assume that a subset of states within the protocol is controllable, and we consider *local strategies*, which select transitions from those controllable states based on the sequence of transitions executed so far in the local run. The question is whether there is a strategy that guarantees that a designated state is never reached in a global run. The COVER problem defined earlier corresponds to the particular case in which no state is controllable.

Definition 3.3 ▶ Broadcast Game with Registers

A *Broadcast Game with Registers* with r registers $\mathcal{G} = (\mathcal{R}, Q_{ctrl}, Q_{env}, q_{err})$ is defined by a register transducer with r registers $\mathcal{R} = (Q, \mathcal{M}, q_{init}, \Delta)$, a partition of its states $Q = Q_{\mathsf{ctrl}} \sqcup Q_{\mathsf{env}}$, and an error state q_{err}. As noted in Remark 3.2.2, we can replace q_{err} with a letter m_{err} that should not be broadcast.

A *control strategy* for $\mathcal{G}$ is a function $\sigma : \Delta^* \to \Delta$ observing a sequence of transitions and choosing the next one[3].

A *σ-local run* is an initial local run $u = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$ such that for all $i \in [1, n]$, if $q_{i-1} \in Q_{\mathsf{ctrl}}$ then $\sigma(\delta_1 \cdots \delta_{i-1}) = \delta_i$. A *$\sigma$-run* is an initial run whose projection on every agent a is a σ-local run.

A control strategy is *winning* if no σ-run covers q_{err}. In this game and all games we construct from it the player trying to construct a winning control strategy will be called Controller and her opponent Environment.

Definition 3.4 ▶ Controller synthesis problem

The *safe strategy problem* SAFESTRAT takes as input a BGR $\mathcal{G}$, and asks whether there is a winning control strategy for $\mathcal{G}$.

[3]We choose to not give access to the data to Controller, as we want to be able to rename data at will. We will discuss the version of the game where Controller can see the data in Section 3.7.1.

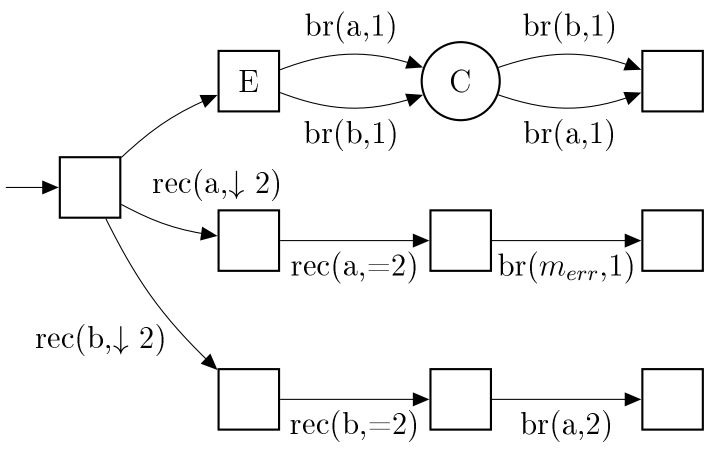


Figure 3.3: An example of BGR. The round state C belongs to Controller, square states to Environment.

Example 3.2.2. *In the BGR displayed in Figure 3.3, Controller has a single winning control strategy, which is to always choose a different letter from the one chosen by Environment from E.*

Indeed, doing otherwise would let an agent broadcast either aa or bb with its initial datum. In the first case, Environment could send an agent in the second row to receive aa and then broadcast m_{err}. In the second case, Environment could send two agents to the third row, who receive bb and broadcast a with the same datum. An agent in the second row could then receive both a broadcasts and then broadcast m_{err}.

By contrast, it is easy to check that if Controller always picks a different letter from the one chosen by Environment in E, there cannot be two broadcasts of a or of b with the same datum. Hence agents sent to the second and third row will be unable to broadcast anything, and in particular m_{err} will never be broadcast.

3.2.3 Well quasi-order toolbox

In this chapter we will rely several times on the theory of well quasi-orders, more precisely the well quasi-order of subwords over a finite alphabet. This section is meant as a summary of the results we will use.

A *well quasi-order* is a set equipped with a preorder relation $(S, \preceq)$ such that in every infinite sequence $s_0, s_1, \ldots$ there exist $i < j$ such that $s_i \preceq s_j$.

Subwords

Given two words $v = a_1 \cdots a_m$ and $w = b_1 \cdots b_n$ in Σ^*, we say that v is a *subword* of w and write $v \sqsubseteq w$ if v can be obtained from w by removing letters, i.e., there are indices $i_1 < \cdots i_m$ such that $v = b_{i_1} \cdots b_{i_m}$.

Given a set of words W, we define its *upward-closure* $W\uparrow = \{u \in \Sigma^* \mid \exists w \in W, w \sqsubseteq u\}$ and its *downward-closure* similarly $W\downarrow = \{u \in \Sigma^* \mid \exists w \in W, u \sqsubseteq w\}$.

A set of words $I \subseteq \Sigma^*$ is called *downward-closed* if it is closed under taking subwords, i.e., $I = I\downarrow$. Similarly, I is *upward-closed* if $I = I\uparrow$. Note that the complement of a downward-closed (resp. upward-closed) set is upward-closed (resp. downward-closed). The set of minimal elements of an upward-closed set I is called its *basis*. Clearly $\sqsubseteq$ does not have infinite decreasing sequences, from which we can infer that if B is the basis of

I, we have $I = B\uparrow$. Given a finite basis B, we define its *norm* as the maximum length of its words: $||B|| = \max\{|w| \mid w \in B\}$.

Example 3.2.3. *The word abba is a subword of bbaababaa.*

The downward-closure of the set $\{a^n b^n \mid n \in \mathbb{N}\}$ *is* $a^* b^*$*, while the upward-closure of* $\{aaa, aba\}$ *is* $b^* a(a+b)^+ ab^*$*. The set of words that contain at least an* a *and a* b *is upward-closed, and its basis is* $\{ab, ba\}$*.*

A seminal result in the study of well quasi-orders is Higman's lemma. It was discovered independently several times, and is now a common tool in formal verification.

Lemma 3.5 ▶ *Higman's lemma* [Higman52; Haines69]

Let Σ be a finite alphabet. Then $(\Sigma^*, \sqsubseteq)$ is a well quasi-order.

An important corollary (or even reformulation) of this result is

Corollary 3.6 ▶ [Higman52; Haines69]

Let Σ be a finite alphabet. Every upward-closed set of words $I \subseteq \Sigma^*$ has a finite basis.

This property will be crucial in the next sections.

Another corollary is that the upwards-closure of any language over a finite alphabet is regular. This is a consequence of Corollary 3.6, along with the following fact.

Lemma 3.7 ▶ Folklore

Given a finite set of words B over a finite alphabet Σ, one can construct a deterministic automaton $\mathcal{A}_{B\uparrow}$ recognising $B\uparrow$ with at most $(||B||+1)^{|B|}$ states.

This automaton can be constructed simply by using states to keep track of the maximal prefix of each element of B that is a subword of the word read so far. Initially, this is ε for all words, we accept when one of the words has been fully seen. The transitions are inferred easily.

Lossy channel systems [AbdullaJ93; Finkel94]

We now present Lossy Channel Systems (LCS), which will help us illustrate the interest of well quasi-orders, and which we will later use for lower bounds.

A *lossy channel system* with a single channel is a finite-state machine that has the ability to buffer symbols in a lossy FIFO queue [**Schnoebelen2002verifying**].

Formally, it is a tuple $\mathcal{S} := (L, \Sigma, T, l_{init})$, where L is a finite set of locations[4], Σ is a finite set of symbols, $T \subseteq L \times \{\mathsf{read}(x), \mathsf{write}(x) \mid x \in \Sigma\} \times L$ is the set of transitions and $l_{init} \in L$ an initial state. The action $\mathsf{write}(x)$ corresponds to writing x at the end of the channel and $\mathsf{read}(x)$ to reading x at the beginning the channel. A configuration of $\mathcal{S}$ is a pair in $L \times \Sigma^*$ denoting the location and the content of the channel. There is a step from (l, w) to (l', w') using $t \in T$, denoted $(l, w) \overset{t}{\rightsquigarrow}_{\mathcal{S}} (l', w')$, when

[4]We call them locations and not states to avoid confusion with the states of other systems in the proofs.

- $t = (l, \mathsf{write}(x), l')$ for some $x \in \Sigma$ and $w' \sqsubseteq w \cdot x$,
- $t = (l, \mathsf{read}(x), l')$ for some $x \in \Sigma$ and $x \cdot w' \sqsubseteq w$.

The existence of a step is denoted $(l, w) \rightsquigarrow_{\mathcal{S}} (l', w')$, and its reflexive-transitive closure is denoted $\overset{*}{\rightsquigarrow}_{\mathcal{S}}$. The (location) *reachability problem* asks, given an LCS $\mathcal{S}$ and a location $l_f \in L$, whether $(l_{init}, \varepsilon) \overset{*}{\rightsquigarrow}_{\mathcal{S}} (l_f, w)$ for some w.

Remark 3.2.3. *Note that if we have $(l, w_1) \rightsquigarrow_{\mathcal{S}} (l', w_1')$ and $w_1 \sqsubseteq w_2$ then there exists w_2' such that $w_1' \sqsubseteq w_2'$ and $(l, w_2) \rightsquigarrow_{\mathcal{S}} (l', w_2')$. Transition systems with this property are called* well-structured*, and have been studied extensively. See [***Finkel16***] for a survey.*

A finite or infinite sequence of words $w_0, w_1, \dots$ is called *good* if there exist $i < j$ such that $w_i \sqsubseteq w_j$, and *bad* otherwise. Higman's lemma [**Higman52**] states that every bad sequence of words over a finite alphabet is finite, but there is no uniform bound: bad sequences of words can be arbitrarily long: take, for instance, the sequence $a^N, a^{N-1}, \dots$ for any $N \in \mathbb{N}$.

However, if we add a constraint so that each word can only have finitely many successors, then a uniform bound exists. Suppose we have a function $g : \mathbb{N} \to \mathbb{N}$ and only consider bad sequences of words $w_0, w_1, \dots$ such that $|w_{i+1}| \leq g(|w_i|)$. Given an initial word w_{init}, the set of such sequences starting with w_{init} can be described as a tree. As all bad sequences are finite, this tree only has finite branches. As each word w has less than $|\Sigma|^{g(|w|)+1}$ successors, each node of the tree has finitely many successors. Hence the tree is finite by König's lemma, and its depth bounds the length of those sequences.

In the next part we will see the Length Function Theorem, which bounds the growth of the function mapping the size of the root w_{init} to the maximal depth of such a tree, assuming g is primitive recursive.

Remark 3.2.3 implies that if there is a run $(l_0, w_0) \rightsquigarrow_{\mathcal{S}} \cdots \rightsquigarrow_{\mathcal{S}} (l_n, w_n)$ reaching a location $l = l_n$ then there is one such that $l_0 w_0, \cdots, l_n w_n$ is a bad sequence over $L \cup \Sigma$.

Furthermore, we are only interested in runs that start at the initial configuration (l_s, ε), and we know that if $(l, w) \rightsquigarrow_{\mathcal{S}} (l', w')$ then $|w'| \leq |w| + 1$.

Those observations allow us to decide the location reachability problem by guessing a run yielding a bad sequence of words and checking if it reaches l_f.

Theorem 3.8 ▶ [AbdullaJ93]

Location reachability is decidable for lossy channel systems.

Length Function Theorem

For α an ordinal in Cantor normal form, we denote by $\mathscr{F}_\alpha$ the class of functions corresponding to level α in the Fast-Growing Hierarchy. We denote by $\mathbf{F}_\alpha$ the associated complexity class and use the notion of $\mathbf{F}_\alpha$-completeness. All these notions are defined in [**Schmitz16**]. We will specifically work with complexity class $\mathbf{F}_{\omega^\omega}$. For readers unfamiliar with these notions, $\mathbf{F}_{\omega^\omega}$-complete problems are decidable but with very high complexity (non-primitive recursive, and even much higher than the Ackermann class $\mathbf{F}_\omega$). We do not formally define this class here, as it requires many notions that we will not use later. We will only use the two theorems below, for the upper and lower bounds respectively.

Given a function $g : \mathbb{N} \to \mathbb{N}$ and an integer $n \in \mathbb{N}$, we say that a sequence of words $w_1, \dots$ is *(g, n)-controlled* if $|w_i| \leq g^{(i)}(n)$ for all $i \geq 1$ (where $g^{(i)}$ denotes g applied

i times). We will use the following result, known as the Length Function Theorem [**SchmitzS2011upperHigman**]:

Theorem 3.9 ▶ *Length Function Theorem* [SchmitzS2011upperHigman]
Let Σ be a finite alphabet and $g : \mathbb{N} \to \mathbb{N}$ a primitive recursive function. There exists a function $f \in \mathscr{F}_{\omega^{|\Sigma|-1}}$ such that, for all $n \in \mathbb{N}$, every (g, n)-controlled bad sequence $w_1, w_2, \ldots$ has at most $f(n)$ terms.

This theorem lets us bound the maximal length of a minimal run covering a location l of an LCS by a function of $\mathscr{F}_{\omega^\omega}$. As a consequence, the location reachability problem for LCS is in $\mathbf{F}_{\omega^\omega}$. In [**ChambartS08ordinal**], Chambart and Schoebelen showed a lower bound on the length of a minimal run reaching a given location, from which we can infer the following result:

Theorem 3.10 ▶ [CichonTB98; ChambartS08ordinal]
Location reachability is $\mathbf{F}_{\omega^\omega}$-complete for lossy channel systems.

3.3 An introductory case: Broadcast networks without data

We start by showing the proof principles in an easy case, when processes do not have registers. In that case communication is made only through letters of $\mathcal{M}$. In this section we will forget the data in messages, and only consider letters. We simplify notations: we write $\mathbf{br}(m)$ for a broadcast of letter m and $\mathbf{rec}(m)$ for a reception of m. We obtain *Reconfigurable Broadcast Networks*, as introduced in [**DelzannoSZ2010Adhoc**]. In all that follows we will thus use the term RBN for this model.

Cover and Synchro are already known to be decidable in polynomial time for those systems [**DelzannoSZ2010Adhoc**; **Fournier15these**]. I could not find a reference stating that SafeStrat is NP-complete in the literature, but closely-related results were proven in [**BertrandFS15**] and [**Stan17**]. We reprove the Cover and SafeStrat upper bounds in a way that illustrates the proof techniques used in the next parts.

Remark 3.3.1. *Formally, BNRA without registers are not well-defined: we need data to put in the messages. However, we can encode them in the model by considering protocols with no record transitions. Hence all agents broadcast only with their initial data, and cannot store each other's data. Equality transitions become pointless and can be removed. The resulting systems can be seen as RBN, where messages carry no data.*

To begin with, we show that we can characterise winning control strategies as the ones which force the set of letters sent to stay within some set $I \subseteq \mathcal{M} \setminus \{m_{err}\}$, called an *invariant*.

This lets us turn the distributed game into a sequential one: If we are given an invariant I, checking whether there is a strategy that maintains it comes down to a two-player safety game. We obtain an algorithm for strategy synthesis: guess an invariant, and then solve the resulting safety game, which can be done in polynomial time.

Definition 3.11 ▶ Invariants for RBN

An *invariant* for an RBN over alphabet $\mathcal{M}$ is a set of letters $I \subseteq \mathcal{M}$. We say that it is *sufficient* for a control strategy σ if it satisfies the following conditions:

- $m_{err} \notin I$
- If a σ-local run receives only messages of I then it broadcasts only messages of I.

Lemma 3.12 ▶ Invariants characterise winning control strategies

A control strategy σ is winning if and only if there exists a sufficient invariant $I \subseteq \mathcal{M}$ for it.

Proof. We prove the two implications, starting by the left-to-right one.

$\Longrightarrow$ Suppose σ is winning. Let I be the set of messages such that there exists a σ-run in which they are broadcast. As σ is winning, m_{err} can never be broadcast, thus $m_{err} \notin I$.

We now prove the second item. Suppose by contradiction that we have a σ-local run $s_0 \xrightarrow{\mathbf{op}_1(m_1)}_{\delta_1} s_1 \xrightarrow{\mathbf{op}_2(m_2)}_{\delta_2} \cdots \xrightarrow{\mathbf{op}_k(m_k)}_{\delta_k} s_k$ with $s_0 = s_{init}$ broadcasting some $m_{out} \notin I$ and only receiving messages of I.

Then we can construct a σ-run in which m_{out} is broadcast.

We proceed by induction: for all $i \in [0, k]$, we show that there is a run ϱ_i whose projection on some agent a is $s_0 \xrightarrow{\mathbf{op}_1(m_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_i(m_i)}_{\delta_i} s_i$. For $i = 0$ this is immediate. Let $i > 0$, suppose we have constructed ϱ_{i-1}. We construct ϱ_i as follows. Let $\mathbb{A}_{i-1}$ be the set of agents of ϱ_{i-1}.

- If $s_{i-1} \xrightarrow{\mathbf{op}_i(m_i)}_{\delta_i} s_i$ is a broadcast step, then we simply execute ϱ_{i-1} and then make a apply that broadcast, which no other agent receives.
- If $s_{i-1} \xrightarrow{\mathbf{op}_i(m_i)}_{\delta_i} s_i$ is a reception step, in which a message type m is received, then we have $m \in I$, by construction of the σ-local run. Hence there exists a σ-run ϱ_m over a set of agents $\mathbb{A}_m$ in which m is broadcast. Up to renaming agents, we can assume that $\mathbb{A}_{i-1}$ and $\mathbb{A}_m$ are disjoint. We then define ϱ_i over $\mathbb{A}_{i-1} \sqcup \mathbb{A}_m$ by executing ϱ_{i-1} over $\mathbb{A}_{i-1}$, then executing ϱ_m over $\mathbb{A}_m$ up to the point before an agent a_m broadcasts m. Finally, we make a_m broadcast m and a receive it.

In both cases we obtain a σ-run in which the local run of a is $s_0 \xrightarrow{\mathbf{op}_1(m_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_i(m_i)}_{\delta_i} s_i$. In particular, for $i = k$, we get a σ-run in which m_{out} is broadcast. As $m_{out} \notin I$, this contradicts the definition of I. Hence I satisfies both items of the lemma.

$\Longleftarrow$ Suppose there exists $I \subseteq \mathcal{M}$ satisfying the conditions of the lemma. Suppose by contradiction that there is a σ-run ϱ in which m_{err} is broadcast. Let m be the first message broadcast in ϱ that is not in I, and a the agent broadcasting it. Those are well-defined as $m_{err} \notin I$. Let ϱ' be the prefix of ϱ stopping right after that broadcast. Then the projection $\pi_a(\varrho')$ of ϱ' on a contains a broadcast of m but no reception of any message $m' \notin I$, a contradiction.

We have shown both directions, concluding the proof. $\square$

Corollary 3.13 ▶ Characterisation of coverability

A message m_{err} is ***not*** coverable if and only if there exists $I \subseteq \mathcal{M}$ such that:

1. $m_{err} \notin I$
2. If a local run receives only messages of I then it broadcasts only messages of I.

Proof. A BNRA can be seen as a BGR where all states belong to Environment. We can thus simply apply Lemma 3.12. □

Proposition 3.14 ▶ [DelzannoSZ2010Adhoc]

Coverability is decidable in polynomial time for BNRA without registers.

Proof. We apply the algorithm described in Algorithm 1.

Algorithm 1 Algorithm for the proof of Proposition 3.14.

```
$J \leftarrow \emptyset$
repeat
    complete ← True
    for $m \in \mathcal{M} \setminus J$ do
        if $\exists$ an initial local run broadcasting $m$ and only receiving messages of $J$ then
            $J \leftarrow J \cup \{m\}$
            complete ← False
        end if
    end for
until complete
return $m_{err} \in J$
```

At each iteration of the while loop we either increase J or exit the loop. As a consequence we make at most $|\mathcal{M}|$ iterations. Furthermore, each iteration takes polynomial time: we can check whether there is an initial run broadcasting m and only receiving messages of J by removing receptions of messages outside of J from the protocol and then looking for a path from the initial state to a transition broadcasting m. Thus the algorithm terminates in polynomial time.

For the correctness, first observe that we can apply Lemma 3.12. Observe that the while loop maintains the invariant that all sets $I \subseteq \mathcal{M}$ satisfying the second condition of Corollary 3.13 must contain J. It trivially holds when $J = \emptyset$. If J satisfies that invariant and there is an initial local run broadcasting m and only receiving messages of J, then by the second item of Corollary 3.13 all such I must also contain m. Hence the invariant is maintained.

In the end, if J contains m_{err} then there cannot be any I satisfying the conditions of Corollary 3.13. On the other hand, if J does not contain m_{err}, then it satisfies those conditions and thus m_{err} is not coverable.

The algorithm is therefore correct. □

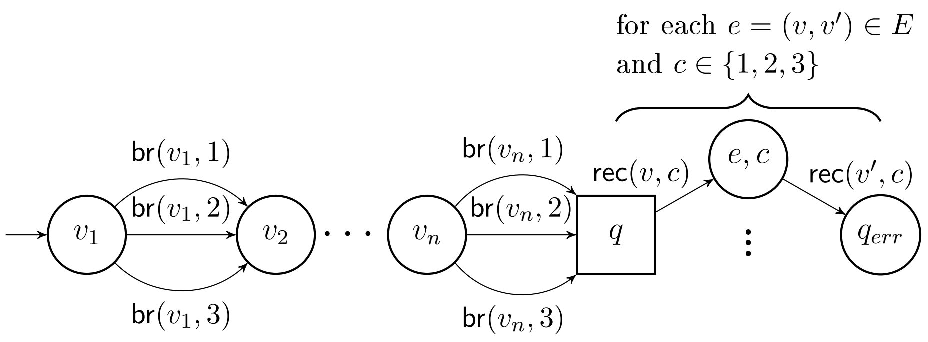


Figure 3.4: Illustration of the lower bound proof from Theorem 3.15.

The result is not new, but the proof is not quite the one presented in [**DelzannoSZ2010Adhoc**]. This version foreshadows the more general decidability proof for BNRA presented in the next sections.

We can also use this invariant characterisation to recover a result of Bertrand, Fournier and Sangnier, which is the NP-completeness of SAFESTRAT on BGR without registers.

Theorem 3.15

Deciding the winner of a BGR without registers is NP-complete.

Proof. For the upper bound, by Lemma 3.12, it suffices to guess a set $I \subseteq \Sigma$ such that $m_{err} \notin I$ and then check if there is a strategy that guarantees that we can only broadcast a message outside of I if we received one beforehand. This is easily encoded into a safety game: Take the states and transitions of the BGR, without the operations, add a sink state with no outgoing transitions, and redirect every reception of a message $m \notin I$ to it. The objective of the first player is to avoid transitions broadcasting letters of $\Sigma \setminus I$.

It is clear that there is a winning control strategy for the BGR if and only if there is an invariant I and a strategy avoiding transitions broadcasting messages outside of I in this safety game. This can be checked in polynomial time, by Proposition 2.8.

For the lower bound, we reduce from the graph 3-colouring problem.

Consider an undirected graph $G = (V, E)$ with $V = \{v_1, \dots, v_n\}$. We build a BGR with no registers as in Figure 3.4. From the initial state Controller chooses one of $(v_i, 1), (v_i, 2), (v_i, 3)$ for each $i \in [1, n]$ and broadcasts it. Then Environment picks an edge $e = (v, v') \in E$ and $c \in \{1, 2, 3\}$ and tries to reach q_{err} by receiving (v, c) and (v', c).

A strategy for Controller comes down to a colouring of V. It is winning if Environment can find an edge e and c such that both ends of e are coloured with c. In other words, Controller wins if and only if the selected colouring of V is a valid 3-colouring of G. In conclusion, we have a reduction from the graph 3-colouring problem to SAFESTRAT for BGR without registers. □

To conclude this section, let us present an argument in favour of *parameterized* distributed synthesis. The fact that we consider an arbitrary amount of agents makes the existence of a winning control strategy less likely. One might wonder what happens if we simply want a strategy that works for a bounded, or even fixed amount of agents. We show that the problem becomes undecidable in this case, even for 3 agents. This indicates that considering arbitrary numbers of agents can be a good approximation as it spectacularly reduces the difficulty of the problem.

Theorem 3.16

Given a BGR $\mathcal{G} = (\mathcal{R}, Q_{\mathsf{ctrl}}, Q_{\mathsf{env}}, q_{err})$, it is undecidable whether there is a control strategy such that no σ-run *with 3 agents* covers q_{err}.

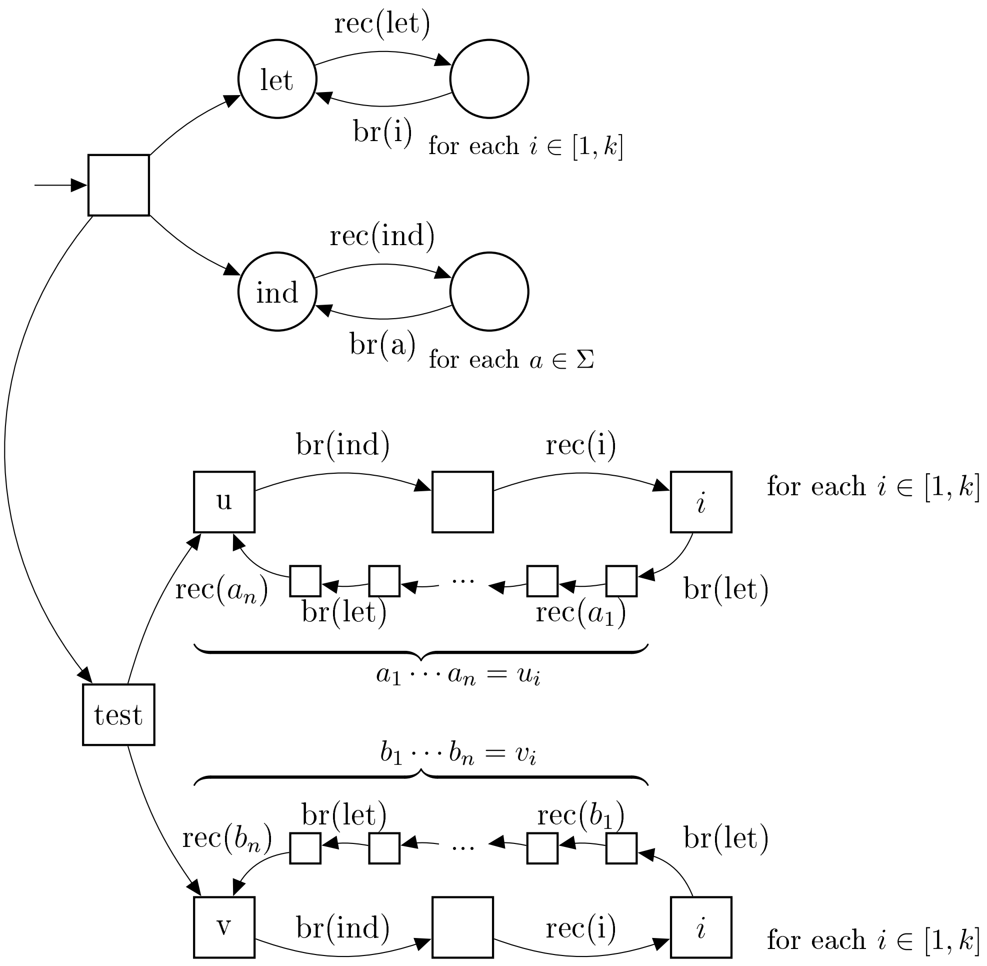


Figure 3.5: System used for the proof of Theorem 3.16.

Proof. We adapt the classic reduction from PCP from [**PnuRos89**].

Let $(u_i, v_i)_{i \in [1,k]}$ be an Infinite PCP instance over an alphabet Σ.

Consider the protocol displayed in Figure 3.5. From the initial state Environment decides to go to either let, ind or test.

From state let (resp. ind), Controller chooses and broadcasts an infinite sequence of letters of Σ (resp. indices of $[1, k]$). Before each broadcast the agent must receive a message *let* (resp. *ind*).

From state test, Environment chooses to either test the (u_i) or (v_i). To check them, he alternates between a reception of an index i and a sequence of letters that form the word u_i. Before receiving a letter (resp. index) he broadcasts *let* (resp. *ind*). If he receives a letter that does not match the current word, he goes to q_{err}.

A strategy σ comes down to a choice of two sequences $i_0 i_1 \cdots$ and $a_0 a_1 \cdots$, which are the sequences of broadcasts made by Controller from states let and ind.

It must be the case that $a_0 a_1 \cdots = u_{i_0} u_{i_1} \cdots$ as otherwise Environment can send an agent in let, another in ind and the third one in u. Then eventually the sequence of indices will not match the sequence of letters and the third agent will reach q_{err}. Similarly, it must be the case that $a_0 a_1 \cdots = v_{i_0} v_{i_1} \cdots$ Hence a winning strategy must yield a solution to the Infinite PCP instance.

Conversely, if Controller broadcasts a solution to the Infinite PCP instance, then Environment cannot reach q_{err}: he has to send an agent in test to reach it, and an agent in let and another one in ind to make broadcasts. Note that agents all alternate between a broadcast and a reception. Further, at all times there is at most one agent who can make a broadcast, and one agent who can receive the message (no message can be received by two agents). If at some point a message is not received, the system is stuck with all agents waiting for a reception, and thus Controller wins. Otherwise, since all messages are received and the broadcasts from let and ind match on indices and letters, no agent can reach q_{err}.

In conclusion, the Infinite PCP instance has a solution if and only if Controller has a control strategy such that no σ-run with 3 agents covers q_{err}. This establishes the undecidability of the problem. □

3.4 Signature BGR

In this section we establish decidability of COVER and SAFESTRAT in a subcase of interest, that illustrates well the decidability proof for the general case, while requiring less technical complications.

Definition 3.17

A *signature protocol* is one where every broadcast is made with the value of register 1, and all receptions are made on other registers.

In other words, there are no transitions of the form $\xrightarrow{\mathbf{br}(m,i)}$ with $i \geq 2$ or $\xrightarrow{\mathbf{rec}(m,\downarrow 1)}$ or $\xrightarrow{\mathbf{rec}(m,=1)}$.

We call *signature BNRA* and *signature BGR* the BNRA and BGR described by signature protocols.

In other words, such a protocol keeps its initial datum in register 1 and uses it for broadcasts, while the other registers are used to store and compare received values. As processes only broadcast with their initial datum, in this section we will call *output* the d-output of a local run u with d its initial datum, and write it $\mathbf{Out}_{sign}(u)$.

An interesting property of those systems is that the datum of a message identifies its sender: Each agent only sends messages with its initial datum, and since those are unique, messages containing the same datum necessarily come from the same agent.

In this section we prove the following theorem:

Theorem 3.18

The COVER and SAFESTRAT problem are decidable and in $\mathbf{F}_{\omega^\omega}$ for signature BGR.

Let us fix $\mathcal{G} = (\mathcal{R}, Q_{\mathsf{ctrl}}, Q_{\mathsf{env}}, m_{err})$ a BGR with r registers.

To prove the theorem, we once again use a characterisation of winning strategies in terms of invariants. Here an invariant is a (downward-closed) set of words of $\mathcal{M}^*$, instead of letters like in the previous section. A witness for non-coverability of a message

m_{err} is a downward-closed set of words that contains ε and not m_{err} and such that an agent following whose output is not in that set has a d-input outside that set as well[5]. Intuitively, if all agents respect that condition, then we can only obtain runs where all agents output words in the invariant, and thus no-one broadcasts m_{err}.

The downward-closed property comes from the fact that if an agent outputs a word w, then other agents can receive any subsequence of letters of that word, as broadcasts can be lost. It is crucial as it gives us a finite representation of invariants, their basis.

Definition 3.19 ▶ Invariants for signature BGR

An *invariant* for a signature BGR over an alphabet $\mathcal{M}$ is a downward-closed set $I \subseteq \mathcal{M}^*$. We say that it is *sufficient* for a control strategy σ if it satisfies the following conditions:

1. $\varepsilon \in I$ and $m_{err} \notin I$
2. For all σ-local run u, if $\mathbf{In}_d(u) \in I$ for all $d \in \mathbb{D}$ then $\mathbf{Out}_{sign}(u) \in I$.

The next step is to show that if we have a winning strategy we can always find a sufficient invariant for it.

Lemma 3.20 ▶ Invariants characterise winning strategies

A control strategy σ is winning if and only if there exists a sufficient invariant $I \subseteq \mathcal{M}^*$ for it.

Proof. $\Rightarrow$ Suppose σ is winning. Let I be the set of words such that there exists a σ-run, an agent a and a datum d such that w is a subword of the d-output of the projection of that run on a. The empty word is in I as it is the output of a local run of length 0, which is a σ-run. As σ is winning, m_{err} can never be broadcast, thus $m_{err} \notin I$. For the other condition, consider a σ-local run $u = (s_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} (s_1, c_1) \xrightarrow{\mathbf{op}_2(m_2,d_2)}_{\delta_2} \cdots \xrightarrow{\mathbf{op}_k(m_k,d_k)}_{\delta_k} (s_k, c_k)$ whose d-input is in I for every datum d.

Then we can construct a σ-run in which some agent has output $\mathbf{Out}_{sign}(u)$, thus proving that $\mathbf{Out}_{sign}(u) \in I$. This construction is illustrated in Figure 3.6.

Let D be the set of data appearing in u. For each datum $d \in D$, let w_d be the d-input of u. As $w_d \in I$, there exists a σ-run ϱ_d such that w_d is a subword of the output of an agent a_d. Let $\mathbb{A}_d$ be the set of agents of that σ-run.

Up to renaming data and agents, we can assume that the initial datum of a_d in ϱ_d is d, and that the σ-runs $(\varrho_d)_{d\in D}$ operate over disjoint sets of data and agents.

We take a fresh agent a. We construct a σ-run ϱ over $\{a\} \sqcup \bigsqcup_{d\in D} \mathbb{A}_d$ as follows. We make a follow the local run u. Whenever a needs to receive a message (m, d), we run ϱ_d over $\mathbb{A}_d$ until a message (m, d) is broadcast by a_d, and make a receive it. Then we continue running u. As $\mathbf{In}_d(u)$ is a subword of the output of $\pi_{a_d}(\varrho_d)$ for all $d \in D$, we eventually run u in full.

This yields a valid σ-run in which u is fully executed by a. Hence we have a σ-run in which agent a outputs $\mathbf{Out}_{sign}(u)$. By definition of I, we thus have $\mathbf{Out}_{sign}(u) \in I$.
$\Leftarrow$ Suppose there exists $I \subseteq \mathcal{M}^*$ satisfying the conditions of the lemma. Suppose by contradiction that there is a σ-run ϱ in which m_{err} is broadcast.

[5]As I is downward-closed, $\varepsilon \in I$ is synonymous with I being non-empty.

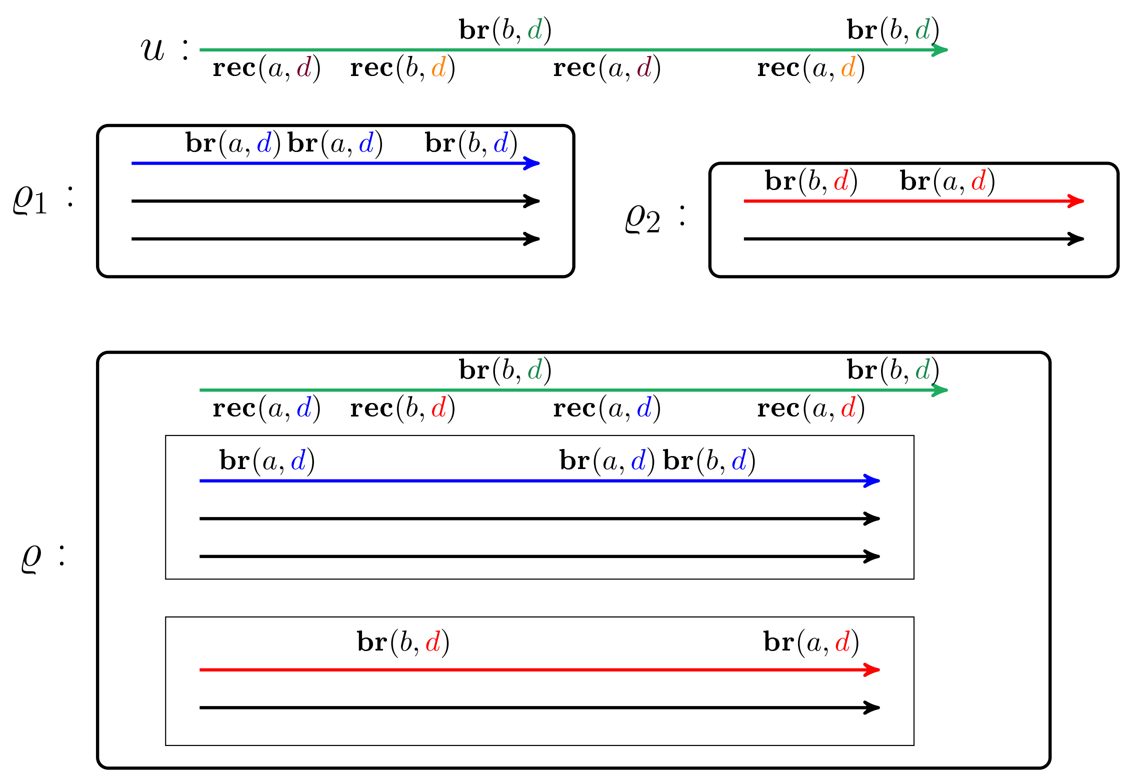


Figure 3.6: Illustration of the proof of Lemma 3.20. Most information is omitted, we only represent schematically the relevant broadcasts and receptions. Data are represented by colours. If we have a local run u outputting bb and for each d a run in which an agent outputs the d-input of u, then we can rename some data and compose those runs to form a run in which an agent outputs bb. Local runs of relevant agents are coloured with their initial datum.

Let ϱ_- be the maximal prefix of ϱ such that the output of each agent is in I. It is well-defined as $\varepsilon \in I$, thus the prefix of ϱ with no step satisfies that condition. As $m_{err} \notin I$ and I is downward-closed, I does not contain any word containing m_{err}. Hence the output of ϱ is not in I, and thus ϱ_- is a strict prefix of ϱ.

Let a be the agent making the broadcast of the step right after ϱ_- in ϱ, and let m be the message it broadcasts. Let ϱ_+ be the prefix of ϱ made of ϱ_- and that extra step.

Let w_- be the output of a in ϱ_-. For all $d \in \mathbb{D}$, the d-input of a in ϱ_- must be a subword of the output of another agent. By definition of ϱ_-, the d-input of a in ϱ_- is thus in I for all d. As the d-input of a in ϱ_- and ϱ_+ is the same, the d-input of a in ϱ_+ is in I for all d. By maximality of ϱ_-, the output of a in ϱ_+ is not in I.

This contradicts the second condition on I given by the lemma. $\square$

Corollary 3.21 ▶ Characterisation of coverability in signature BNRA

Let $\mathcal{R}$ be a register transducer with r registers representing a BNRA. A message m_{err} is *not* coverable if and only if there exists a downward-closed set $I \subseteq \mathcal{M}^*$ such that:

1. $\varepsilon \in I$ and $m_{err} \notin I$
2. For all local run u, if $\mathbf{In}_d(u) \in I$ for all $d \in \mathbb{D}$ then $\mathbf{Out}_{sign}(u) \in I$.

Recall that, as a corollary of Higman's lemma, upward-closed sets of words can be finitely described by their finite basis.

In Appendix A we present a direct proof that COVER is decidable for signature BNRA. As we can enumerate runs, we know that the problem is recursively enumerable. To show that it is also co-recursively enumerable, we can enumerate invariants. The catch is that we need to be able to decide the second condition in Definition 3.19. This is the main content of the appendix.

For SAFESTRAT, we cannot enumerate strategies (there are uncountably many) thus we need a different technique. Our algorithm enumerates invariants and checks for each one whether there is a strategy such that the conditions listed in Lemma 3.20 are satisfied. While the first item is straightforward to check on the invariant, the second is not. To verify it, we design a game in which the two players construct a local run, and the received data are chosen by Environment.

Invariant game for signature BGR

Intuitively, the two players pick the transitions from their respective states, and Environment picks the data received at each step, when they are not already determined by the chosen transition. If at some point the d-input gets out of I for some d then the game stops and Controller wins. If the output gets out of I then the game stops and Environment wins. If none of the two happen and the game goes on forever then Controller wins.

This characterises the capacity of Controller to keep outputs within a given invariant, but if we made the choice of data explicit this game would be infinite.

We reduce it to a finite reachability game, called the invariant game which we can solve by a simple fixpoint computation.

A first observation is that it is always in Environment's best interest to choose fresh data that were never seen before, as they come with the smallest d-input. Thus whenever we receive a datum that is not in the registers we can assume that the associated d-input is empty. This means that we do not need to remember the d-inputs associated to every datum of the local run, but only those that are currently in the registers.

In order to formalise the definition of the game, we need to define notions of input and output on sequences of transitions. The idea is that we can assume that every datum that disappears from the registers will never appear again. In order to check whether some d-input gets out of I, we only need to keep track of the sequences of letters received with the data currently in the registers. We call those the recent inputs. Furthermore, in our model of register transducers, a received datum always appears in at most one register at a time, and while it is not forgotten, it stays in that one register. This will allow us to read the recent inputs directly from the sequence of transitions.

We fix a signature BGR $\mathcal{G} = (\mathcal{R}, Q_{\mathsf{ctrl}}, Q_{\mathsf{env}}, q_{err})$ for the rest of this section. Given a sequence of transitions $\delta_1 \cdots \delta_k$ of $\mathcal{R}$, we define its *output* as the sequence of letters sent by broadcasts. For all registers $i \in [1, r]$, we also define its *recent input on i* as the sequence of letters received with an equality transition with register i since it was last updated.

Formally, the output of $\delta_1 \cdots \delta_k$ is defined inductively as $\mathsf{Out}(\varepsilon) = \varepsilon$ and

$$\mathsf{Out}(\delta_1 \cdots \delta_{k+1}) = \begin{cases} \mathsf{Out}(\delta_1 \cdots \delta_k) & \text{if } \delta_{k+1} \text{ is a reception,} \\ \mathsf{Out}(\delta_1 \cdots \delta_k)m & \text{if } \delta_{k+1} = \xrightarrow{\mathbf{br}(m,1)} \text{for some } m. \end{cases}$$

The recent input on i is defined as $\mathsf{recentIn}_i(\varepsilon) = \varepsilon$ and:

$$\mathsf{recentIn}_i(\delta_1 \cdots \delta_{k+1}) = \begin{cases} m & \text{if } \delta_{k+1} = \xrightarrow{\mathbf{rec}(m,\downarrow i)} \text{ for some } m \text{ and } i, \\ \mathsf{recentIn}_i(\delta_1 \cdots \delta_k)m & \text{if } \delta_{k+1} = \xrightarrow{\mathbf{rec}(m,=i)} \text{ for some } m \text{ and } i, \\ \mathsf{recentIn}_i(\delta_1 \cdots \delta_k) & \text{otherwise.} \end{cases}$$

Note that we always have $\mathsf{recentIn}_1(\delta_1 \cdots \delta_k) = \varepsilon$, as we assumed that no reception is made using register 1.

The *invariant game* $\mathcal{IG}(\mathcal{R}, I)$ goes as follows. The set of vertices is simply $Q_{\mathcal{R}}$. From each vertex $q \in Q_{\mathcal{R}}$, players choose a transition from q in $\Delta_{\mathcal{R}}$.. Controller chooses the next transition when the current vertex is in Q_{ctrl}, Environment when it is in Q_{env}.

- If at some point the play $\pi = \delta_1 \cdots \delta_k$ is such that $\mathsf{recentIn}_i(\pi) \notin I$ for some $i \geq 2$ then Controller wins.
- If at some point we see a reception transition receiving a letter $m \notin I$ then Controller wins. This case is not covered by the previous one as we may receive messages with data that do not appear in the registers and not store it. The received letter then does not appear in the recent inputs.
- If at some point the play $\pi = \delta_1 \cdots \delta_k$ is such that $\mathsf{Out}(\pi) \notin I$ then Environment wins.
- If the play goes on forever without any of those things happening then Controller wins.

We start by showing that we can solve this game by considering it as a regular safety game. We obtain as a corollary that if Environment wins then he can win in a bounded number of steps.

Define $\varphi(\mathcal{R}, B) := |\mathcal{R}|(||B|| + 1)^{|\mathcal{R}|(|B|+1)}$

Lemma 3.22 ▶ Decidability of the invariant game

Given a BGR over protocol $\mathcal{R}$ and a finite set of words B, we can decide in exponential time whether Controller has a winning strategy in $\mathcal{IG}(\mathcal{R}, (B\uparrow)^{\mathsf{c}})$. Furthermore, if Environment has a winning strategy then he has a strategy to win in at most $\varphi(\mathcal{R}, B)$ steps.

Proof. By Lemma 3.7, $B\uparrow$ is a regular language, recognised by a deterministic finite automaton $\mathcal{A}_{B\uparrow} = (Q_B, \mathcal{M}, \Delta_B, q_0^B, F_B)$ with $(||B|| + 1)^{(|B|+1)}$ states.

We can construct a deterministic automaton $\mathcal{B}$ over the alphabet $\Delta_{\mathcal{R}}$ that reads plays $\delta_1 \cdots \delta_k$ of $\mathcal{IG}(\mathcal{R}, (B\uparrow)^c)$ and accepts exactly the winning plays for Environment.

Its set of states is $(Q_B)^r$, plus a rejecting sink state $\bot$ and an accepting sink state $\top$, which is the only accepting state. The first component keeps track of the state reached in $\mathcal{A}_{B\uparrow}$ by the output of the sequence of transitions. The others keep track, for each register $i \geq 2$, of the state reached by the recent input on i in $\mathcal{A}_{B\uparrow}$.

Transitions of that automaton are easy to infer from the definition of output and recent input on i.

The automaton goes to $\bot$ if the recent input on i is in $B\uparrow$ for some i, or if it sees a reception transition of a message $m \in B\uparrow$.

It goes to $\top$ if the output is in $B\uparrow$.

By Proposition 2.8, we can solve this game in polynomial time in the size of the automaton $\mathcal{B}$ and the size of the arena of $\mathcal{IG}(\mathcal{R}, (B\uparrow)^c)$ (i.e., $|\mathcal{R}|$), that is, in exponential time in $||B|| + |B| + |\mathcal{R}|$.

Furthermore, if Environment has a winning strategy then he has one that guarantees that he wins in at most $\varphi(\mathcal{R}, B) = |\mathcal{A}_{B\uparrow}|^r |\mathcal{R}|$ steps. □

We have two things to prove: First that a winning strategy for Controller in $\mathcal{IG}(\mathcal{R}, I)$ yields a control strategy σ for which I is a sufficient invariant. Then, that a winning strategy for Environment in $\mathcal{IG}(\mathcal{R}, I)$ implies that there is no control strategy for which I is a sufficient invariant.

Lemma 3.23

Let $I \subseteq \mathcal{M}^*$ be a downward-closed set of words containing ε and not m_{err}. If Controller wins the invariant game $\mathcal{IG}(\mathcal{R}, I)$ then there is a control strategy σ such that I is a sufficient invariant for σ.

Proof. Let $\sigma_{\mathcal{IG}}$ be a winning strategy for Controller in $\mathcal{IG}(\mathcal{R}, I)$.

We define σ as the control strategy in which Controller follows $\sigma_{\mathcal{IG}}$. That is, given a local run $u = (s_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_k(m_k,d_k)}_{\delta_k} (s_k, c_k)$, we set $\sigma(u) = \sigma_{\mathcal{IG}}(\delta_1 \cdots \delta_k)$.

We show that I is a sufficient invariant for σ. To do so, we assume by contradiction that we have a σ-local run $u = (s_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_k(m_k,d_k)}_{\delta_k} (s_k, c_k)$ such that u has an output outside of I, and its d-input is in I for all $d \in \mathbb{D}$.

We then show that $\pi = \delta_1 \cdots \delta_k$ is a losing $\sigma_{\mathcal{IG}}$-play for Controller in $\mathcal{IG}(\mathcal{R}, I)$. As u is a σ-local run, by definition of σ, $\delta_1 \cdots \delta_k$ is a $\sigma_{\mathcal{IG}}$-play.

For all $j \in [0, k]$ let u_j be the prefix of u up to (s_j, c_j) and $\pi_j = \delta_1 \cdots \delta_j$.

Claim 3.23.1. *For all $j \in [0, k]$ and $i \in [2, r]$ we have* $\mathsf{recentIn}_i(\pi_j) \sqsubseteq \mathbf{In}_{u_j}(c_j(i))$ *and* $\mathsf{Out}(\pi_j) = \mathbf{Out}_{sign}(u_j)$.

Proof of the claim. By a straightforward induction on j. ■

We can instantiate the previous claim with $j = k$ to obtain $\mathsf{Out}(\pi) = \mathbf{Out}_{sign}(u)$. As we assumed that $\mathbf{Out}_{sign}(u) \notin I$, we have $\mathsf{Out}(\pi) \notin I$.

As the d-input of u is in I for all $d \in \mathbb{D}$, and I is downward-closed, the letters of all messages received in u are in I. Moreover, by the previous claim, for all $j \in [0, k]$, we have $\mathsf{recentIn}_i(\pi_j) \sqsubseteq \mathbf{In}_{u_j}(c_j(i)) \sqsubseteq \mathbf{In}_u(c_j(i)) \in I$. As I is downward-closed, we have $\mathsf{recentIn}_i(\pi_j) \in I$ for all i, j.

As a result, π is a losing $\sigma_{\mathcal{IG}}$-play for Controller. This contradicts the assumption that $\sigma_{\mathcal{IG}}$ is a winning strategy for $\mathcal{IG}(\mathcal{R}, I)$. In consequence, there is no σ-local run u whose output is outside of I, and whose d-input is in I for all $d \in \mathbb{D}$.

This means that I is a sufficient invariant for σ. ☐

Lemma 3.24

Let σ be a control strategy. Let $I \subseteq \mathcal{M}^*$ be a downward-closed set of words containing ε and not m_{err}, and let B be the basis of I^c.
If Environment wins the invariant game $\mathcal{IG}(\mathcal{R}, I)$ then there is a σ-local run of length at most $\varphi(\mathcal{R}, B)$ with an output not in I and all d-inputs in I.

Proof. By Proposition 2.8 there exists $\tau_{\mathcal{IG}}$ a winning strategy $\tau_{\mathcal{IG}}$ for Environment in the invariant game $\mathcal{IG}(\mathcal{R}, I)$ such that Environment always wins in at most $\varphi(\mathcal{R}, B)$ steps.

We construct a σ-local run of length at most $\varphi(\mathcal{R}, B)$ with an output not in I and all d-inputs in I. To do so, we apply $\tau_{\mathcal{IG}}$ to choose transitions and we choose data by always picking a datum never seen before in the run, when the datum is not determined by the transition.

Let (s_0, c_0) be an initial configuration of $\mathcal{R}$. We define iteratively a sequence of steps $(s_{k-1}, c_{k-1}) \xrightarrow{\mathbf{op}_k(m_k, d_k)}_{\delta_k} (s_k, c_k)$ as follows. Suppose we defined them up to (s_{k-1}, c_{k-1}), and let u_{k-1} be the local run defined so far. We first choose δ_k:

- If $s_{k-1} \in Q_{\mathsf{ctrl}}$ then $\delta_k = \sigma(u_{k-1})$,
- otherwise $\delta_k = \tau_{\mathcal{IG}}(\delta_1 \cdots \delta_{k-1})$.

We then choose d_k:

- If δ_k is a broadcast transition of letter m, we set $d_k = c_k(1)$ (the initial datum of the local run).
- If δ_k is a record transition or a disequality transition, we pick a datum d_k that does not appear in u_{k-1} before.
- If $\delta_k = s_{k-1} \xrightarrow{\mathbf{rec}(m,=i)} s_k$ is an equality transition of letter m, we set $d_k = c_{k-1}(i)$.

Clearly we maintain the fact that u_k is a σ-local run and $\delta_1 \cdots \delta_k$ is a $\tau_{\mathcal{IG}}$-play in $\mathcal{IG}(\mathcal{R}, I)$. We stop when $\delta_1 \cdots \delta_k$ is winning for Environment in $\mathcal{IG}(\mathcal{R}, I)$, which happens for some $k \leq \varphi(\mathcal{R}, B)$. Let K be the final value of k and $u = u_K$ be the local run obtained at the end.

It remains to show that the output of u is not in I while all its d-inputs are in I. To do so, we rely on the following claim:

Claim 3.24.1. *For all register i and index k, $\mathsf{recentIn}_i(\delta_1 \cdots \delta_k) = \mathbf{In}_{u_k}(c_k(i))$. Furthermore, $\mathsf{Out}(\delta_1 \cdots \delta_k) = \mathbf{Out}_{sign}(u_k)$*

Proof. By a straightforward induction on k. ☐

By definition $\delta_1 \cdots \delta_K$ is a winning $\tau_{\mathcal{IG}}$-play for Environment, hence its output is not in I, thus $\mathbf{Out}_{sign}(u) = \mathbf{Out}_{sign}(u_K)$ is not in I either. Let $d \in \mathbb{D}$ a datum appearing in u, and let k be such that (s_k, c_k) is the last configuration in which d appears. Let i be the register such that $c_k(i) = d$. Then we have $\mathbf{In}_u(d) = \mathbf{In}_{u_k}(c_k(i)) = \mathsf{recentIn}_i(\delta_1 \cdots \delta_k)$. As $\tau_{\mathcal{IG}}$ is winning for Environment, $\mathsf{recentIn}_i(\delta_1 \cdots \delta_k) \in I$, and thus $\mathbf{In}_u(d) \in I$.

We have found a σ-local run of length at most $\varphi(\mathcal{R}, B)$ whose output is not in I while all its d-inputs are. □

Our next step is to bound the minimal size of a sufficient invariant for some winning control strategy σ when there is one. The idea is as follows: Take an invariant I such that the basis $\{w_1, \dots, w_k\}$ of I^{c} has as few elements as possible. We can assume that $|w_1| \leq \cdots \leq |w_k|$. Then we know that, for all i, $\{w_1, \dots, w_i\}$ is not a sufficient invariant for σ. Hence by Lemma 3.24 we get a σ-local run of bounded size breaking the invariant $\{w_1, \dots, w_i\}$, which forces $\{w_{i+1}, \dots, w_k\}$ to contain a word of bounded size. This bounds the size of w_{i+1} with respect to $w_1, \dots, w_i$, as stated in the lemma below. We will then be able to leverage the Length Function Theorem to bound the size of the basis of I^{c}.

Define $\psi(n) = |\mathcal{R}|(n+1)^{|\mathcal{M}|^{n+1}+1}$

Lemma 3.25 ▶ Bounding the size of the invariant

Let $\mathcal{G}$ a signature BGR. There is a winning control strategy for $\mathcal{G}$ if and only if there is a sequence of words $w_0, \dots, w_k \in \mathcal{M}^*$ such that

- Controller wins $\mathcal{IG}(\mathcal{R}, \{w_1, \dots, w_k\}{\uparrow}^{\mathsf{c}})$,
- and for all $i \in [1, k]$, $|w_i| \leq \psi(|w_{i-1}|)$.

Proof. By Lemma 3.23, if there is a sequence of words $w_0, w_1, \dots, w_k$ such that Controller wins the invariant game $\mathcal{IG}(\mathcal{R}, (\{w_0, w_1, \dots, w_k\}{\uparrow})^{\mathsf{c}})$ then there is a control strategy such that $(\{w_0, w_1, \dots, w_k\}{\uparrow})^{\mathsf{c}}$ is a sufficient invariant for σ. Hence, by Lemma 3.20, σ is a winning control strategy.

Conversely, suppose there is a winning control strategy σ. By Lemma 3.20 there is a downward-closed sufficient invariant $I \subseteq \mathcal{M}^*$ for σ.

First, by (the contraposition of) Lemma 3.24 Controller wins $\mathcal{IG}(\mathcal{R}, I)$, so the first condition of the lemma is satisfied.

For the second condition, as I^{c} is upward-closed it has a finite basis B. Let $w_0, w_1, \dots, w_k$ be the elements of B sorted by length, i.e., $|w_i| \leq |w_{i+1}|$ for all i. We can assume that we took I so that k is minimal. For all $j \in [1, k]$, we define $B_j = \{w_i \mid i < j\}$ and $I_j = B_j{\uparrow}^{\mathsf{c}}$. Note that we have $I \subseteq I_k \subseteq \ldots \subseteq I_0$. As I contains ε and not m_{err}, we can assume $w_0 = m_{err}$. By minimality of k, for all $j \in [1, k]$ the set I_j is not a sufficient invariant for σ.

By Lemma 3.24, there is a σ-local run of length at most $\varphi(\mathcal{R}, B_j)$ whose output is not in I_j and whose d-inputs are all in I_j. As I is a sufficient invariant for σ, one of those d-inputs must not be in I. We choose one of those and call it w. As a consequence, there exists w_ℓ with $\ell \geq j$ such that $w_\ell \sqsubseteq w$, and thus $|w_\ell| \leq |w| \leq \varphi(\mathcal{R}, B_j)$. As $|w_i| \leq |w_{i+1}|$ for all i, this implies $|w_j| \leq \varphi(\mathcal{R}, B_j) = |\mathcal{R}|(||B_j|| + 1)^{(|B_j|+1)}$. As w_{j-1} is of maximal length among words of B_j, we have $||B_j|| = |w_{j-1}|$ and $|B_j| \leq |\mathcal{M}|^{|w_{j-1}|+1}$.

As a result, $|w_j| \leq |\mathcal{R}|(|w_{j-1}|+1)^{|\mathcal{M}|^{|w_{j-1}|+1}+1} = \psi(|w_{j-1}|)$. Thus the second condition of the lemma is also satisfied. □

Theorem 3.26

SafeStrat is decidable and in $\mathbf{F}_{\omega^\omega}$ for signature BGR.

Proof. Let $\mathcal{G}$ a BGR. We apply the Length Function Theorem with $\Sigma = \mathcal{M}$ and $g(n) = n(n+1)^{n^{n+1}+1}$. We obtain a function $f \in \mathscr{F}_{\omega^{|\mathcal{M}|-1}}$ such that every (g,n)-controlled bad sequence of words $w_0, w_1, ..., w_k$ has at most $f(n)$ terms.

We use a non-deterministic algorithm that guesses a sequence of words $w_1, ..., w_k$ such that $w_1 = m_{err}$ and $|w_i| \leq |w_{i+1}| \leq \psi(|w_i|)$ for all i. One can straightforwardly check that then we have $|w_i| \leq g^{(i)}(|\mathcal{R}| + |\mathcal{M}| + 1)$ for all i.

Let $B = \{w_0, w_1, ..., w_k\}$.

The algorithm then checks that there exists a strategy σ such that the complement of $\{w_0, w_1, ..., w_k\}\uparrow$ is a sufficient invariant for σ, by solving the invariant game $\mathcal{IG}(\mathcal{R}, (\{w_0, w_1, ..., w_k\}\uparrow)^c)$. This can be done in exponential time in $|\mathcal{R}| + k + |w_k|$, by Lemma 3.22. We accept if there is such a strategy and reject otherwise.

By Lemma 3.25, this algorithm is correct. We can make it deterministic with an exponential blow-up in the time complexity. The time required by this algorithm is therefore $h(f(|\mathcal{R}| + |\mathcal{M}| + 1))$ with h a primitive recursive function. As $\mathscr{F}_{\omega^{|\mathcal{M}|-1}}$ is closed under composition with primitive recursive functions, the algorithm takes a time bounded by a function of $\mathscr{F}_{\omega^{|\mathcal{M}|-1}}$.

As a consequence, the problem is in $\mathbf{F}_{\omega^\omega}$. □

Corollary 3.27

Cover is decidable and in $\mathbf{F}_{\omega^\omega}$ for signature BGR.

3.5 General case

In this section we generalise the previous result to all BGR.

Theorem 3.28

The Cover and SafeStrat problem are decidable in $\mathbf{F}_{\omega^\omega}$ for general BGR.

We fix a BGR $\mathcal{G} = (\mathcal{R}, Q_{ctrl}, Q_{env}, m_{err})$ for the rest of this section. Note that we choose to use an error letter m_{err} instead of an error state q_{err}: This is more convenient for the proof, and does not weaken our results in light of Remark 3.2.2.

The general structure of the proof is the same as before, but the removal of the signature hypothesis makes it significantly more technical. The main difference between the signature and general models is that in the latter a process can send acknowledgements to a process it received messages from, as in the right protocol in Figure 3.2. As we will see, this phenomenon requires us to use more complex invariants to extend the proof to all BGR.

We make the following informal observation: Say an agent receives a message (m, d) with d its initial datum; this is possible in general BGR but not in signature ones. Then this means that other agents, which did not have this datum initially, received enough messages with datum d to be able to broadcast (m, d). Intuitively, we can copy these agents many times, which allows us to assume that we have an unlimited supply of messages (m, d). In sum, we will show that if an agent sends a message (m, d) with d that is not its initial datum, then from this point on we can assume that messages (m, d) are for free. This intuition justifies the definition of decomposition, which describes the sequence of letters sent with a given datum during a run. It details the sequence of letters sent by the agent with that datum initially, and the points at which each letter is first broadcast with that datum by another agent.

Definition 3.29

A *decomposition* is a tuple $\mathsf{dec} = (v_0, m_1, \dots, v_{k-1}, m_k, v_k)$ with $m_0, \dots, m_k$ distinct letters of $\mathcal{M}$ and $v_i \in \mathcal{M}^*$ for all i.
A word $w \in \mathcal{M}^*$ *matches* dec if $w = w_0 \cdots w_k$ where each w_i can be obtained by inserting letters from $\{m_1, \dots, m_i\}$ in v_i.

Example 3.5.1. *Let $\mathcal{M} = \{a, b, c\}$. Then $\mathsf{dec} = (abba, a, cbc, b, abc)$ is a decomposition. The word abbacabaacbabbc matches dec as we can cut in in three parts abbacabaacabbabca, and cabaac can be obtained by adding some a to cbc and abbabca can be obtained by adding some a and b to abc.*

We write $\mathcal{L}_{\mathsf{dec}}$ for the language of words that match dec.

Given a family of upward-closed sets of words $(J_m)_{m\in\mathcal{M}}$, we define $\mathcal{D}((J_m)_{m\in\mathcal{M}})$ as the set of decompositions

$$\mathcal{D}((J_m)_{m\in\mathcal{M}}) = \{(v_0, m_1, \dots, v_{k-1}, m_k, v_k) \mid \forall i, \mathcal{L}_{(v_0, m_1, \dots, v_{i-1})} \cap J_{m_i} \neq \emptyset\}.$$

With an additional downward-closed set I, we also define

$$\mathcal{D}(I, (J_m)_{m\in\mathcal{M}}) = \{(v_0, m_1, \dots, v_{k-1}, m_k, v_k) \mid v_0 \cdots v_k \in I, \forall i, \mathcal{L}_{(v_0, m_1, \dots, v_{i-1})} \cap J_{m_i} \neq \emptyset\}.$$

Finally, the set of words producible by $I, (J_m)_{m\in\mathcal{M}}$ is

$$\mathcal{L}(I, (J_m)_{m\in\mathcal{M}}) = \bigcup_{\mathsf{dec}\in\mathcal{D}(I,(J_m)_{m\in\mathcal{M}})} \mathcal{L}_{\mathsf{dec}}.$$

We say that a local run u with initial datum d is *compatible* with a decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_{k-1}, m_k, v_k)$ if $u = u_0 \cdots u_k$ where $v_i \sqsubseteq \mathbf{Out}_d(u_i)$ and $\mathbf{In}_d(u_i) \in \{m_1, \dots, m_i\}^*$ for all i.

Let us take a moment to explain those definitions. Here I should be thought of as the set of words over $\mathcal{M}$ that can be broadcast by an agent with its initial datum. Meanwhile, J_m represents the set of words w over $\mathcal{M}$ such that an agent can broadcast (m, d) with d not its initial datum while having received before only (a subword of) w with that datum. It can be read as the "cost" of a message m: in order to receive a message (m, d) you should first broadcast a sequence of letters of J_m with datum d.

A decomposition $(v_0, m_1, \dots, v_k)$ is a scenario of the sequence of letters broadcast over a datum d during a run: The agent who has d as initial datum broadcasts $v_0 \cdots v_k$ with it, while $m_1, \dots, m_k$ mark the points at which each of those letters is first broadcast with datum d by another agent.

Then, we can see $\mathcal{D}(I, (J_m)_{m\in\mathcal{M}})$ as the set of decompositions $(v_0, m_1, \dots, v_k)$ that are compatible with the invariant $I, (J_m)_{m\in\mathcal{M}}$. The condition $v_0 \cdots v_k \in I$ means that an agent with d as initial datum should be able to broadcast $v_0 \cdots v_k$ with it. The other condition says that for all i there is a word $w \in \mathcal{L}_{(v_0, m_1, \dots, v_{i-1})} \cap J_{m_i}$. This should be read as follows:

- $w \in J_{m_i}$ means that if we can broadcast the sequence w with datum d, we can make an agent broadcast (m_i, d)

- $w \in \mathcal{L}_{(v_0, m_1, \dots, v_{i-1})}$ means that we can broadcast the sequence w with datum d, as we can obtain it from $v_0 \cdots v_{i-1}$ by adding enough $m_1, \dots, m_{i-1}$.

3.5.1 Characterisation of winning strategies with invariants

An *invariant* for general BGR is made of a downward-closed set of words $I \subseteq \mathcal{M}^*$ (the sequences of letters that may be produced over some datum) and an upward-closed set of words $J_m \subseteq \mathcal{M}^*$ for each letter m (the sequences of letters that allow an agent to send a message (m, d) with d that is not its initial datum).

Definition 3.30 ▶ Invariants for BGR

An *invariant* for general BGR is a pair $(I, (J_m)_{m \in \mathcal{M}})$ with $I \subseteq \mathcal{M}^*$ a downward-closed set of words and, for all m, $J_m \subseteq \mathcal{M}^*$ an upward-closed set of words.
We say that it is *sufficient* for a control strategy σ if the following conditions hold.

1. $\varepsilon \in I$, $m_{err} \notin I$ and $J_{m_{err}} \cap I = \emptyset$
2. $\mathcal{L}(I, (J_m)_{m \in \mathcal{M}}) \subseteq I$
3. For all initial σ-local run u with initial datum d, if:
   (i) u is compatible with a decomposition $\mathsf{dec} \in \mathcal{D}((J_m)_{m \in \mathcal{M}})$, and
   (ii) for all $d' \neq d$, $\mathbf{In}_{d'}(u) \in I$,

   then we have that

   (a) $\mathbf{Out}_d(u) \in I$
   (b) for all $m \in \mathcal{M}$ and $d' \neq d$, if u contains a broadcast of (m, d') then $\mathbf{In}_{d'}(u) \in J_m$.

We once again prove that every winning control strategy has a sufficient invariant.

Lemma 3.31 ▶ Invariants characterise winning strategies

A control strategy σ is winning if and only if there exists a sufficient invariant $(I, (J_m)_{m \in \mathcal{M}})$ for it.

The rest of this section is dedicated to the proof of this lemma. To do so, we need an argument that resembles the construction illustrated in Figure 3.6. However, the construction gets more involved in this case.

We can start by proving the easier direction of the equivalence, given by the following lemma.

Lemma 3.32

If there exists a sufficient invariant $(I,(J_m)_{m\in\mathcal{M}})$ for a control strategy σ then σ is winning.

Proof. Suppose σ has a sufficient invariant $(I,(J_m)_{m\in\mathcal{M}})$. Suppose by contradiction that there is a σ-run ϱ in which m_{err} is broadcast.

Let a be an agent broadcasting m_{err} in ϱ, let u be its local run.

We first show that the local run of a in ϱ does not satisfy (a) and (b).

If m_{err} is broadcast in u with its initial datum then, as $m_{err}\notin I$ and I is downward-closed, we cannot have $\mathbf{Out}_d(u)\in I$. On the other hand, if m_{err} is broadcast in u with another datum d' then as $J_{m_{err}}=\emptyset$, $\mathbf{In}_{d'}(u)\notin J_{m_{err}}$. Hence u does not satisfy (a) and (b).

Let ϱ_- be the maximal prefix of ϱ such that the local runs of all agents satisfy (a) and (b). It is well-defined: we saw that the full run ϱ does not satisfy this requirement, and as $\varepsilon\in I$, the prefix of ϱ with no step satisfies it. Furthermore ϱ_- must be a strict prefix of ϱ.

Let a be the agent making the broadcast of the step right after ϱ_- in ϱ, and let m be the message it broadcasts. Let ϱ_+ be the prefix of ϱ made of ϱ_- and that extra step.

By maximality of ϱ_-, there must be an agent whose local run in ϱ_+ does not satisfy (a) and (b). This agent can only be a: all agents satisfied both conditions in ϱ_-, an agent cannot switch from satisfying to not satisfying those conditions without making a broadcast (for (b), this is due to the fact that all J_m are upward-closed), and a is the only one who made a broadcast in the last step.

As a consequence, the local run u_+ of a in ϱ_+ must dissatisfy either (a) or (b). It remains to show that u_+ satisfies both (i) and (ii) to obtain a contradiction.

We start by showing that the local run u_- of a in ϱ_- satisfies (i) and (ii).

- Let d be the initial datum of u_-. Let $m_1,\dots,m_k$ be the letters such that (m_i,d) is broadcast by an agent that is not a during ϱ_-. Let us cut ϱ_- into sections $\varrho_0\cdots\varrho_k$ such that $\varrho_0\cdots\varrho_i$ is the maximal prefix of ϱ_- in which (m_i,d) has not been broadcast by any agent apart from a. For each i let u_i be the projection of ϱ_i on a. We thus have $u_-=u_0\cdots u_k$. Let a_{m_i} be the first agent different from a who broadcasts (m_i,d) and let u_{m_i} be the projection of ϱ_- on a_{m_i}. Let w_i be the sequence of letters broadcast in ϱ_i with datum d.

  Consider the decomposition $\mathsf{dec}=(v_0,m_1,\dots,v_{k-1},m_k,v_k)$ where $v_i=\mathbf{Out}_d(u_i)$. By definition u_- must be compatible with it. Let $i\in[1,k]$. As u_{m_i} satisfies (b), we have $\mathbf{In}_d(u_{m_i})\in J_{m_i}$. By definition, we must have $\mathbf{In}_d(u_{m_i})\sqsubseteq w_0\cdots w_{i-1}$ and thus $w_0\cdots w_{i-1}\in J_{m_i}$ as J_{m_i} is upward-closed. Furthermore, each w_j (the letters sent in ϱ_j with datum d) can be obtained from v_j (the ones sent by a) by adding letters of $\{m_1,\dots,m_j\}$ (the broadcasts of other agents). As a result, we have $w_0\cdots w_{i-1}\in\mathcal{L}_{(v_0,m_1,\dots,v_{i-1})}$. We obtain that $w_0\cdots w_{i-1}\in J_{m_i}\cap\mathcal{L}_{(v_0,m_1,\dots,v_{i-1})}$, thus $J_{m_i}\cap\mathcal{L}_{(v_0,m_1,\dots,v_{i-1})}$ is not empty. In conclusion, $\mathsf{dec}\in\mathcal{D}((J_m)_{m\in\mathcal{M}})$.

- We now show that u_- satisfies (ii).

  Let $d'\neq d$. If d' does not appear in ϱ_- then $\mathbf{In}_{d'}(u_-)=\varepsilon\in\mathcal{L}(I,(J_m)_{m\in\mathcal{M}})$. Otherwise, let a' be the agent whose initial datum in ϱ is d'. We set w' the sequence of letters broadcast with datum d' in ϱ_-. Clearly $\mathbf{In}_{d'}(u_-) \sqsubseteq w'$. In order to show that $\mathbf{In}_{d'}(u_-)$, it suffices to show that $w' \in \mathcal{L}(I, (J_m)_{m \in \mathcal{M}})$.

We use the same arguments as for the previous item: we cut ϱ_- into sections $\varrho'_0 \cdots \varrho'_k$ according to the times at which new letters are broadcast with d' by agents other than a. We then construct a decomposition $\mathsf{dec}' = (v'_0, m'_1, \dots, v'_{k-1}, m'_k, v'_k)$ where m'_i is the message broadcast with d' at the start of ϱ'_i and v'_i is the sequence of letters broadcast by a' in ϱ'_i.

We argue as before that $w' \in \mathcal{L}_{\mathsf{dec}'}$ and $\mathsf{dec}' \in \mathcal{D}(I, (J_m)_{m \in \mathcal{M}})$.

We have shown that u_- satisfied (i) and (ii). To obtain a contradiction, we must show that u_+ satisfies them as well. By definition, u_+ is u_- with an additional broadcast at the end.

- Let $\mathsf{dec} = (v_0, m_0, \dots, v_k) \in \mathcal{D}((J_m)_{m \in \mathcal{M}})$ be a decomposition such that u_- is compatible with dec. We have $u_- = u_0 \cdots u_k$ where $v_i \sqsubseteq \mathbf{Out}_d(u_i)$ and $\mathbf{In}_d(u_i) \in \{m_1, \dots, m_i\}^*$ for all i. Let u_k^+ be u_k to which we append the last broadcast in u_+. We obtain $u_- = u_0 \cdots u_{k-1} u_k^+$. Since $v_k \sqsubseteq \mathbf{Out}_d(u_k) \sqsubseteq \mathbf{Out}_d(u_k^+)$ and $\mathbf{In}_d(u_k) = \mathbf{In}_d(u_k^+) \in \{m_1, \dots, m_k\}^*$, we conclude that u_+ is compatible with dec. Hence u_+ satisfies (i).
- As $\mathbf{In}_{d'}(u_-) = \mathbf{In}_{d'}(u_+)$ for all $d' \in \mathbb{D}$, u_+ satisfies (ii).

In conclusion, we have constructed a σ-local run u_+ such that u_+ satisfies (i) and (ii) but not (a) and (b), yielding a contradiction.

□

We must now prove the other implication of Lemma 3.31. Intuitively, the argument goes as follows.

We define a notion of partial run. This is a run of a set of agents, but some messages can be received without being broadcast. They are called unmatched receptions. A local run is a particular case of partial run, with a single agent.

We assume that σ is winning. We take I as the downward-closure of the set of words $w \in \mathcal{M}^*$ such that there is a σ-run ϱ in which the sequence of messages w is broadcast, all with the same datum. For each m, we set J_m to be the upward-closure of the set of words $w = m_1 \cdots m_n$ such that there is a σ-partial run in which the sequence of unmatched receptions is of the form $(m_1, d) \cdots (m_n, d)$ for some $d \in \mathbb{D}$, and (m, d) is broadcast at some point. This should be understood as follows: if we have a run in which $(m_1, d) \cdots (m_n, d)$ is broadcast, then we can compose it with the partial run above, match all the unmatched receptions and obtain an extra broadcast of m.

The difficulty is to show that those sets form a sufficient invariant for σ. In particular, we need to take a σ-local run u satisfying (i) and (ii) and show that it satisfies (a) and (b). We do that by building σ-runs in which the local run of some agent is u.

We rely on several technical lemmas. Lemma 3.34, 3.35 and 3.36. Their statements are involved but they come with illustrations that should give helpful intuition. Before reading the details of those lemma we recommend that the reader reads the proof of Theorem 3.31 at the end of this section, to better understand how those lemmas are used.

Definitions for partial runs

For the following proof we need to introduce the notion of *partial run*, which describes the projection of a run on a subset of agents. We then show a key technical lemma that allows us to construct a run from a local run and a set of suitable partial runs.

We will use this lemma to prove a characterisation of winning control strategies using some invariants, like in the previous sections.

Definition 3.33

Let γ, γ' two configurations.
A *partial step* $\gamma \rightarrow_p \gamma'$ is defined if either $\gamma \rightarrow \gamma'$ (normal step) or there exist $m \in \mathcal{M}$, $d \in \mathbb{D}$ such that for all agent a either $\gamma(a) = \gamma'(a)$ or $\gamma(a) \xrightarrow{\mathbf{rec}(m,d)}_{\delta} \gamma'(a)$ for some reception transition δ (*unmatched reception* of (m,d)).
A *partial run* ϱ is a sequence of partial steps. It is *initial* if it starts in an initial configuration. Its *d-input* $\mathbf{In}_d(\varrho)$ is the sequence $m_0 \cdots m_k$ of letters corresponding to unmatched receptions with datum d in ϱ. Its *d-output* $\mathbf{Out}_d(\varrho)$ is the sequence of letters corresponding to broadcasts with datum d in ϱ.

Note that a local run can be seen as a partial run with a single agent. Given a control strategy σ, a *σ-partial run* is a partial run in which the local runs of all agents are σ-local runs.

A datum d is *initial* in ϱ if it appears in the first configuration. We extend the notion of compatible to partial runs: A partial run ϱ is *compatible* over d with a decomposition $\mathsf{dec} = (v_0, m_1, \ldots, v_k)$ if $\varrho = \varrho_0 \cdots \varrho_k$ and for all $i \in [0, k]$, $v_i \sqsubseteq \mathbf{Out}_d(\varrho_i)$ and $\mathbf{In}_d(\varrho_i) \in \{m_1, \ldots, m_i\}^*$, with d an initial datum of some agent in ϱ.

The following lemmas give us ways to compose partial runs to obtain complete runs.

Suppose we have a partial run ϱ compatible with a decomposition $\mathsf{dec} = (v_0, m_1, \ldots, v_k)$ over an initial datum d.

Suppose that we have, for each non-initial datum d', a run $\varrho_{d'}$ such that $\mathbf{In}_{d'}(\varrho) \sqsubseteq \mathbf{Out}_{d'}(\varrho_{d'})$.

Also suppose that for each $j \in [1, k]$ we have a partial run ϱ_j such that $\mathbf{In}_d(\varrho'_j) \in \mathcal{L}_{(v_0, m_1, \ldots, v_{i-1})}$ and which contains a broadcast of (m, d), and no unmatched receptions on data other than d.

- First, we show that given a word $w \in \mathcal{L}_{\mathsf{dec}}$ we can use the ϱ_i to extend ϱ and obtain a σ-partial run which is still compatible with dec and whose d-output contains w as a subword. This is done by composing ϱ with many copies of each ϱ_i to fill in the missing broadcasts.

- Then, we show that we can again use many copies of the ϱ_i to eliminate the unmatched receptions with datum d. We do this by carefully adding the necessary copies of ϱ_i, by decreasing i. Each time we fill in a missing broadcast of m_i while possibly adding new ones for some of the m_j with $j < i$. This terminates as the number of unmatched receptions of each letter m_i decreases with respect to the lexicographic ordering.

- We show that for each non-initial d' we can eliminate the unmatched receptions with datum d' by composing that partial run with the σ-runs $\varrho_{d'}$. We use the broadcasts in $\varrho_{d'}$ to match the unmatched receptions in ϱ over d'.

- Finally, we combine the two first steps to show that given a run compatible with a decomposition dec over some datum d and a word $w \in \mathcal{L}_{\mathsf{dec}}$, we can extend this run to obtain another run whose d-output contains w.

Extending the output

Lemma 3.34

Let $\mathsf{dec} = (v_0, m_1, \ldots, v_k)$ be a decomposition, let $w \in \mathcal{L}_{\mathsf{dec}}$.
Let d a datum and ϱ an initial σ-partial run compatible with dec over d.
Suppose that for all $j \in [1,k]$ there exist an initial σ-partial run ϱ'_j such that $\mathbf{In}_d(\varrho'_j) \in \mathcal{L}_{\mathsf{dec}_j}$ where $\mathsf{dec}_j = (v_0, m_1, \ldots, v_{j-1})$, $\mathbf{In}_{d'}(\varrho'_j) = \varepsilon$ for all $d' \neq d$ and $m_j \sqsubseteq \mathbf{Out}_d(\varrho'_j)$.
Then, there is a partial run $\tilde{\varrho}$ such that

- $\tilde{\varrho}$ is compatible with dec over d,
- $w \sqsubseteq \mathbf{Out}_d(\tilde{\varrho})$
- for all $d' \neq d$, either $\mathbf{In}_{d'}(\tilde{\varrho}) = \varepsilon$ or $\mathbf{In}_{d'}(\tilde{\varrho}) = \mathbf{In}_{d'}(\varrho)$

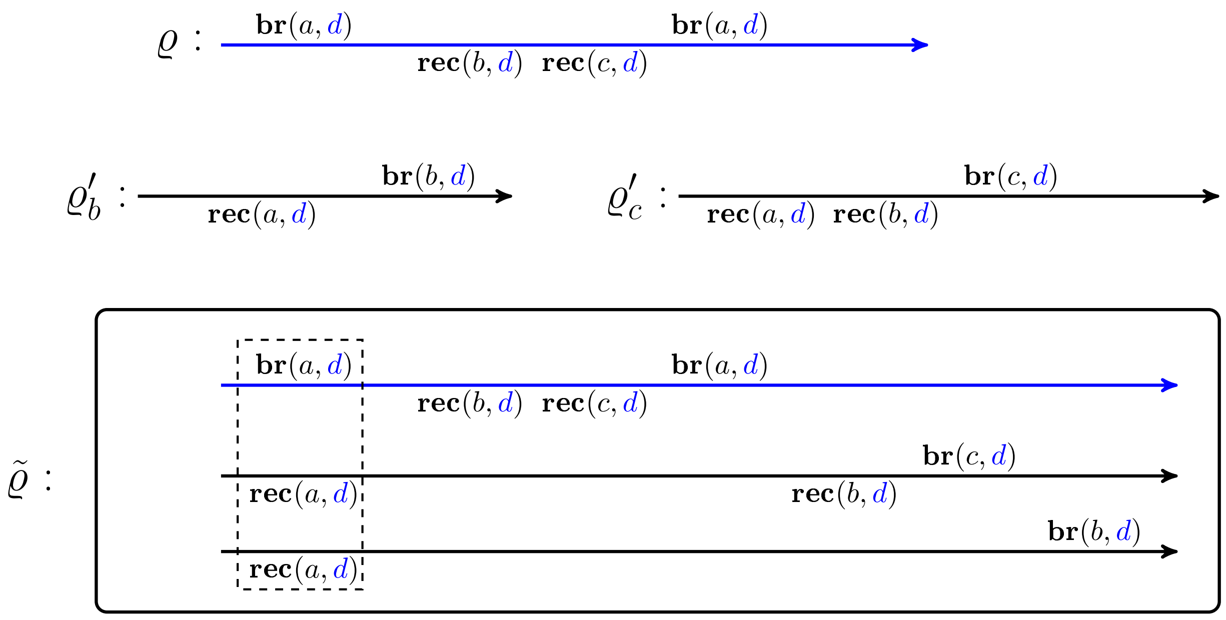


Figure 3.7: An illustration of the proof of Lemma 3.34. The partial run ϱ is compatible with decomposition (a,b,a,c,ε). We have $\mathbf{In}_d(\varrho_b) = a \in \mathcal{L}_{(a)}$ and $\mathbf{In}_d(\varrho_b) = ab \in \mathcal{L}_{(a,b,a)}$. We build a partial run $\tilde{\varrho}$ such that $aacb \sqsubseteq \mathbf{Out}_d(\tilde{\varrho})$. Note that $\tilde{\varrho}$ is also compatible with decomposition (a,b,a,c,ε). We ignore data other than d in this picture.

Proof. As $w \in \mathcal{L}_{\mathsf{dec}}$, we have $w = w_0 \cdots w_k$, where each w_i can be obtained by adding some letters of $\{m_1, \ldots, m_i\}$ to v_i. As u is compatible with dec, $u = u_0 \cdots u_k$ with $v_i \sqsubseteq \mathbf{Out}_d(u_i)$ for all i and $\mathbf{In}_d(u_i) \in \{m_1, \ldots, m_i\}^*$. As a consequence, to obtain a d-output that contains w, it suffices to show that we can add a letter from $\{m_1, \ldots, m_i\}$ at any point of u_i. We do so using $\tilde{\varrho}_i$: Since $\mathbf{In}_d(\tilde{\varrho}_i) \in \mathcal{L}_{(v_0,m_1,\dots,v_{i-1})}\downarrow$, we can split $\tilde{\varrho}_j$ into $\tilde{\varrho}_{j,0},\dots,\tilde{\varrho}_{j,j-1}$ so that $\mathbf{In}_d(\tilde{\varrho}_{j,i}) \sqsubseteq \tilde{w}_{j,i}$ where $\tilde{w}_{j,i}$ can be obtained by adding letters from $\{m_1,\dots,m_j\}$ to v_i.

We use the following composition operation: consider ϱ and one of the ϱ'_j. We can build a new run in which we execute both runs in parallel over disjoint sets of agents. We match each $\tilde{\varrho}_{i,j}$ with ϱ_j so that the broadcasts of ϱ_j with d forming v_i are received in $\tilde{\varrho}_{i,j}$ and the only remaining missing broadcasts in that section of the run are with letters $m_1,\dots,m_i$. We obtain a run section whose d-output still contains v_i and whose d-input only contains $m_1,\dots,m_i$. This lets us get to a point where the next step in $\tilde{\varrho}_j$ is a broadcast of (m_j,d) and ϱ has been executed up to the beginning of ϱ_j. We may then use the (m_j,d) broadcast at any moment in the rest of ϱ to extend the d-output. As a consequence, we can compose ϱ with the ϱ'_i as many times as necessary to obtain a run $\tilde{\varrho}$ whose d-output contains w.

Each composition maintains the fact that the run is compatible with dec. Further, for all $d' \neq d$, either d' does not appear in ϱ and $\mathbf{In}_d(\tilde{\varrho}) = \varepsilon$ or d' appears in ϱ and then $\mathbf{In}_{d'}(\tilde{\varrho}) = \mathbf{In}_{d'}(\varrho)$. □

Unmatched receptions with initial data

Lemma 3.35

Let $\mathsf{dec} = (v_0,m_1,\dots,v_k)$ be a decomposition, d a datum, ϱ an initial σ-partial run compatible with dec over d.
Suppose that for all $j \in [1,k]$ there exist an initial partial run ϱ'_j in which d is not initial such that $\mathbf{In}_{\varrho'_j}(d) \in \mathcal{L}_{\mathsf{dec}_j}$ where $\mathsf{dec}_j = (v_0,m_1,\dots,v_{j-1})$, $\mathbf{In}_{\varrho'_j}(d') = \varepsilon$ for all $d' \neq d$ and $m_j \sqsubseteq \mathbf{Out}_d(\varrho'_j)$.
Then, there exist a σ-partial run $\tilde{\varrho}$ such that

- $\mathbf{In}_d(\tilde{\varrho}) = \varepsilon$,
- $\mathbf{Out}_d(\varrho) \sqsubseteq \mathbf{Out}_d(\tilde{\varrho})$,
- for all $d' \neq d$, $\mathbf{In}_{d'}(\tilde{\varrho}) = \mathbf{In}_{d'}(\varrho)$

Proof. We proceed in the same way as in the previous part: the goal is now to use the partial runs ϱ'_j to eliminate the d-input of ϱ.

As ϱ is compatible with dec over d, we can split ϱ into $\varrho_0,\dots,\varrho_k$ with $w_i \sqsubseteq \mathbf{Out}_d(\varrho_i)$ and $\mathbf{In}_d(\varrho_i) \in \{m_1,\dots,m_i\}^*$ for all i. Again, we rename agents and data so that the sets of agents of ϱ and of every ϱ'_j are all disjoint and the only shared datum between any two of these runs is d.

We once again use the composition operation described in the proof of Lemma 3.34: consider ϱ and one of the ϱ'_j. We execute both runs in parallel and match each $\tilde{\varrho}_{i,j}$ with ϱ_j so that the broadcasts of ϱ_j with d forming v_i are received in $\tilde{\varrho}_{i,j}$, leaving only unmatched receptions with letters $m_1,\dots,m_i$. We obtain a run section whose d-output still contains v_i and whose d-input only contains $m_1,\dots,m_i$. We can do that until the next step in $\tilde{\varrho}_j$ is a broadcast of (m_j,d) and ϱ has been executed up to the beginning of ϱ_j. We may then use the (m_j,d) broadcast at any moment in the rest of ϱ to match an unmatched reception of ϱ. As a consequence, we can compose ϱ with the ϱ'_i as many times as necessary to obtain a run $\tilde{\varrho}$ with no unmatched receptions on d.

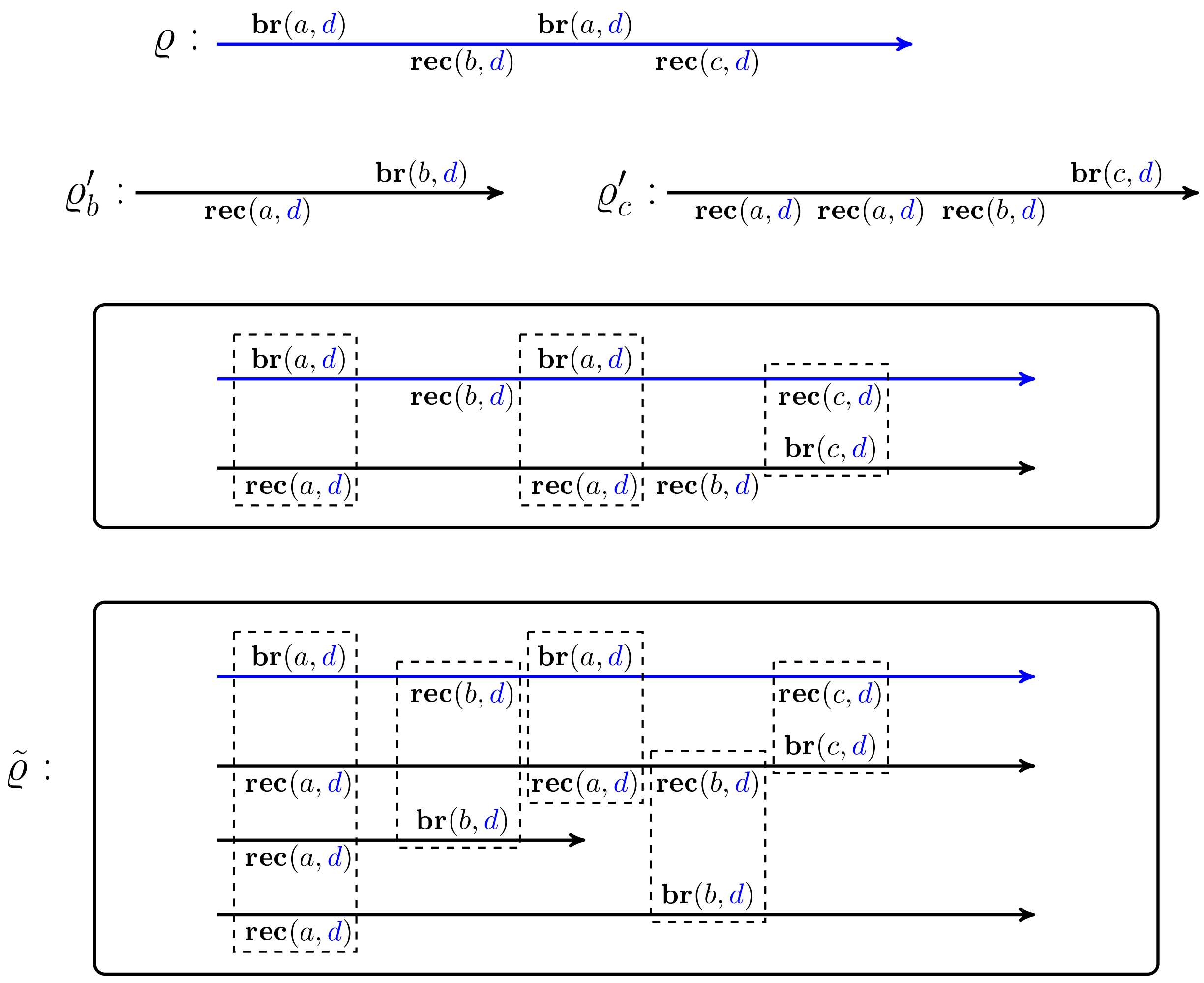


Figure 3.8: An illustration of the proof of Lemma 3.35. The partial run ϱ is compatible with decomposition $(a, b, a, c, \varepsilon)$. We have $\mathbf{In}_d(\varrho_b) = a \in \mathcal{L}_{(a)}$ and $\mathbf{In}_d(\varrho_b) = aab \in \mathcal{L}_{(a,b,a)}$. We build $\tilde{\varrho}$ such that $\mathbf{In}_d(\tilde{\varrho}) = \varepsilon$. We start by using ϱ'_c to eliminate the unmatched receptions of c (while adding some unmatched receptions of b), then we use ϱ'_b to eliminate the unmatched receptions of b. We ignore data other than d in this picture.

Each composition maintains the fact that the run is compatible with dec. When we do a composition with ϱ'_j to match a reception of (m_j, d), we may add some receptions of $m_1, \ldots, m_{j-1}$ to the run (the ones of ϱ'_j). However, every composition decreases the number of unmatched receptions of $m_k, \ldots, m_1$ for the lexicographic ordering.

As a result, in the end we obtain a run $\tilde{\varrho}$ without any unmatched reception on datum d. As ϱ is fully contained in $\tilde{\varrho}$, $\mathbf{Out}_d(\varrho) \sqsubseteq \mathbf{Out}_d(\tilde{\varrho})$. Moreover, for all $d' \neq d$, either d' does not appear in ϱ and then $\mathbf{In}_{d'}(\tilde{\varrho}) = \varepsilon$ or d' appears in ϱ and $\mathbf{In}_{d'}(\tilde{\varrho}) = \mathbf{In}_{d'}(\varrho)$ $\square$

Unmatched receptions with non-initial data

Lemma 3.36

Let ϱ be an initial σ-partial run, d' a datum, ϱ' an initial σ-run. If $\mathbf{In}_d(\varrho) \sqsubseteq \mathbf{Out}_{d'}(\varrho')$ and d' an initial datum value in ϱ' but not in ϱ, then there exists an initial σ-partial run $\tilde{\varrho}$ such that

- $\mathbf{In}_{d'}(\tilde{\varrho}) = \varepsilon$
- for all $d'' \neq d'$, $\mathbf{In}_{d''}(\tilde{\varrho}) = \mathbf{In}_{d''}(\varrho)$
- for all $d'' \neq d'$, $\mathbf{Out}_{d''}(\varrho) \sqsubseteq \mathbf{Out}_{d''}(\tilde{\varrho})$

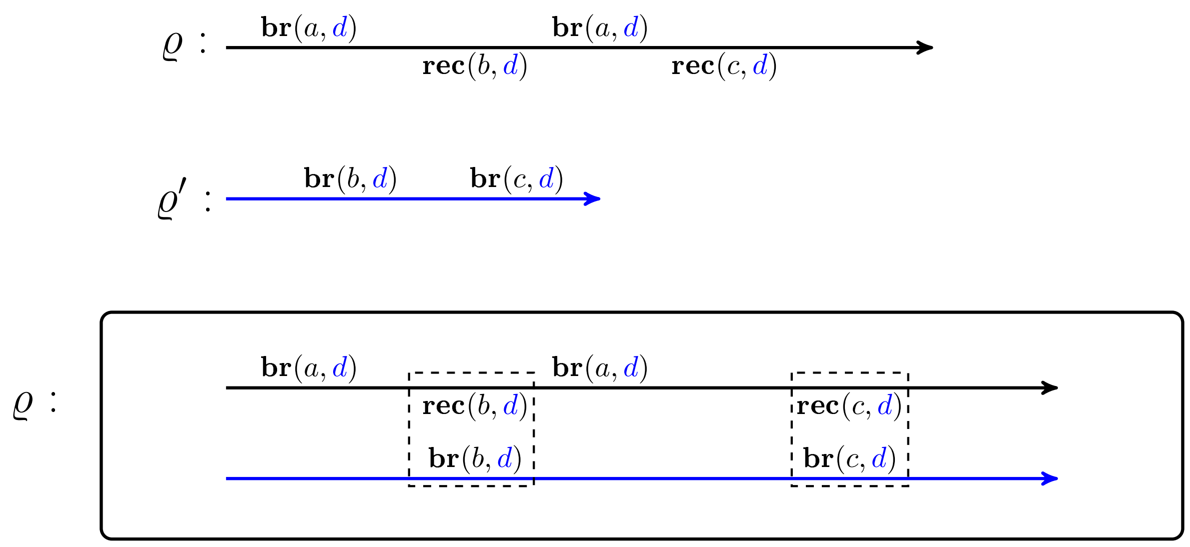


Figure 3.9: An illustration of the proof of Lemma 3.36. Datum d is initial in ϱ' and not ϱ. We ignore data other than d in this picture.

Proof. Up to renaming agents, assume that ϱ and ϱ' have disjoint agents. We rename data in ϱ' so that ϱ' has no shared data with ϱ besides d'.

We build $\tilde{\varrho}$ by running ϱ and ϱ' over their respective agents separately. We use the broadcasts made by ϱ' with d' to match the unmatched receptions with datum d' in ϱ: this gives us a new partial run ϱ with no unmatched reception with datum d'. Furthermore, for every datum d'', either the sequence of broadcasts and unmatched receptions is the same as before, or $\mathbf{In}_d(\varrho) = \varepsilon$ (if d'' appears in $\varrho_{d'}$).

The d''-output can only increase as ϱ is fully executed within $\tilde{\varrho}$. □

How to obtain a word $w \in \mathcal{L}(I, (J_m)_{m \in \mathcal{M}})$

We now combine Lemmas 3.34 and 3.35 to obtain one last useful technical lemma for the proof of Lemma 3.31. It will be used to prove the second condition of Definition 3.30 when showing that an invariant $(I, (J_m)_{m \in \mathcal{M}})$ is sufficient for a strategy σ.

Lemma 3.37

Let σ be a control strategy, I a downward-closed set of words, and $(J_m)_{m\in\mathcal{M}}$ upward-closed ones.
Suppose that for all $w \in I$ there is an initial σ-run and a datum d such that $w \sqsubseteq \mathbf{Out}_d(\varrho)$. Suppose also that for all $m \in \mathcal{M}$ and $w \in J_m$ there is a σ-partial run ϱ and a datum d that is not initial in ϱ such that $\mathbf{In}_d(\varrho) \sqsubseteq w$, $m \sqsubseteq \mathbf{Out}_d(\varrho)$ and $\mathbf{In}_{d'}(\varrho) = \varepsilon$ for all $d' \neq d$.
Then for all $w \in \mathcal{L}(I, (J_m)_{m\in\mathcal{M}})$, there is a σ-run ϱ and a datum d such that $w \sqsubseteq \mathbf{Out}_d(\varrho)$.

Proof. Let $w \in \mathcal{L}(I, (J_m)_{m\in\mathcal{M}})$, w matches a decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_k)$ such that $v_0 \cdots v_k \in I$ and, for all j, $\mathcal{L}_{(v_0,m_1,\dots,v_{j-1})} \cap J_{m_j} \neq \emptyset$. Hence we have a σ-run ϱ and a datum d such that $v_0 \cdots v_k \sqsubseteq \mathbf{Out}_d(\varrho)$. Note that as ϱ has no unmatched reception, in particular, it is compatible with dec. Also, for all j we have a σ-partial run ϱ_j and a datum d_j not initial in ϱ_j such that $\mathbf{In}_{d_j}(\varrho_j) \in \mathcal{L}_{(v_0,m_1,\dots,v_{j-1})}\downarrow$, $m_j \sqsubseteq \mathbf{Out}_{d_j}(\varrho_j)$ and $\mathbf{In}_{d'}(\varrho_j) = \varepsilon$ for all $d' \neq d_j$.

By Lemma 3.34, this means that we can obtain a σ-partial run whose d-output contains w, with no unmatched receptions on data other than d, and compatible with dec.

We can then use Lemma 3.35 to eliminate all unmatched receptions and obtain a σ-run whose d-output contains w. □

Proof of the characterisation lemma

Proof of Lemma 3.31. $\Longrightarrow$ Suppose σ is winning. Consider R the set of σ-runs.

Let $I = \{\mathbf{Out}_d(\varrho) \mid \varrho \in R, d \in \mathbb{D}\}\downarrow$ be the set of all outputs of all σ-runs.

For all $m \in \mathcal{M}$, we set J_m as the upward-closure of the set of $\mathbf{In}_d(\varrho)$ with ϱ a σ-partial run such that d is not an initial datum of ϱ, ϱ contains a broadcast of (m, d) and $\mathbf{In}_{d'}(\varrho) = \varepsilon$ for all $d' \neq d$.

Let us now prove that $(I, (J_m)_{m\in\mathcal{M}})$ is sufficient for σ. As σ is winning, m_{err} is never broadcast, and thus never received, in any σ-run. Hence $m_{err} \notin I$. Furthermore, if we had a word $w \in I \cap J_{m_{err}}$, then we would have a σ-run ϱ and a σ-partial run ϱ' such that m_{err} is broadcast in ϱ', $\mathbf{In}_{d'}(\varrho') \sqsubseteq w \sqsubseteq \mathbf{Out}_d(\varrho)$ and $\mathbf{In}_{d''}(\varrho') = \varepsilon$ for all $d'' \neq d'$. We can assume $d = d'$, as we can rename data.

As a result, we could form a σ-run by renaming data and agents such that their sets of data and agents are disjoint except for d. We then execute the two runs in parallel, and match the unmatched receptions of ϱ' with broadcasts in ϱ, to obtain a σ-run, with no unmatched receptions. This contradicts the fact that σ is winning. Hence $I \cap J_{m_{err}} = \emptyset$. Further, as an empty run is a σ-run, we have $\varepsilon \in I$.

For the second point, we can simply apply Lemma 3.37.

It remains to show that a σ-local run satisfying (i) and (ii) also satisfies (a) and (b). Let u be a σ-local run satisfying (i) and (ii).

- First, we construct a σ-run ϱ whose projection on some agent is u, which shows that u satisfies (a). Let d be the initial datum of u. As u satisfies (i), there is some $\mathsf{dec} = (v_0, m_1, \dots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})$ such that u is compatible with dec.

  By definition of $(J_m)_{m\in\mathcal{M}}$, for each j we have a σ-partial run ϱ_j and a non-initial datum d_j such that $\mathbf{In}_{d_j}(\varrho_j) \in \mathcal{L}_{(v_0,m_1,\dots,v_{j-1})}\downarrow$, there are no unmatched receptions with data other than d_j, and $m_j \sqsubseteq \mathbf{Out}_{d_j}(\varrho_j)$.

We can thus apply Lemma 3.35, to obtain a σ-partial run with no unmatched reception over d such that $\mathbf{Out}_d(u) \sqsubseteq \mathbf{Out}_d(\varrho)$.

Furthermore, as u satisfies (ii), by definition of I, for all $d' \neq d$ there is a σ-run $\varrho_{d'}$ such that $\mathbf{In}_{d'}(u) \sqsubseteq \mathbf{Out}_{d'}(\varrho_{d'})$. We can then apply Lemma 3.36 on ϱ, with every $d' \neq d$ appearing in u to obtain a σ-run ϱ' such that $\mathbf{Out}_d(u) \sqsubseteq \mathbf{Out}_d(\varrho')$. This shows that $\mathbf{Out}_d(u) \in I$, by definition.

- Let $d' \neq d$ and $m \in \mathcal{M}$ be such that (m, d') is broadcast in u. We can apply Lemma 3.35 on u and then Lemma 3.36 on the resulting run, with every $d'' \notin \{d, d'\}$. We obtain a σ-partial run in which (m, d') is broadcast and whose d'-input is the same as u. As a consequence, u satisfies (b) by definition of $(J_m)_{m\in\mathcal{M}}$.

This concludes the proof of that direction.

$\Leftarrow$ By Lemma 3.32.

□

3.5.2 The invariant game

We have characterised winning control strategies using invariants.The next step is to consider an invariant $(I, (J_m)_{m\in\mathcal{M}})$ and show that we can construct a game in which the two players determine whether there is a control strategy for which this invariant is sufficient.

To do so, we proceed in two steps, as in Section 3.4. First we consider a game played on $\mathcal{R}$ where players pick a sequence of transitions (the next transition is chosen by the player owning the current state), and Environment picks the data when needed. The goal of Environment is to eventually obtain a local run u that satisfies either (a) or (b) but neither (i) nor (ii), i.e., contradicting the invariant. The goal of Controller is to avoid this forever.

A key observation is that when a record transition or disequality transition is taken, it is always in Environment's best interest to choose a datum that was never seen before in the local run. Essentially, this is because in order to satisfy (b) without satisfying (ii) Environment wants the d'-input to be as small as possible. We will also see that the second condition in Lemma 3.31 makes it pointless for Environment to choose to receive its initial datum again after having forgotten it.

The *invariant game* $\mathcal{IG}(\mathcal{G}, I, (J_m)_{m\in\mathcal{M}})$ is defined as follows: The set of vertices is $Q_{\mathcal{R}}$: the current state in the protocol and a set of registers, which are the ones supposed to contain the initial datum. The initial vertex is q_{init}. From each vertex $q \in Q_{\mathcal{R}}$, players choose a transition from q in $\Delta_{\mathcal{R}}$. Controller chooses the next transition when q is in Q_{ctrl}, Environment when it is in Q_{env}. The state is updated to the target of the transition.

For all play π, we define $\mathtt{reg}(\pi)$ as the set of registers on which there were no record transition in π. Intuitively, $\mathtt{reg}(\pi)$ is the set of registers that contain the initial datum of the local run.

Given a play π, we define its *initial input* $\mathsf{initIn}(\pi)$ as the sequence of letters received with equality transitions with registers of $\mathtt{reg}$. This represents the sequence of letters received with the initial datum. Formally, $\mathsf{initIn}(\varepsilon) = \varepsilon$, and

$$\mathsf{initIn}(\delta_1 \cdots \delta_{k+1}) = \begin{cases} \mathsf{initIn}(\delta_1 \cdots \delta_k)m \text{ if } \delta_{k+1} \text{ is an equality transition } \xrightarrow{\mathbf{rec}(m,=i)} \\ \qquad\qquad\qquad \text{with } i \in \mathtt{reg}(\delta_1 \cdots \delta_k), \\ \mathsf{initIn}(\delta_1 \cdots \delta_k)\text{otherwise.} \end{cases}$$

For all registers i, we define its recent input on i, written $\mathsf{recentIn}_i(\pi)$ like in the previous section: it is the sequence of messages received with equality transitions over register i since its last reset.

We define the *output* $\mathsf{Out}(\pi)$ of π in a different way as in the signature case. It is the sequence of letters broadcast from registers that were in $\mathtt{reg}$ at the time of the broadcast. Intuitively, this is the sequence of letters that are broadcast with the initial datum in the local run. Formally,

$$\mathsf{Out}(\delta_1 \cdots \delta_{k+1}) = \begin{cases} \mathsf{Out}(\delta_1 \cdots \delta_k)m \text{ if } \delta_{k+1} \text{ is a broadcast transition } \xrightarrow{\mathbf{br}(m,i)} \\ \qquad\qquad\qquad \text{with } i \in \mathtt{reg}(\delta_1 \cdots \delta_k), \\ \mathsf{Out}(\delta_1 \cdots \delta_k) \text{ otherwise.} \end{cases}$$

Given a decomposition $\mathsf{dec} = (v_0, m_1, \ldots, v_k)$, we say that a play π is *compatible* with dec if $\pi = \pi_0 \cdots \pi_k$ and for all j we have $\mathsf{initIn}(\pi_j) \in \{m_1, \ldots, m_j\}^*$ and $v_j \sqsubseteq \mathsf{Out}(\pi_j)$.

The objective of the game is then described as follows.

(A) If at some point the play $\pi = \delta_1 \cdots \delta_k$ is not compatible with any decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$ then Controller wins.

(B) If at some point the play $\pi = \delta_1 \cdots \delta_k$ is such that $\mathsf{recentIn}_i(\pi) \notin I$ for some $i \notin \mathtt{reg}$ then Controller wins.

(C) If at some point we see a disequality transition receiving a letter $m \notin I$ then Controller wins.

(D) If at some point the play $\pi = \delta_1 \cdots \delta_k$ is such that $\mathsf{Out}(\pi) \notin I$ then Environment wins.

(E) If at some point of the play a broadcast transition with $i \notin \mathtt{reg}(\pi)$ and $\xrightarrow{\mathbf{br}(m,i)}$ is taken while $\mathsf{recentIn}_i(\pi) \notin J_m$ (with π the play formed so far) then Environment wins.

(F) If the play goes on forever without any of those things happening then Controller wins.

Lemma 3.38 ▶ Deciding the invariant game

There is an elementary function $\varphi(N)$ such that:
Given a protocol $\mathcal{R}$ and finite sets of words B and $(B_m)_{m\in\mathcal{M}}$, we can decide in time $\varphi(|\mathcal{R}| + ||B|| + |B| + \sum_{m\in\mathcal{M}} ||B_m|| + |B_m|)$ whether Controller has a winning strategy in $\mathcal{IG}(\mathcal{R}, (B\uparrow)^{\mathsf{c}}, (B_m\uparrow)_{m\in\mathcal{M}})$.
Furthermore, if Environment has a winning strategy then he has a strategy to win in at most $\varphi(|\mathcal{R}| + ||B|| + |B| + \sum_{m\in\mathcal{M}} ||B_m|| + |B_m|)$ steps.

Proof. By Lemma 3.7, $B\uparrow$ is a regular language, recognised by a deterministic finite automaton $\mathcal{A}_{B\uparrow} = (Q_B, \mathcal{M}, \Delta_B, q_0^B, F_B)$ with $(||B||+1)^{(|B|+1)}$ states. Similarly, for each m we can construct a deterministic automaton $\mathcal{A}_{B_m\uparrow} = (Q_{B_m}, \mathcal{M}, \Delta_{B_m}, q_0^{B_m}, F_{B_m})$

Let $I = B\uparrow^c$ and for each $m \in \mathcal{M}$, $J_m = B_m\uparrow$.

We define a deterministic automaton over the alphabet $\Delta_{\mathcal{R}}$ that reads plays $\delta_1 \cdots \delta_k$ of $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$ and accepts exactly the winning plays for Environment.

Consider the alphabet $\mathcal{M} \sqcup \bar{\mathcal{M}}$, where $\bar{\mathcal{M}} = \{\bar{m} \mid m \in \mathcal{M}\}$ is a copy of $\mathcal{M}$.

We define the useful automata in the following claims. Let us define $K = |\mathcal{R}| + |B| + ||B|| + \sum_{m\in\mathcal{M}} |B_m| + ||B_m||$

Claim 3.38.1. *We can construct an NFA of exponential size in K and recognising the language $\{v_0\bar{m}_1 \cdots v_k \mid (v_0, m_1 \cdots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})\}$.*

Proof of the claim.
Consider the language of decompositions defined as $\{v_0\bar{m}_1 \cdots v_{k-1}\bar{m}_k v_k \mid v_0, \ldots, v_k \in \mathcal{M}^*, \bar{m}_1, \ldots, \bar{m}_k \in \bar{\mathcal{M}}$ distinct.$\}$
This language is recognised by an automaton of exponential size which simply checks that each letter of $\bar{\mathcal{M}}$ appears at most once.
We can turn this automaton into a non-deterministic transducer $\mathcal{T}$ that reads a decomposition $v_0\bar{m}_1 \cdots v_{k-1}\bar{m}_k v_k$, outputs all the letters of $\mathcal{M}$ that it reads, and can output letters of $\bar{\mathcal{M}}$ arbitrarily as soon at it has read them before. If some letter of $\bar{\mathcal{M}}$ is repeated then the run is rejected. The set of images of $v_0\bar{m}_1 \cdots v_{k-1}\bar{m}_k v_k$ is exactly $\mathcal{L}_{(v_0, m_1, \ldots, v_k)}$.
By composing this transducer with an automaton recognising J_m, we obtain an automaton $\mathcal{A}_m$ recognising decompositions that have an image in J_m by the transducer, i.e., the language $\{v_0\bar{m}_1 \cdots v_{k-1}\bar{m}_k v_k \mid \mathcal{L}_{(v_0, m_1, \ldots, v_k)} \cap J_m \neq \emptyset\}$.
It is then easy to obtain an automaton recognising $\{v_0\bar{m}_1 \cdots v_k \mid (v_0, m_1 \cdots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})\}$ using a product of the automata $(\mathcal{A}_m)_{m\in\mathcal{M}}$.
The resulting automaton is of exponential size in K. ■

Claim 3.38.2. *We can construct a deterministic automaton of double-exponential size in K recognising plays compatible with a decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$.*

Proof of the claim. We use the automaton recognising $\{v_0\bar{m}_1 \cdots v_k \mid (v_0, m_1 \cdots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})\}$ defined in the first claim.
We can define a non-deterministic transducer that takes as input a sequence of transitions $\pi = \delta_1 \cdots \delta_p$ and outputs some decomposition with which it is compatible. The transducer keeps track of $\texttt{reg}(\pi)$ while reading the play.
The transducer simply guesses a sequence $\bar{m}_1 \cdots \bar{m}_k$ of distinct letters of $\bar{\mathcal{M}}$. It outputs them in that order at arbitrary moments while reading π.
When it reads a broadcast transition $\xrightarrow{\mathbf{br}(m,i)}$ over a register currently in $\texttt{reg}(\pi)$, it non-deterministically outputs m or not.
When it reads an equality transition $\xrightarrow{\mathbf{rec}(m,=i)}$ over a register currently in $\texttt{reg}(\pi)$, if $\bar{m}$ has not been broadcast before it goes to a rejecting sink state.
The set of images of a play π are the decompositions it is compatible with. We compose this transducer with the automaton from the first claim to get an automaton recognising the set of plays compatible with some decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$. ■

We have automata for I and each J_m, as well as for plays compatible with a decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$. From those it is straightforward to define an automaton $\mathcal{C}$ reading plays and accepting the ones winning for Environment. We can then determinise it at the cost of an exponential blow-up.

By Proposition 2.8, we can solve this game in polynomial time in the size of the resulting automaton $\mathcal{C}$ and the size of the arena of $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$ (i.e., $|\mathcal{R}|$), that is, in double-exponential time in $||B|| + |B| + |\mathcal{R}|$.

Furthermore, if Environment has a winning strategy then he has one that guarantees that he wins in at most double-exponentially many steps in K. ☐

We have to show that Controller wins the invariant game if and only if she has a winning control strategy.

Lemma 3.39 ▶ From the invariant game to control strategies

Let $I \subseteq \Sigma^*$ be a downward-closed set and $(J_m)_{m\in\mathcal{M}}$ upward-closed sets such that I contains ε and not m_{err}, $J_{m_{err}} \cap I = \emptyset$, and $\mathcal{L}(I, (J_m)_{m\in\mathcal{M}}) \subseteq I$.
If Controller wins the invariant game $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$ then there is a control strategy σ such that $(I, (J_m)_{m\in\mathcal{M}})$ is a sufficient invariant for σ.

Proof. Let $\sigma_{\mathcal{IG}}$ be a winning strategy for Controller in $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$.

We define σ as the control strategy in which Controller follows $\sigma_{\mathcal{IG}}$. That is, given a local run $u = (s_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_k(m_\ell,d_\ell)}_{\delta_\ell} (s_\ell, c_\ell)$, we set $\sigma(u) = \sigma_{\mathcal{IG}}(\delta_1 \cdots \delta_\ell)$. Let d be the initial datum of u. We show that $(I, (J_m)_{m\in\mathcal{M}})$ is a sufficient invariant for σ. To do so, we assume by contradiction that we have a σ-local run $u = (s_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_\ell(m_\ell,d_\ell)}_{\delta_\ell} (s_\ell, c_\ell)$ such that u satisfies (i) and (ii) but does not satisfy either (a) or (b). Let d be its initial datum.

We then show that $\pi = \delta_1 \cdots \delta_\ell$ is a losing $\sigma_{\mathcal{IG}}$-play for Controller in $\mathcal{IG}(\mathcal{R}, I)$. As u is a σ-local run, by definition of σ, $\delta_1 \cdots \delta_\ell$ is a $\sigma_{\mathcal{IG}}$-play.

For all $j \in [0, \ell]$ let u_j be the prefix of u up to (s_j, c_j) and $\pi_j = \delta_1 \cdots \delta_j$.

Claim 3.39.1. *For all index j and $i \notin \mathtt{reg}(\pi_j)$ we have $\mathsf{recentIn}_i(\pi_j) \sqsubseteq \mathbf{In}_{u_j}(c_j(i))$ and $\mathsf{initIn}(\pi_j) \sqsubseteq \mathbf{In}_{u_j}(d)$. Furthermore, if $\mathtt{reg}(\pi_j) \neq \emptyset$ then $\mathsf{Out}(\pi_j) = \mathbf{Out}_d(u_j)$.*

Proof of the claim. By a straightforward induction on j. ■

As u satisfies (i), it is compatible with a decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})$. We thus have $u = u^0 \cdots u^k$ with $v_i \sqsubseteq \mathbf{Out}_d(u^i)$ and $\mathbf{In}_d(u^i) \in \{m_1, \dots, m_i\}^*$ for all i.

Let j be the maximal index such that $\mathtt{reg}(\pi_j) \neq \emptyset$, and i_0 the maximal index such that $\pi^0 \cdots \pi^i$ is a prefix of π_j.

Hence we can cut π in the same way: $\pi = \pi^0 \cdots \pi^k$ where π^i is the sequence of transitions of u_i. We can infer using the previous claim that $v_j \sqsubseteq \mathbf{Out}_d(u^i) = \mathsf{Out}(\pi^i)$ for all $i \leq i_0$ and $\mathsf{initIn}(\pi^j) \sqsubseteq \mathbf{In}_d(u^i) \in \{m_1, \dots, m_i\}^*$.

As a consequence, π_j is compatible with $\mathsf{dec}' = (v_0, m_1, \dots, m_{i_0}, \varepsilon)$. Furthermore, we have $\mathsf{initIn}(\pi_j) = \mathsf{initIn}(\pi)$ and we can conclude that π is compatible with dec', which is in $\mathcal{D}((J_m)_{m\in\mathcal{M}})$.

We can also infer that all its prefixes π' are compatible with a decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$: it suffices to consider the decomposition $(v_0, m_1, \dots, m_i, \varepsilon)$, with i the maximal index such that m_i is appears in $\mathsf{initIn}(\pi')$.

Furthermore, as u satisfies (ii), for all $d' \neq d$, we have $\mathbf{In}_d(u_\ell) \in I$. For all j, $\mathsf{recentIn}_{\pi_j}(i) \sqsubseteq \mathbf{In}_{c_j(i)}(u_j) \sqsubseteq \mathbf{In}_{c_\ell(i)}(u_\ell)$. As I is downward-closed, we have $\mathsf{recentIn}_{\pi_j}(i) \in I$ for all $j \in [0, \ell]$.

Moreover, π cannot contain a disequality transition receiving a letter $m \notin \mathcal{L}(I, (J_m)_{m \in \mathcal{M}})\downarrow$: Otherwise u would also contain it and thus would not satisfy (ii).

We know that either u does not satisfy (a) or does not satisfy (b).

Let us first assume that u does not satisfy (b). Let $d' \neq d$ and $m \in \mathcal{M}$ be such that u contains a broadcast of (m, d') while $\mathbf{In}_{d'}(u) \notin J_m$. Let j be the index of the first broadcast of (m, d') in u and i the register containing d' at that point. Then δ_j is a broadcast transition $\xrightarrow{\mathbf{br}(m,i)}$, while $\mathsf{recentIn}_{\pi_j}(i) \sqsubseteq \mathbf{In}_{d'}(u_j) \sqsubseteq \mathbf{In}_{d'}(u)$. As J_m is upward-closed, $\mathsf{recentIn}_{\pi_j}(i) \notin J_m$, which means that π is losing for Controller.

Now we assume that u does not satisfy (a). Let $u = u_- u_+$ be such that u_- is the maximal prefix of u in which d appears at all times. We can cut $\pi = \pi_- \pi_+$ the same way: π_- is the sequence of transitions of u_-, and is also the maximal prefix of π such that $\mathtt{reg}(\pi_-) \neq \emptyset$.

Claim 3.39.2. *Suppose that π is a winning play for Controller. Then there is a decomposition* $\mathsf{dec} = (v_0, m_1, \dots, v_k) \in \mathcal{D}((J_m)_{m \in \mathcal{M}})$ *such that* $v_0 \cdots v_k \sqsubseteq \mathbf{Out}(\pi)$ *and* $\mathbf{Out}_d(u) \in \mathcal{L}_{\mathsf{dec}}$.

Proof. Let $\mathsf{dec} = (v_0, m_1, \dots, v_k)$ be a decomposition of $\mathcal{D}((J_m)_{m \in \mathcal{M}})$ such that u is compatible with dec. It exists as u satisfies (i).

Furthermore, we choose it so that $|v_0 \cdots v_k|$ is minimal. Among the ones with minimal $|v_0 \cdots v_k|$, we choose one with k maximal.

Suppose $v_0 \cdots v_k$ is not a subword of $\mathbf{Out}(\pi)$. Then, as $\mathbf{Out}(\pi) = \mathbf{Out}(\pi_-) = \mathbf{Out}_d(u_-)$, we get that $v_0 \cdots v_k \not\sqsubseteq \mathbf{Out}_d(u_-)$.

Let i be the minimal index such that $v_0 \cdots v_i \not\sqsubseteq \mathbf{Out}_d(u_-)$, and let v_{i-} be the maximal prefix of v_i such that $v_0 \cdots v_{i-1} v_{i-} \sqsubseteq \mathbf{Out}_d(u_-)$, and m the letter right after v_{i-} in v_i. Let v_{i+} such that $v_i = v_{i-} m v_{i+}$. The letter m must be broadcast with d in u_+. The same broadcast appears in π_+, say at step j on register i_0. As we assumed that π is winning, we have $\mathsf{recentIn}_{\pi_j}(i_0) \in J_m$. Hence $\mathbf{In}_d(u_j) \in J_m$, as J_m is upward-closed and $\mathsf{recentIn}_{\pi_j}(i_0) \sqsubseteq \mathbf{In}_d(u_j)$.

We have three cases:

- $m \in \{m_1, \dots, m_{i-1}\}$: then it is easily checked that we can remove m from v_i without affecting the properties of dec, contradicting the minimality of $|v_0 \cdots v_k|$.
- $m = m_\ell$ for some $\ell \in [i, k]$: then we can use the following decomposition: $(v_0, m_1, \dots, v_{i-1}, m_i, v_{i-}, m, v_{i+}, \dots, m_{\ell-1}, v_{\ell-1} v_\ell, m_{\ell+1}, \dots, v_k)$ instead of dec, again contradicting the minimality of $|v_0 \cdots v_k|$.
- $m \notin \{m_1, \dots, m_k\}$. Then we use the following decomposition instead of dec: $(v_0, m_1, \dots, v_{i-1}, m_i, v_{i-}, m, v_{i+}, m_{i+1}, \dots, v_k)$. This contradicts the minimality of $|v_0 \cdots v_k|$.

As a consequence, we obtain that $v_0 \cdots v_k$ is a subword of $\mathbf{Out}(\pi)$. It remains to show that $\mathbf{Out}_d(u) \in \mathcal{L}_{\mathsf{dec}}$. To do that, let us assume that u_+ contains a broadcast with d of a letter that is not in $\{m_1, \dots, m_k\}$. Let m be the letter in the first such broadcast of u_+, i the corresponding register, and j the index of the step. Since we assumed that π is winning, we have $\mathsf{recentIn}_{\pi_j}(i) \in J_m$. Hence $\mathbf{In}_d(u_j) \in J_m$, as J_m is upward-closed and

$\mathsf{recentIn}_{\pi_j}(i) \sqsubseteq \mathbf{In}_d(u_j)$. Moreover, every letter in $\mathbf{In}_d(u_j)$ must be in $\{m_1, \dots, m_k\}$, as u is compatible with dec.

As a result, $\mathbf{In}_d(u_j) \in J_m \cap \mathcal{L}_{\mathsf{dec}}$, hence $J_m \cap \mathcal{L}_{\mathsf{dec}} \neq$ and thus $(v_0, m_1, \dots, v_k, m, \varepsilon) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})$. Moreover, u is compatible with this decomposition. This contradicts the maximality of k.

In conclusion, we have shown that dec matches all the conditions of the claim. □

Suppose π is winning, then by this claim we have a decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_k) \in \mathcal{D}((J_m)_{m\in\mathcal{M}})$ such that $v_0 \cdots v_k \sqsubseteq \mathbf{Out}(\pi)$ and $\mathbf{Out}_d(u) \in \mathcal{L}_{\mathsf{dec}}$.

As π is winning, we have $\mathbf{Out}(\pi) \in I$, and thus $\mathbf{Out}_d(u) \in \mathcal{L}(I, (J_m)_{m\in\mathcal{M}})$. Since $\mathcal{L}(I, (J_m)_{m\in\mathcal{M}}) \subseteq I$, we get $\mathbf{Out}_d(u) \in I$, and thus u satisfies (a), a contradiction.

In conclusion, we obtained that π is a losing $\sigma_{\mathcal{IG}}$-play, which contradicts the assumption that $\sigma_{\mathcal{IG}}$ is winning. As a consequence, $(I, (J_m)_{m\in\mathcal{M}})$ is a sufficient invariant for σ.¨ □

Lemma 3.40 ▶ From control strategies to the invariant game

Let σ be a control strategy.
Let $I \subseteq \Sigma^*$ be a downward-closed set and $(J_m)_{m\in\mathcal{M}}$ upward-closed sets such that I contains ε and not m_{err}, $J_{m_{err}} \cap I = \emptyset$, and $\mathcal{L}(I, (J_m)_{m\in\mathcal{M}})) \subseteq I$.
Let B be the basis of I^c and B_m the basis of J_m for all m.
If Environment wins the invariant game $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$ then there is a σ-local run of length at most $\varphi(|\mathcal{R}| + |B| + ||B|| + \sum_{m\in\mathcal{M}} |B_m| + ||B_m||)$ satisfying (i) and (ii) and dissatisfying either (a) or (b).

Proof. Let $N = |\mathcal{R}| + |B| + ||B|| + \sum_{m\in\mathcal{M}} |B_m| + ||B_m||$. By Lemma 3.38 and Proposition 2.8 there exists $\tau_{\mathcal{IG}}$ a winning strategy $\tau_{\mathcal{IG}}$ for Environment in the invariant game $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m\in\mathcal{M}})$ such that Environment always wins in at most $\varphi(N)$ steps. We construct a σ-local run of length at most $\varphi(N)$ satisfying (i) and (ii) and dissatisfying either (a) or (b). To do so, we apply $\tau_{\mathcal{IG}}$ to choose transitions and we choose data by always picking a datum never seen before in the run, when the datum is not determined by the transition. Let (s_0, c_0) be an initial configuration of $\mathcal{R}$. We define iteratively a sequence of steps $(s_{\ell-1}, c_{\ell-1}) \xrightarrow{\mathbf{op}_\ell(m_\ell, d_\ell)}_{\delta_\ell} (s_\ell, c_\ell)$ as follows. Suppose we defined them up to $(s_{\ell-1}, c_{\ell-1})$, and let $u_{\ell-1}$ be the local run defined so far. We first choose δ_ℓ:

- If $s_{\ell-1} \in Q_{\mathsf{ctrl}}$ then $\delta_\ell = \sigma(\delta_1 \cdots \delta_{\ell-1})$,
- otherwise $\delta_\ell = \tau_{\mathcal{IG}}(\delta_1 \cdots \delta_{\ell-1})$.

We then choose d_ℓ:

- If δ_ℓ is a broadcast transition of letter m, we set d_ℓ as the initial datum of the local run.
- If δ_ℓ is a record transition or a disequality transition, we pick a datum d_k that does not appear in $u_{\ell-1}$.
- If $\delta_\ell = s_{\ell-1} \xrightarrow{\mathbf{rec}(m,=i)} s_\ell$ is an equality transition of letter m, we set $d_\ell = c_{\ell-1}(i)$.

Clearly we maintain the fact that u_ℓ is a σ-local run and $\delta_1 \cdots \delta_\ell$ is a $\tau_{\mathcal{IG}}$-play in $\mathcal{IG}(\mathcal{R}, I)$. We stop when $\delta_1 \cdots \delta_\ell$ is winning for Environment in $\mathcal{IG}(\mathcal{R}, I, (J_m)_{m \in \mathcal{M}})$, which happens for some $\ell \leq \varphi(N)$. Let M be the final value of ℓ and $u = u_M$ be the local run obtained at the end. Let d be its initial datum. It remains to show that u satisfies (i) and (ii) and dissatisfies either (a) or (b). To do so, we rely on the following claim:

Claim 3.40.1. *For all register i and index ℓ such that $i \notin \mathtt{reg}(\delta_1 \cdots \delta_\ell)$, $\mathsf{recentIn}_i(\delta_1 \cdots \delta_\ell) = \mathbf{In}_{u_\ell}(c_\ell(i))$. Furthermore, $\mathsf{Out}(\delta_1 \cdots \delta_\ell) = \mathbf{Out}_d(u_\ell)$ and $\mathsf{initIn}(\delta_1 \cdots \delta_\ell) = \mathbf{In}_d(u_\ell)$.*

Proof of the claim. By a straightforward induction on ℓ. ■

Let $\pi_\ell = \delta_1 \cdots \delta_\ell$ for all ℓ, and let $\pi = \pi_M$.

First we show that u satisfies (i): As π is winning for Environment, it is compatible with some decomposition $\mathsf{dec} = (v_0, m_1, \ldots, v_k) \in \mathcal{D}((J_m)_{m \in \mathcal{M}})$. Thus $\pi = \pi^0 \cdots \pi^k$ with $v_j \sqsubseteq \mathsf{Out}(\pi^j)$ and $\mathsf{initIn}(\pi^j) \in \{m_1, \ldots, m_j\}^*$, for all j.

We divide u like π, $u = u^0 \cdots u^k$. As a consequence of the claim, we obtain $v_j \sqsubseteq \mathbf{Out}_d(u^j)$ and $\mathbf{In}_d(u^j) \in \{m_1, \ldots, m_j\}^*$, for all j. Thus u is compatible with dec.

Now, we show that u satisfies (ii). Let $d' \neq d$. If d' is stored in a register at some point in u, let ℓ be the maximal index such that $c_\ell(i) = d'$ for some i. There can be no step involving d' after ℓ, as d' would need to be stored in a register, contradicting the maximality of ℓ, or received using a disequality transition, which is impossible as we constructed u so that those are always made with fresh data. As a consequence, $\mathbf{In}_{d'}(u) = \mathsf{recentIn}_{\pi_\ell}(i)$. As π is winning for Environment, we have $\mathsf{recentIn}_{\pi_\ell}(i) \in I$. If $\mathbf{In}_{d'}(u) = \varepsilon$ then clearly $\mathbf{In}_{d'}(u) \in I$ by assumption on I. If d' is never stored in a register and $\mathbf{In}_{d'}(u) \neq \varepsilon$, the only possibility is that d' is received once with a disequality transition with letter m. In that case, since π is winning for Environment, we have $\mathbf{In}_{d'}(u) = m \in I$

We have shown that u satisfies (i) and (ii).

If $\mathsf{Out}(\pi) \notin I$ then $\mathbf{Out}_d(u) \notin I$, by the claim, hence u does not satisfy (a).

If $\mathsf{Out}(\pi) \in I$, since π is winning for Environment, there must be an index ℓ such that the ℓth transition of π is a broadcast transition $\xrightarrow{\mathbf{br}(m,i)}$, but $\mathsf{recentIn}_\pi(i) \notin J_m$. In that case, we have $\mathbf{In}_{c_{\ell+1}(i)}(u_{\ell+1}) \notin J_m$ and $u_{\ell+1}$ contains a broadcast of $(m, c_{\ell+1}(i))$.

As a consequence, we have found a prefix $u_{\ell+1}$ of u which does not satisfy (b). As (i) and (ii) hold for u, it is easy to see that they must also hold for all its prefixes.

In all cases we have found a σ-local run of length at most $\varphi(N)$ which satisfies (i) and (ii) but dissatisfies either (a) or (b).

This concludes our proof. □

Lemma 3.41 ▶ Bounding invariants

There is an elementary function $\psi(N)$ such that the following statement holds. Let $\mathcal{G}$ a BGR. There is a winning control strategy for $\mathcal{G}$ if and only if there is a sequence of words $w_1, \dots, w_k \in \mathcal{M}^*$ and subsets $B, (B_m)_{m \in \mathcal{M}}$ of $\{w_1, \dots, w_k\}$ such that

- B contains m_{err} and not ε and $B_{m_{err}}\uparrow \cap B\uparrow^{\mathsf{c}} = \emptyset$
- $\mathcal{L}(B\uparrow^{\mathsf{c}}, (B_m\uparrow)_{m\in\mathcal{M}}, \subseteq)B\uparrow^{\mathsf{c}}$
- Controller wins $\mathcal{IG}(\mathcal{G}, B\uparrow^{\mathsf{c}}, (B_m\uparrow)_{m\in\mathcal{M}})$,
- B and all B_m are antichains for the subword order $\sqsubseteq$,
- $B \cup \bigcup_{m\in\mathcal{M}} B_m = \{w_1, \dots, w_k\}$,
- for all $i \in [1, k]$, $|w_i| \leq \psi(|w_{i-1}|)$.

Proof. By Lemma 3.39, if there are such sets of words B and $(B_m)_{m\in\mathcal{M}}$, then there is a control strategy such that $(B\uparrow^{\mathsf{c}}, (B_m\uparrow)_{m\in\mathcal{M}}$ is a sufficient invariant for σ. Hence, by Lemma 3.31, σ is a winning control strategy.

Conversely, suppose there is a winning control strategy σ. By Lemma 3.31 there is a sufficient invariant $(I, (J_m)_{m\in\mathcal{M}})$ for σ. As I^{c} is upward-closed it has a finite basis B. Similarly, each J_m has a finite basis B_m.

The first two conditions hold by definition, as $(I, (J_m))$ is a sufficient invariant.

By Lemma 3.40 Controller wins $\mathcal{IG}(\mathcal{G}, I, (J_m)_{m\in\mathcal{M}})$, so the third condition of the lemma is satisfied.

For the third condition, by definition, all basis are antichains.

Let $w_0, w_1, ..., w_k$ be the elements of $B \cup \bigcup_{m\in\mathcal{M}} B_m$ sorted by length, i.e., $|w_i| \leq |w_{i+1}|$ for all i. We can assume that we chose I and $(J_m)_{m\in\mathcal{M}}$ so that k would be minimal.

By minimality of k, for all $j \in [1, k]$, $(B', (B'_m)_{m\in\mathcal{M}})$ is not a sufficient invariant for σ, with $B' = B \cap \{w_i \mid i < j\}$ and for all m, $B'_m = B_m \cap \{w_i \mid i < j\}$. Let $I' = B'\uparrow^{\mathsf{c}}$ and $J'_m = B'_m\uparrow$ for all m. Note that $I \subseteq I'$ while $J'_m \subseteq J_m$ for all m.

A possibility is that $m_{err} \in I'$. As $m_{err} \in B$, we then have $|w_j| \leq 1$.

Another possibility is that $I' \cap J'_{m_{err}} \neq \emptyset$. As a consequence, there is a word $w \in B'_{m_{err}}$ with no subword in B'. As this word is of length at most $|w_{j-1}|$, we conclude that there is a word of length at most $|w_{j-1}|$ in $B \setminus B'$, hence $|w_j| \leq |w_{j-1}|$.

Thirdly, we may have $\mathcal{L}(I', (J'_m)_{m\in\mathcal{M}}) \nsubseteq I'$. Then there is a decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_k) \in \mathcal{D}(I', (J'_m)_{m\in\mathcal{M}})$ and $w \in \mathcal{L}_{\mathsf{dec}}$ such that $w \notin I'$.

It is easy to construct deterministic automata recognising $\mathcal{L}(I', (J'_m)_{m\in\mathcal{M}})$ and I' of double-exponential size in $|\mathcal{R}|$, $|\mathcal{M}|$, B' and $(B'_m)_{m\in\mathcal{M}}$, by using Lemma 3.7 and Claim 3.38.1.

Hence we can find such a w of at most double-exponential size, and thus the decomposition $\mathsf{dec} = (v_0, m_1, \dots, v_k)$ also has at most double-exponential size. Now note that $v_0 \cdots v_k$ is in I', but cannot be in I: otherwise, we would have $w \in \mathcal{L}(I, (J'_m)_{m\in\mathcal{M}}) \subseteq \mathcal{L}(I, (J_m)_{m\in\mathcal{M}})$, while $w \notin I' \supseteq I$, a contradiction. Hence there is a word of at most double-exponential size in $|\mathcal{G}|$, B' and $(B'_m)_{m\in\mathcal{M}}$ (thus of at most triple-exponential size in $|w_{j-1}|$) that is in B' but not B. As a consequence, $|w_j|$ is at most triply-exponential in $|w_{j-1}| + |\mathcal{M}| + |\mathcal{R}|$.

The last case is when there is a run σ-local run which satisfies (i) and (ii) but dissatisfies either (a) or (b), with respect to $(I', (J'_m)_{m\in\mathcal{M}})$. By Lemma 3.40, there is such a σ-local run u of length at most $K = \varphi(|\mathcal{R}| + ||B'|| + \sum_{m\in\mathcal{M}} ||B'_m||) \leq \varphi(|\mathcal{R}| + (|\mathcal{M}|+1)|w_{j-1}|)$.

As $J'_m \subseteq J_m$ for all m, we have $\mathcal{D}((J'_m)_{m\in\mathcal{M}}) \subseteq \mathcal{D}((J_m)_{m\in\mathcal{M}})$. As a consequence, u satisfies (i) with respect to $(I, (J_m)_{m\in\mathcal{M}})$.

As $I \subseteq I'$, if u satisfies (a) with respect to $(I, (J_m)_{m\in\mathcal{M}})$ then it also satisfies it with respect to $(I', (J'_m)_{m\in\mathcal{M}})$.

Two cases remain: either u satisfies (ii) with respect to $(I', (J'_m)_{m\in\mathcal{M}})$ and not $(I, (J_m)_{m\in\mathcal{M}})$, or satisfies (b) with respect to $(I, (J_m)_{m\in\mathcal{M}})$ and not $(I', (J'_m)_{m\in\mathcal{M}})$.

We examine the two cases:

- Suppose u satisfies (ii) with respect to $(I', (J'_m)_{m\in\mathcal{M}})$ and not $(I, (J_m)_{m\in\mathcal{M}})$. Let d' be a datum such that $\mathbf{In}_{d'}(u) \notin I$. As $\mathbf{In}_{d'}(u) \in I'$, we found a word of length at most K that is in I' but not I.

- Suppose u satisfies (b) with respect to $(I, (J_m)_{m\in\mathcal{M}})$ and not $(I', (J'_m)_{m\in\mathcal{M}})$. Then there exist $m \in \mathcal{M}$ and $d' \neq d$ such that u contains a broadcast of (m, d') and $\mathbf{In}_{d'}(u) \notin J'_m$, while $\mathbf{In}_{d'}(u) \in J_m$. Furthermore, we have $|\mathbf{In}_{d'}(u)| \leq |u| \leq K$

In both cases, there exists w_ℓ with $\ell \geq j$ such that $w_\ell \sqsubseteq w$, and thus $|w_\ell| \leq |w| \leq K$.

As $|w_j| \leq |w_\ell|$, we have $|w_j| \leq K$. As $||B'|| \leq |w_{j-1}|$ and $||B'_m|| \leq |w_{j-1}|$, we obtain $|w_j| \leq \varphi(|\mathcal{R}| + (|\mathcal{M}|+1)|w_{j-1}|)$.

We can then simply take a suitable elementary function so that $|w_{j-1}| \leq \psi(|\mathcal{R}| + |\mathcal{M}| + |w_j|)$ □

Theorem 3.42 ▶ Main theorem of this chapter

SafeStrat is decidable and in $\mathbf{F}_{\omega^\omega}$ for arbitrary BGR.

Proof. Let $\mathcal{G}$ a BGR. We once again apply the Length Function Theorem.

Consider a sequence of words $w_1, \dots, w_k \in \mathcal{M}^*$ and subsets $B, (B_m)_{m\in\mathcal{M}}$ of $\{w_1, \dots, w_k\}$ satisfying the conditions of Lemma 3.41.

We use a fresh letter $\# \notin \mathcal{M}$. For each w_i we define $w'_i = \#^{|\mathcal{R}|+|\mathcal{M}|} w_i \#\#$ if $w_i \in B$, and $w'_i = \#^{|\mathcal{R}|+|\mathcal{M}|} w_i \# m$ with m such that $w_i \in B_m$ otherwise.

By the second condition of Lemma 3.41, w'_i is well-defined for all i.

Note that the sequence $w'_1 \cdots w'_k$ is an antichain: as $\#$ does not appear in any w_i, $w'_i \sqsubseteq w'_j$ implies that $w_i \sqsubseteq w_j$, and that they both belong to B or to some common B_m. This is impossible as all those sets are antichains.

Furthermore, for all i, we have $|w'_{i+1}| \leq \psi(|\mathcal{R}| + |\mathcal{M}| + |w_i|) + |\mathcal{R}| + |\mathcal{M}| + 2 \leq \psi(|w'_i|) + |w'_i|$. As $g : n \mapsto \psi(n) + n$ is a primitive recursive function, by the Length function theorem we obtain a function $f \in \mathscr{F}_{\omega^{|\mathcal{M}|}}$ such that every (g, n)-controlled bad sequence of words $w_0, w_1, ..., w_k$ has at most $f(n)$ terms.

As m_{err} is in B, $|w_0| \leq 1$, thus $|w'_0| \leq |\mathcal{R}| + |\mathcal{M}| + 3$ We therefore have $|w_i| \leq g^{(i)}(|\mathcal{R}| + |\mathcal{M}| + 3)$ for all i. As a consequence, we have $k \leq f(|\mathcal{R}| + |\mathcal{M}| + 3)$.

Our algorithm guesses a sequence of words of sorted by length $w_1, ..., w_k$ with $k \leq f(|\mathcal{R}| + |\mathcal{M}| + 3)$ such that $|w_{i+1}| \leq \psi(|w_i|)$ for all i. The algorithm then guesses subsets B and $(B_m)_{m\in\mathcal{M}}$ that cover $\{w_i \mid i \in [1,k]\}$.

It checks that Controller wins $\mathcal{IG}(\mathcal{R}, B\!\uparrow^{\mathsf{c}}, (B_m\!\uparrow)_{m\in\mathcal{M}},)$. We accept if she does and reject otherwise.

This can be done in double-exponential time in $|\mathcal{R}|+k+|w_k|$, by Lemma 3.38. We can make this algorithm deterministic with an exponential blow-up in the time complexity. By Lemma 3.41, this algorithm is correct.

The time required by this algorithm is therefore $h(f(|\mathcal{R}|+|\mathcal{M}|+3))$ with h a primitive recursive function. As $\mathscr{F}_{\omega^{|\mathcal{M}|}}$ is closed under composition with primitive recursive functions, the algorithm takes a time bounded by a function of $\mathscr{F}_{\omega^{|\mathcal{M}|}}$. As a consequence, the problem is in $\mathbf{F}_{\omega^\omega}$. □

3.6 Lower bounds

In this section we prove that the coverability and safe strategy synthesis problems are $\mathbf{F}_{\omega^\omega}$-complete, even when restricted to signature protocols.

The upper bounds were proven in Section 3.5. In this section we show that the COVER problem is $\mathbf{F}_{\omega^\omega}$-hard for signature BNRA. We obtain the hardness for SAFESTRAT as an immediate corollary.

Along with the upper bound obtained in the previous section (Proposition 3.28), this yields the following results.

Theorem 3.43

COVER and SAFESTRAT are $\mathbf{F}_{\omega^\omega}$-complete for (signature) BGR.

We contrast these decidability results with an undecidability one: the target reachability problem is undecidable, even on signature BGR with only two registers, as stated in Theorem 3.45.

3.6.1 Coverability lower bound

Proposition 3.44

COVER for signature BNRA is $\mathbf{F}_{\omega^\omega}$-hard.

Proof. We reduce from the reachability problem for lossy channel systems. Our reduction is computable in polynomial-time, proving the proposition.

Let $\mathcal{S} := (L, \Sigma, T, l_{init})$ be a lossy channel system and $l_f \in L$.

We construct a signature protocol $\mathcal{R}$ with two registers: all broadcast transitions are on register 1 and all reception transitions on register 2. It is such that:

- $\mathcal{R}$ makes sure that all received messages have the same datum,
- $\mathcal{R}$ only broadcasts messages with its initial datum

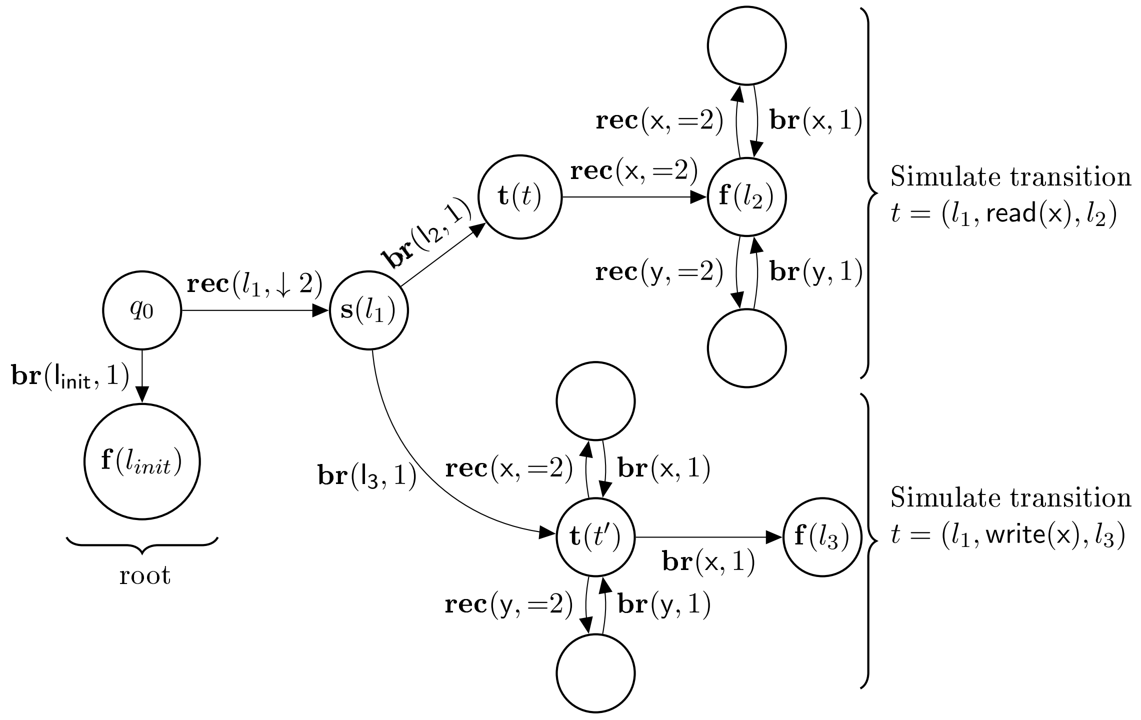
Intuitively, we simulate an LCS $\mathcal{S}$ by making each agent receive a configuration of $\mathcal{S}$ from another agent and broadcast a successor configuration. The fact that some broadcasts may not be received simulates the lossiness of the LCS.

In some initial phase, each agent may become either a *root* or a *link*. If it becomes a root then it only broadcasts and never receives messages. If it is a link, it stores some other agent's identifier (its *predecessor*); in that case, it will test further messages for equality so that only messages from the predecessor are accepted.

A *root* agent simply broadcasts the initial state l_s of $\mathcal{S}$ and stops. A *link* agent a receives from another agent (its *predecessor*) a location l of the system and a sequence of letters w, i.e., a channel content. Agent a then picks a transition of $\mathcal{S}$ from l and broadcast the new location of the system and the new content of the channel which a rebroadcasts on-the-fly letter by letter as it receives it. Agent a only removes a letter at the beginning of w to encode a read and adds a letter at the end to encode a write.

Finally, the error state of our system is $q_{err} := \mathbf{f}(l_f)$.

See Figure 3.10 for an example of protocol obtained with this reduction.



q_0
$\mathbf{rec}(l_1, \downarrow 2)$
$\mathbf{s}(l_1)$
$\mathbf{br}(\mathsf{l}_2, 1)$
$\mathbf{t}(t)$
$\mathbf{rec}(\mathsf{x}, =2)$
$\mathbf{f}(l_2)$
$\mathbf{rec}(\mathsf{x}, =2)$
$\mathbf{br}(\mathsf{x}, 1)$
$\mathbf{rec}(\mathsf{y}, =2)$
$\mathbf{br}(\mathsf{y}, 1)$
Simulate transition
$t = (l_1, \mathsf{read}(\mathsf{x}), l_2)$
$\mathbf{br}(\mathsf{l_{init}}, 1)$
$\mathbf{f}(l_{init})$
root
$\mathbf{br}(\mathsf{l}_3, 1)$
$\mathbf{rec}(\mathsf{x}, =2)$
$\mathbf{br}(\mathsf{x}, 1)$
$\mathbf{t}(t')$
$\mathbf{f}(l_3)$
$\mathbf{br}(\mathsf{x}, 1)$
$\mathbf{rec}(\mathsf{y}, =2)$
$\mathbf{br}(\mathsf{y}, 1)$
Simulate transition
$t = (l_1, \mathsf{write}(\mathsf{x}), l_3)$

Figure 3.10: A depiction of the protocol $\mathcal{R}$ built in the lossy channel system reduction of Proposition 3.44.

We claim that q_{err} is coverable in $\mathcal{R}$ if and only if l_f is reachable in $\mathcal{S}$.

First, suppose that there exists a run of $\mathcal{S}$ $(l_0, w_0) \rightsquigarrow_{\mathcal{S}} (l_1, w_1) \rightsquigarrow_{\mathcal{S}} (l_2, w_2) \cdots \rightsquigarrow_{\mathcal{S}} (l_n, w_n)$ with $l_n = l_f$. We build an initial run of $\mathcal{R}$ that covers q_{err} as follows. More precisely, for all $i \in [0, n]$, we build a run ϱ_i with $i+1$ agents $a_0, \dots, a_n$ in which agent a_i outputs $l_i w_i'$ with $w_i \sqsubseteq w_i'$.

For $i = 0$, agent a_0 becomes the root and outputs l_0. Suppose we constructed ϱ_i, we construct ϱ_{i+1}. We make agents $a_0, \dots, a_i$ execute ϱ_i, while agent a_{i+1} behaves as follows. It receives from agent i state l_i and goes to $\mathbf{s}(l_i)$. Let $t \in T$ be such that $(l_i, w_i) \overset{t}{\rightsquigarrow}_{\mathcal{S}} (l_{i+1}, w_{i+1})$.

- if $t = (l_i, \mathsf{write}(x), l_{i+1})$ is a write, then $w_{i+1} \sqsubseteq w_i \cdot x$.

  Agent a_{i+1} moves to $\mathbf{t}(t)$ and broadcasts l_{i+1}. It then receives the sequence of letters w_i from a_i (which is possible as w_i is a subword of the output of a_i) and rebroadcasts it while looping on $\mathbf{t}(t)$. It then broadcasts x to get to $\mathbf{f}(l_{i+1})$.

- if $t = (l_i, \mathsf{read}(x), l_{i+1})$ is a read then $x \cdot w_{i+1} \sqsubseteq w_i$. Agent a_{i+1} moves to $\mathbf{t}(t)$ and broadcasts l_{i+1}. It then receives $x \cdot w_{i+1}$ from a_i. Upon receiving x it goes to $\mathbf{f}(l_{i+1})$ and then receives and broadcasts each letter of w_{i+1}.

This concludes the induction step. When applied to $i = n$, this builds an initial run where agent n ends on $\mathbf{f}(l_n) = q_{err}$, which is a witness that q_{err} is coverable.

Suppose now that q_{err} is coverable. Let $\varrho : \gamma_0 \xrightarrow{*} \gamma_f$ where γ_f covers q_{err}. Let a_0 be an agent in state q_{err} in γ_f. We build a sequence of agents by defining a_{i+1} as the predecessor of a_i if it has one. This sequence is finite as there are finitely many agents and the predecessor of each agent sends its first broadcast before that agent. We thus get a subset of agents $a_0, \dots, a_n$, with a_n a root agent.

For each $i \geq 1$, let w_i be the word of letters By structure of the protocol, every agent a_i with $i \geq 1$ receives a word $l_i w_i$. It then suffices to analyse the behaviour of each a_{i+1} to prove that $(l_{i+1}, w_{i+1}) \rightsquigarrow_{\mathcal{S}} (l_i, w_i)$. In particular, because a_n is a root, $l_n = l_{init}$ and $w_n = \varepsilon$. We can then define $l_0 = l_f$ and w_0 such that a_0 outputs $l_0 w_0$, and observe that $(l_1, w_1) \rightsquigarrow_{\mathcal{S}} (l_0, w_0)$. This concludes the proof. □

3.6.2 Undecidability of target reachability

The decidability of the coverability problem immediately leads us to ask whether some other reachability problems are decidable on this model. We show in this section that there is little hope to find in that direction.

Theorem 3.45

Synchro is undecidable for (signature) BNRA.

Proof. We simulate a Minsky machine with two counters. As in Proposition 3.44, at the start each agent stores some other agent's identifier, called its "predecessor". It then only accepts messages from its predecessor. As there are finitely many agents, there is a cycle in the predecessor graph. In a cycle, we use the fact that all agents must reach state q_{err} to simulate faithfully a run of the machine: agents alternate between receptions and broadcasts so that, in the end, they have received and sent the same number of messages, implying that no message has been lost along the cycle. We then simulate the machine by having an agent (the leader) choose transitions and the other ones simulate the counter values by memorizing a counter (1 or 2) and a binary value (0 or 1). For instance, an increment of counter 1 takes the form of a message propagated in the cycle from the leader until it finds an agent simulating counter 1 and having bit 0. This agent switches to 1 and sends an acknowledgment that propagates back to the leader.

Let us now give a more formal proof. We reduce from the halting problem for Minsky machines.

Let $M = (\mathrm{Loc}, \Delta, \mathtt{X}, \ell_0, \ell_f)$ be a Minsky Machine. We build a signature protocol $\mathcal{R}$ with a state q_{err} such that $(\mathcal{R}, q_{err})$ is a positive instance of Synchro if and only if ℓ_f is coverable in M. As in the proof of Proposition 3.44, in an initial phase, each agent broadcasts its identifier and picks a predecessor by storing its identifier in register 2. Each agent only listens to its predecessor afterwards: it only receives messages with equality transitions $\xrightarrow{\mathbf{rec}(m,=2)}$, to check that they all come from its predecessor.

We call *cycle* a sequence of agents $a_1, a_2, \dots, a_n = a_1$ where agent a_i is the predecessor of a_{i+1} for all $i < n$. We allow each agent to make the previous broadcast and reception in any order, as otherwise cycles would never form. As all agents have to reach the end state, they must all pick a predecessor. As there are finitely many agents in a run, a cycle

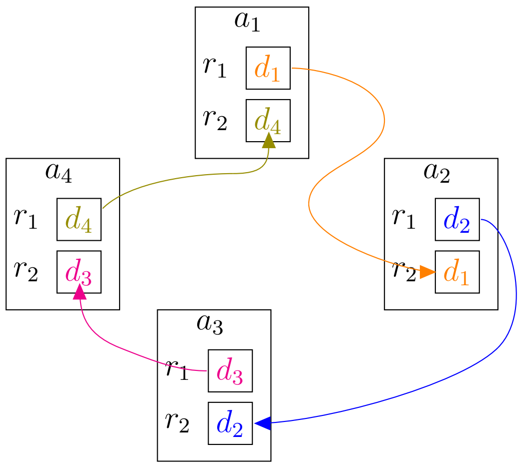


Figure 3.11: Cycle of 4 agents storing their prdecessors' identifiers. Each box represents an agent along with its register values.

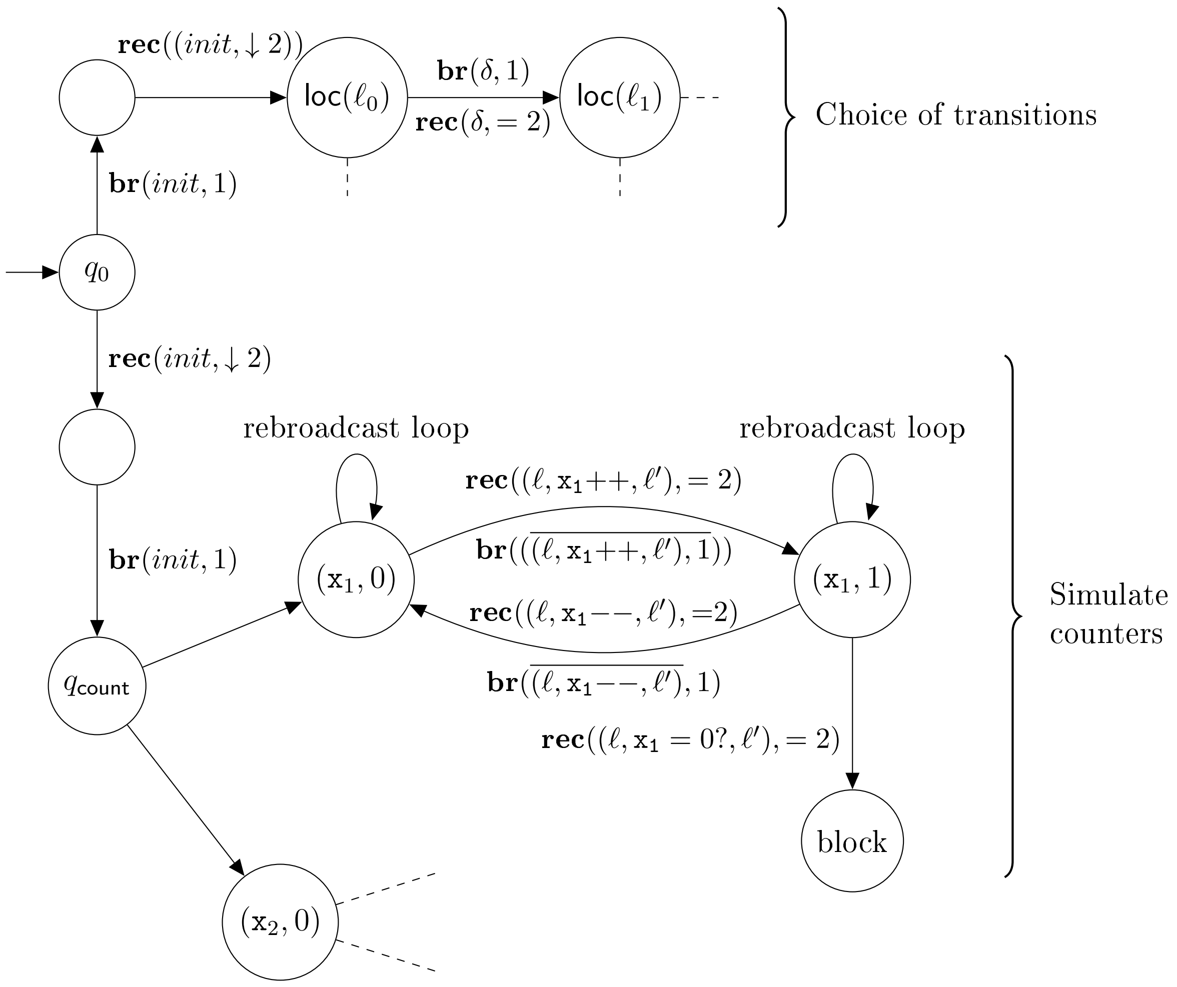

$\mathbf{rec}((\mathit{init}, \downarrow 2))$
$\mathsf{loc}(\ell_0)$
$\mathbf{br}(\delta, 1)$
$\mathbf{rec}(\delta, = 2)$
$\mathsf{loc}(\ell_1)$
Choice of transitions
$\mathbf{br}(\mathit{init}, 1)$
q_0
$\mathbf{rec}(\mathit{init}, \downarrow 2)$
rebroadcast loop
rebroadcast loop
$\mathbf{rec}((\ell, \mathtt{x}_1{+}{+}, \ell'), = 2)$
$\mathbf{br}((\overline{(\ell, \mathtt{x}_1{+}{+}, \ell'), 1}))$
$(\mathtt{x}_1, 0)$
$(\mathtt{x}_1, 1)$
$\mathbf{br}(\mathit{init}, 1)$
$\mathbf{rec}((\ell, \mathtt{x}_1{-}{-}, \ell'), =2)$
$\mathbf{br}(\overline{(\ell, \mathtt{x}_1{-}{-}, \ell')}, 1)$
q_{count}
$\mathbf{rec}((\ell, \mathtt{x}_1 = 0?, \ell'), = 2)$
block
Simulate counters
$(\mathtt{x}_2, 0)$

Figure 3.12: Illustration of the protocol constructed in the proof of Theorem 3.45. The "rebroadcast" loops mean that any letter received with the datum of register 2 (and that does not match one of the indicated transitions) is broadcast immediately after with the datum of register 1. '

will necessarily be formed in any run satisfying the SYNCHRO requirement. Such a cycle is drawn in Figure 3.11.

The rest of the construction aims at faithfully simulating the machine in a cycle. The protocol is drawn in Figure 3.12.

At the start, each agent can either enter the $\mathcal{R}_{\mathsf{loc}}$ or $\mathcal{R}_{\mathsf{count}}$ part of the protocol. In the first case, the agent starts by broadcasting its initial datum and then receive a message and store the associated datum. In the second case, it does those two things in the reverse order. Agents in $\mathcal{R}_{\mathsf{loc}}$ send a sequence of instructions and wait after each one for a confirmation that it was executed. Agents in $\mathcal{R}_{\mathsf{count}}$ simulate counter values. The messages circulating in a cycle contain either a transition $\delta \in \Delta$ or an acknowledgment $\overline{\delta}$ with $\delta \in \Delta$. An agent a in $\mathcal{R}_{\mathsf{count}}$ first picks a counter $\mathtt{x}_i$ it simulates, and goes to state $(\mathtt{x}_i, 0)$. If a is in $(\mathtt{x}_i, 0)$ and receives δ corresponding to an increment of $\mathtt{x}_i$, it goes to $(\mathtt{x}_i, 1)$ and broadcast an acknowledgment $\overline{\delta}$, and conversely for decrements. If δ is a zero-test $\mathtt{x_i} = 0?$ and a is on state $(\mathtt{x}_i, 1)$ then it stops, making the whole cycle fail. Otherwise it propagates the message by broadcasting δ. Other messages are rebroadcast as such.

An agent a in $\mathcal{R}_{\mathsf{loc}}$ starts in state $\mathsf{loc}(\ell_0)$. When in state $\mathsf{loc}(\ell)$, it picks and broadcasts a transition $\delta = (\ell, \alpha, \ell') \in \Delta$, waits for the acknowledgment $\overline{\delta}$ and goes to $\mathsf{loc}(\ell')$. In the case where δ is a zero-test, we have $\overline{\delta} = \delta$: there is no need for a distinct acknowledgment because there is no action to perform (if the test fails then no message is transmitted). When in $\mathsf{loc}(\ell_f)$, a broadcasts a special message end that propagates in the cycle and makes everyone go to q_{err}. When it receives itself the message *end*, it goes to q_{err}.

It is quite easy to see that, if ℓ_f can be covered in M, one can build a run of $\mathcal{R}$ where all agents end in q_{err}. Let N the highest counter value in the execution of M covering q_{err}. The run of $\mathcal{R}$ first puts all its agents in the same cycle; exactly one agent a_{lead} goes in $\mathcal{R}_{\mathsf{loc}}$ and $2N$ agents go in $\mathcal{R}_{\mathsf{count}}$; half of these simulate $\mathtt{x}_1$ and half $\mathtt{x}_2$, so that the largest counter value is never exceeded. It then suffices to faithfully simulate the execution of M: a_{lead} selects the corresponding sequence of transitions, their effect is always applied as we have enough agents simulating each counter. After each round the number of agents in state $(\mathtt{x}_i, 1)$ is exactly the value of $\mathtt{x}_i$ at this point in the run of the machine, hence zero-tests never cause failure. In the end a_{lead} reaches $\mathsf{loc}(\ell_f)$ and broadcasts end, allowing every agent in $\mathcal{R}_{\mathsf{count}}$ to get to q_{err}.

For the converse implication, suppose that we have a run ϱ of $\mathcal{R}$ where all agents end in q_{err}. As mentioned before, there must be a cycle of agents $a_1, \ldots, a_n$ in this run. Observe that all agents alternate between broadcasts and receptions, so that to reach q_{err} they must all have made the same number of broadcasts and receptions. This implies that no message was lost along the cycle.

Note that there may be several agents in $\mathcal{R}_{\mathsf{loc}}$ along the cycle; however, they must all broadcast exactly the same sequence of transitions, otherwise one of them would lack an acknowledgment and would not get to q_{err}. Let a be the agent that first reaches $\mathsf{loc}(\ell_f)$ and a' the first agent in $\mathcal{R}_{\mathsf{loc}}$ after a in the cycle; there are only agents in $\mathcal{R}_{\mathsf{count}}$ between a and a' in the cycle, we call these agents *intermediate agents*. The intermediate agents faithfully encode the two counters and all decrements and zero-tests pass, otherwise a' would lack an acknowledgment. Therefore, the sequence of transitions of a defines an execution of M that covers ℓ_f, which concludes the proof. □

3.7 Extensions

In this section we discuss several possible extensions of the previous results to other models of computation.

3.7.1 Allowing agents to see data

In this chapter we only considered control strategies that chose transitions based on the previous sequence of transitions, and not the sequence of data received. It is natural to wonder what happens if we use strategies of the form $\sigma : (\Delta\mathbb{D}^r)^* \to \Delta$. For this section we will only consider the signature case to make things easier. We conjecture that the following proof can be adapted to the general case.

A central ingredient in this proof is Ramsey's theorem on infinite hypergraphs, which extends naturally Ramsey's theorem on graphs.

Given a set S and $k \in \mathbb{N}$, we use the notation $\binom{S}{k}$ for the set of subsets of S of size k.

Theorem 3.46 ▶ Ramsey's theorem on infinite hypergraphs

Let V be an infinite set of vertices and $k \in \mathbb{N}$. Let $\mathbf{col} : \binom{V}{k} \to C$ with C a finite set of colours. Then there exists an infinite subset $V' \subseteq V$ and $c \in C$ such that $\mathbf{col}(\binom{V'}{k}) = \{c\}$.

Let us fix $\mathcal{G} = (\mathcal{R}, Q_{ctrl}, Q_{env}, m_{err})$ a signature BGR. We say that a local run has *organised data* if

- if whenever a datum is received for the first time, it is greater than the initial datum and all data received previously.
- Each datum is recorded at most once in the registers.

Proposition 3.47

There is a function $h : \mathbb{N} \to \mathbb{N}$ (not depending on $\mathcal{R}$) such that:
If a control strategy is losing then there exists a σ-run ϱ in which every local run has length at most $h(|\mathcal{R}|)$ and has organised data in which m_{err} is broadcast.

Proof. Let us start by defining an *execution tree* as a tree of the following form:

- There are two types of nodes, *word nodes* and *run nodes*
- The children of a word node are run nodes, and the children of a run node are word nodes.
- For all run node ν with a label u and all $d \in \mathbb{D}$ such that $\mathbf{In}_d(u) \neq \varepsilon$, ν has a child with a label w such that $\mathbf{In}_d(u) \sqsubseteq w$.
- For all word node ν labelled w, for all child ν' of ν labelled u, $w \sqsubseteq \mathbf{Out}_{sign}(u)$

Consider the following algorithm: We start with an execution tree made only of a root labelled m_{err}. We maintain a set of word nodes O, initially containing only the root. The word nodes in O are called *open*, others are called *closed*

While O is not empty, we apply the following steps:

- If there is a run node ν whose children are all closed, let ν' be its parent, labelled w. We remove every node that was added to the tree after ν' (in particular, we remove all of its descendants). Then, we remove ν from O.
- Otherwise, let B be the set of labels of open nodes, we define $I = B\!\uparrow^{\mathsf{c}}$. By Lemma 3.24, there exists a σ-local run u of length at most $\varphi(\mathcal{R}, B)$ such that $\mathbf{Out}_{sign}(u) \notin I$, $\mathbf{In}_d(u) \in I$ for all d and no datum is recorded twice. Let ν be an open node with a label w such that $w \sqsubseteq \mathbf{Out}_{sign}(u)$. It exists by definition of B and I. We add a child ν' to ν, labelled by u. Then, consider the set $\{\mathbf{In}_d(u) \mid d \in \mathbb{D}\} \setminus \{\varepsilon\}$, let B_u be its set of minimal elements for $\sqsubseteq$. For each $v \in B_u$ we add a child labelled v to ν', and we add all those children to O.

Note that when we remove a node we remove all nodes added after that one. As a consequence, at all times we can enumerate open nodes $O = \{\nu_0, \nu_1, \dots, \nu_k\}$ in their order of appearance, and we obtain $|w_i| \leq |u_i| \leq \varphi(\mathcal{R}, \{w_1, \dots, w_{i-1}\})$ for all i, with $w_1, \dots, w_k$ the labels of $\nu_1, \dots, \nu_k$ and $u_1, \dots, u_k$ the labels of their respective parents. Additionally, we maintain the fact that the sequence $w_1, \dots, w_k$ is a bad sequence. We can then apply the Length Function Theorem to bound k by $f(|\mathcal{R}|)$ with f a function of $\mathbf{F}_{\omega^\omega}$.

We also obtain a bound $h(|\mathcal{R}|)$ on the length of run node labels. As a consequence, the number of data appearing in each local run is bounded by that same bound, and thus the degree of the tree is at most $h(|\mathcal{R}|)$.

As every node is a leaf or an open node or the child of an open node, we get a bound $b(|\mathcal{R}|)$ on the size of the tree. As a consequence, the set of trees we see is finite. In order to show that the algorithm terminates, we simply have to show that we cannot loop.

Given the tree at some point of the algorithm, let $\nu_0, \dots, \nu_k$ be the set of word nodes in their order of creation. For each i, let x_i be 0 if ν_i is open and 1 if it is closed. It is easy to check that the sequence $x_0 \cdots x_k$ increases at each step for the lexicographic ordering. As a result, we never see the same tree twice. The algorithm therefore terminates in at most $c(|\mathcal{R}|)$ steps.

Claim 3.47.1. *After each iteration, for all closed node ν labelled w, ν is a leaf and there is a σ-run in which every local run has organised data and has length at most $h(|\mathcal{R}|)$ and in which the sequence w is broadcast by some agent.*

> *Proof of the claim.* We proceed by induction on the number of iterations. This property is clearly true at the beginning as there are no closed nodes.
> For the induction step, note that we never add children to closed nodes and only turn leaves into closed nodes. Hence we maintain the fact that every closed node is a leaf. Furthermore, say we turn an open node labelled w into a closed one. We do so when it has a child ν' whose children are all closed. Let u be the label of ν' and $w_1, \dots, w_n$ the labels of its children. By induction hypothesis, for each i we have a σ-run ϱ_i in which each local run has organised data and length at most $h(|\mathcal{R}|)$ and in which an agent broadcasts w_i.
> For each d received in u, we know that there is an i such that $\mathbf{In}_d(u) \sqsubseteq w_i$. We define ϱ_d as ϱ_i where data have been renamed so that w_i is broadcast with datum d and all other data are fresh and do not appear in u.
> Let $d_1, \dots, d_m$ be the data received in u, in order of appearance. Let d_0 be the initial datum of u. For all $j \in [2, m]$, in increasing order, let A_{j-1} be such that all data appearing in ϱ_{j-1} are below A_{j-1}. Let $A_0 = d_0 + 1$. Define ϱ'_j as ϱ_{d_j} where each datum d' has been renamed into $d' + A_{j-1}$. The runs ϱ'_j use disjoint sets of data and in each

ϱ'_j an agent broadcasts $\mathbf{In}_{d_j}(u)$ with datum $d'_j = d_j + A_{j-1}$. In particular we have $d_0 < d'_1 < \cdots < d'_m$. We have also maintained the fact that all local runs in those runs have organised data.
We rename each datum d_j with in u to d'_j and obtain a local run u' with organised data and $\mathbf{In}_{d_j}(u) = \mathbf{In}_{d'_j}(u')$. We can execute all runs ϱ'_j and u over disjoint sets of agents, and use the broadcasts in each ϱ'_j to match the receptions in u. This gives us a σ-run ϱ_i in which each local run has increasing data and length at most $h(|\mathcal{R}|)$ and in which an agent broadcasts w (by executing u). ■

As the algorithm terminates, eventually the root is closed. By the claim above, we have a σ-run ϱ in which each local run has length at most $h(|\mathcal{R}|)$ and in which an agent broadcasts m_{err}. □

We have shown that if a control strategy is not winning then there exists a σ-run in which all local runs have bounded size B in which m_{err} is broadcast, with B depending only on the size of the system. Furthermore when constructing this run, we have ensured that in every local run data always appear in increasing order with respect to their order in $\mathbb{N}$, and each datum is recorded at most once.

We now define *data-aware control strategies*. They are functions $\sigma : \mathbb{D}(\Delta \times \mathbb{D})^* \to \Delta$. The choice of the next transition is based on the sequence of transitions seen so far, along with the initial datum and the data received.

A σ-local run is an initial local run $u = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$ such that for all $i \in [1, n]$, if $q_{i-1} \in Q_{\mathsf{ctrl}}$ then $\sigma(c_0(1)\delta_1 d_1 \cdots \delta_{i-1} d_{i-1}) = \delta_i$. A σ-run is an initial run whose projection on every agent a is a σ-local run. We say that σ is winning if no σ-run contains a broadcast of m_{err}.

Theorem 3.48

There is a winning data-aware control strategy for $\mathcal{G}$ if and only if there is a winning control strategy for $\mathcal{G}$.

Proof. As a control strategy is a particular case of data-aware control strategy, the right-to-left direction is clear.

For the left-to-right direction, suppose there is a winning data-aware control strategy σ for $\mathcal{G}$. Let $K = h(|\mathcal{R}|)$ with h as defined in Proposition 3.47. Let R_K be the set of σ-local runs with organised data of length at most K.

We define a function $\mathbf{col} : \binom{\mathbb{D}}{K+1} \to 2^{\mathsf{R}_K}$ as follows. Let D be a set of $R+1$ data. Let $d_0, \cdots, d_R$ be the elements of $\mathbb{D}$ in increasing order. Then $\mathbf{col}(D)$ is the set of σ-local runs with organised data of length at most R such that the initial datum is d_0 the other data appearing in the run are $d_1, \ldots, d_k$ for some k. With the organised data property and those conditions, the local run is entirely determined by its sequence of transitions. As a result, $|\mathbf{col}(D)| \leq |\Delta|^K$.

In a local run with at most B steps, at most $B + 1$ data appear. As a result, every element of R_K has an antecedent by $\mathbf{col}$. We can now apply Theorem 3.46. We obtain an infinite set $\mathbb{D}' \subseteq \mathbb{D}$ of data and a set of local runs R such that $\mathbf{col}(\binom{\mathbb{D}'}{K+1}) = \{R\}$.

Let $d_0, \ldots, d_K \in \mathbb{D}'$ with $d_0 < \cdots < d_K$. Define the strategy $\sigma' : \Delta^* \to \Delta$ which, given a sequence of transitions, takes the same decision as σ over the unique local run with that sequence of transitions, organised data, and using data $\{d_0, \ldots, d_k\}$ for some k, with d_0 the initial datum.

If σ' was losing, we would have a run in which every local run has length at most K and organised data in which m_{err} is broadcast. This run is, however, also a σ-run, which is a contradiction.

As a result, σ' is winning. $\square$

As a consequence, by Theorem 3.43, we obtain the following result.

Corollary 3.49

The existence of a winning data-aware control strategy for a BGR is decidable and $\mathbf{F}_{\omega^\omega}$-complete.

3.7.2 Pushdown protocols

In this section we show that decidability results obtained so far can be generalised to pushdown BGR, where processes have local stacks. This may at first be surprising, as we saw that we can encode lossy channel systems in BGR, and reachability in LCS with a stack is a longstanding open problem. In fact, the decidability of reachability in pushdown VASS (which are much weaker than pushdown LCS) is also wide open.

The subtlety lies in the fact that we cannot encode pushdown LCS into pushdown BGR: this would require to allow processes to broadcast the full content of their stack, which is not possible in the model.

The decidability proofs are not hard to extend to the pushdown case: it suffices to use Lemma 2.10, which allows us to decide the winner of a pushdown game and bound the time it takes the first player to win if he does.

Define a *pushdown BGR* with r registers over $\mathbb{D}$ as a BGR whose protocol has infinitely many states, encoded as follows.

Let $Op_{\mathcal{M},r} = \{\mathbf{br}(m,i), \mathbf{rec}(m,=i), \mathbf{rec}(m,\downarrow i), \mathbf{rec}(m,\neq) \mid m \in \mathcal{M}, i \in [1,r]\}$ be the set of operations of register transducers over letters of $\mathcal{M}$ and r registers.

We have a pushdown automaton $\mathcal{A} = (Q, \Sigma, \Gamma, \Delta, q_{init})$ (we do not include final states as they are irrelevant here) and a function $\mathsf{op} : \Sigma \to Op_{\mathcal{M},r}$, as well as a partition of Q into Q_{ctrl} and Q_{env}. We can assume without loss of generality that $\mathcal{A}$ is deterministic, by extending the alphabet.

The register transducer encoded this way has set of states $Q\Gamma^*$, with q_{init} the initial state. It has a transition $qy \xrightarrow{\alpha} q'y'$ with $\alpha \in Op_{\mathcal{M},r}$ when $qy \xrightarrow{\alpha} q'y'$ is a valid step of $\mathcal{A}$. The set of states of Controller is $Q_{\mathsf{ctrl}}\Gamma^*$, the set of states of Environment is $Q_{\mathsf{env}}\Gamma^*$.

Theorem 3.50

SafeStrat is $\mathbf{F}_{\omega^\omega}$-complete for pushdown BGR.

Proof. The lower bound is immediate from Proposition 3.44.

For the upper bound, the proof is essentially the same as the one from Section 3.5. We simply point out what should be adapted. Lemma 3.31 does not make any finiteness assumption on the system, thus the same characterisation of winning strategies with invariants can be applied to pushdown BGR. Lemma 3.38 needs a slight adaptation: it suffices to notice that the winning condition of the invariant game solely depends on the sequence of operations taken. The construction in the proof of Lemma 3.38 can thus be applied to sequences of operations instead of plays. We obtain an NFA of exponential size recognising sequences of operations making Environment win. This immediately yields an NFA of the same size recognising sequences $a_1 \cdots a_n \in \Sigma^*$ such that $\mathsf{op}(a_1) \cdots \mathsf{op}(a_n)$ makes Environment win. We can determinise this automaton to obtain a double-exponential automaton $\mathcal{B}$ recognising those sequences.

The invariant game is thus a pushdown game played on $\mathcal{A}$ with the objective described by $\mathcal{B}$. By Lemma 2.10, we can solve this game in exponential time in the sizes of $\mathcal{A}$ and $\mathcal{B}$, thus in triple-exponential time in the sizes of $\mathcal{A}$, B and $(B_m)_{m \in \mathcal{M}}$.

Furthermore, if Environment wins, he can do so in triple-exponential time. This gives the existence of the elementary function φ in the statement of Lemma 3.38.

The rest of the proof can be conducted exactly as for the finite-state case, yielding the result. □

The entire reduction from the SafeStrat problem to invariant games stays the same. What remains is to be able to decide the winner of an invariant game, and bound the length of a winning play for Environment if there is one with an elementary bound. Both those conditions are provided by the lemma above.

3.7.3 Leader

Another possible extension of the model would be to allow the presence of a leader. To formalise this, we call a *protocol with leader* a pair $(\mathcal{R}, q_{lead})$ with $\mathcal{R} = (Q, \mathcal{M}, q_{init}, \Delta)$ a protocol with r registers and q_{lead} a state of $\mathcal{R}$. The only difference in the semantics is that runs start with one agent in q_{lead} and all others in the initial state of $\mathcal{R}$.

An *initial configuration with leader* of $(\mathcal{R}, q_{lead})$ is a configuration $\gamma : \mathbb{A} \to Q \times \mathbb{D}^r$ of $\mathcal{R}$ such that:

- There exists $a_\ell \in \mathbb{A}$ such that $\mathsf{st}(\gamma)(a_\ell) = q_{lead}$ and $\mathsf{st}(\gamma)(a) = q_{init}$ for all $a \in \mathbb{A} \setminus \{a_\ell\}$
- for all $a \in \mathbb{A}$ and $i, i' \in [1, r]$, $\mathsf{data}(\gamma)(a, i) = \mathsf{data}(\gamma)(a, i')$
- for all $a \neq a' \in \mathbb{A}$ and $i, i' \in [1, r]$, $\mathsf{data}(\gamma)(a, i) \neq \mathsf{data}(\gamma)(a', i')$

We can then formulate the following problem:

Definition 3.51 ▶ Cover with leader

The *coverability problem with leader* asks, given a protocol with leader $\mathcal{R}$ and a state q_{err}, whether there is a run of $\mathcal{R}$ starting in an initial configuration with leader in which at least one agent reaches q_{err}.

This problem is undecidable, as soon as $r \geq 2$:

Theorem 3.52

The coverability problem with leader is undecidable.

Proof. We follow the same idea as for the proof of Theorem 3.45. We form a cycle as in Figure 3.11 and use it to faithfully simulate a Minsky machine.

We use an alphabet $\mathcal{M}$ containing a letter *start*, a letter δ for each transition $\delta \in \Delta$ in M, and a letter $\overline{\delta}$ if δ is an increment or decrement.

Let $M = (\mathrm{Loc}, \Delta, \mathtt{X}, \ell_0, \ell_f)$ be a Minsky machine. The difference is in the way we form that cycle. The leader starts by broadcasting a message *start* with its identifier and then receiving *start* and storing the received datum in register 2. It then simulates the Minsky machine. It has a state $\mathsf{loc}(\ell)$ for each state $\ell \in \mathrm{Loc}$. It selects a sequence of transitions of M and, for each transition δ, it broadcasts δ with its identifier and waits for an acknowledgement: $\bar{\delta}$ if δ is an increment or decrement, and just δ if δ is a zero test.

Other agents start by receiving a message *start*, storing the received datum in register 2, and then broadcasting *start* with their own identifier. Then they choose a counter $\mathtt{x}_i$, with $i \in \{1, 2\}$, and go to state 0_i. During the rest of the run the agent only receives messages with the datum in its second register. It also only broadcasts messages with its own identifier, in register 1. When it receives a message with a letter δ it proceeds as follows:

- if δ is an increment of $\mathtt{x}_i$ and the agent is in state 0_i then it goes to state 1_i and broadcasts $\bar{\delta}$
- if δ is a decrement of $\mathtt{x}_i$ and the agent is in state 1_i then it goes to state 0_i and broadcasts $\bar{\delta}$
- if δ is a zero-test of $\mathtt{x}_i$ and the agent is in state 0_i then it stays in state 0_i and broadcasts δ
- if δ is a zero-test of $\mathtt{x}_i$ and the agent is in state 1_i then it does not broadcast anything

In all other cases the agent stays in the same state and broadcasts the same letter that it received.

We now argue that ℓ_f is reachable in M if and only if $\mathsf{loc}(\ell_f)$ in coverable in $(\mathcal{R}, q_{lead})$
$\Longrightarrow$ If ℓ_f can be covered in M, one can build a run of $\mathcal{R}$ where th leader ends in $\mathsf{loc}(\ell_f)$. Consider an execution of M covering ℓ_f. Let N the highest counter value in that execution. The run of $\mathcal{R}$ first puts all its agents in the same cycle; exactly one agent a_{lead} goes in $\mathcal{R}_{\mathsf{loc}}$ and $2N$ agents go in $\mathcal{R}_{\mathsf{count}}$; half of these simulate $\mathtt{x}_1$ and half $\mathtt{x}_2$, so that the largest counter value is never exceeded. It then suffices to faithfully simulate the execution of M: a_{lead} selects the corresponding sequence of transitions, their effect is always applied as we have enough agents simulating each counter. After each round the number of agents in state $(\mathtt{x}_i, 1)$ is exactly the value of $\mathtt{x}_i$ at this point in the run of the machine, hence zero-tests never cause failure. In the end a_{lead} reaches $\mathsf{loc}(\ell_f)$ and broadcasts end, allowing every agent in $\mathcal{R}_{\mathsf{count}}$ to get to q_{err}.
$\Longleftarrow$ For the converse implication, suppose that we have a run ϱ of $\mathcal{R}$ where all agents end in q_{err}. As mentioned before, there must be a cycle of agents $a_1, \ldots, a_n$ in this run. Observe that all agents alternate between broadcasts and receptions, so that to reach $\mathsf{loc}(\ell_f)$ they must all have made the same number of broadcasts and receptions. This implies that no message was lost along the cycle. □

The key difference with the coverability problem without leader model is that if we can have several agents behaving like the leader, when an agent receives $\bar{m}$ with an identifier, it has no way to know if this comes from a chain of messages that was started by him or by another leader. There is no way to make sure that we obtain a cycle.

3.8 The case of one register

In the previous sections we established the theoretical complexity of Cover, Synchro and SafeStrat for BNRA and BGR. Our constructions for lower bounds in Proposition 3.44 and 3.45 only require two registers: one to remember the agent's identifier and sign messages, and one to ensure that we receive messages from a single other process.

We studied in Section 3.3 the complexity of those problems when protocols do not have registers. The remaining gap is for protocols with one register. We call BNRA (resp. BGR) with a single register *1BNRA* (resp. *1BGR*). They offer an intermediate step in terms of tractability and expressivity between the protocols without registers and the general case. We will also see in Section 3.8.3 that their study connects to a new line of research on population protocols with data.

Essentially, those protocols can sign messages with their initial identifier, and check that several messages have the same datum, but not simultaneously. In order to record a received datum and be able to compare it with the ones of following messages, an agent must forget its own identity.

We start by studying the Cover and Synchro problems, and show that they are both NP-complete. As Cover reduces to Synchro, we simply prove the upper bound for Synchro and the lower bound for Cover.

We treat the SafeStrat problem separately, as it requires a different approach in this case.

3.8.1 Coverability

Our aim in this section is to present the following theorem.

Theorem 3.53

Cover is NP-complete on 1BNRA.

The proof is quite tedious, and was already published in [**GuillouMW24**], therefore we only include a proof sketch in this document.

The NP lower bound follows from a reduction from 3SAT (an agent a sends a sequence of messages representing a valuation, with its identifier, to other agents which broadcast it back, playing the role of external memory, allowing a to check the satisfaction of a 3SAT formula).

To prove the NP upper bound, we present an abstraction on configurations and runs. The main ingredient of our abstraction is twofold:

- First, if there is a run ϱ sending an agent to a state q, then we can construct a run ϱ' executing that same run over disjoint sets of agents as many times as we want to obtain as many agents as we need in q (with different values).

- Furthermore, if in a run ϱ an agent a gets in a state q with a datum d that is not its initial one, it means that at some point in the run, it was in a state q' and executed a transition in which it received a message with datum d, stored it in its register and went to a state q''. As mentioned before, we can copy the run up to that point to have an unlimited supply of agents in q', and thus an unlimited supply of agents in q'' with datum d. We can then make all those agents copy the broadcasts of a

and receive the same messages so that they all reach q with datum d. Hence if we have an agent in a state q with a datum d that is not its initial one, we can assume that we have as many agents in q with value d as we need.

This leads us to define an abstraction of runs. We start by defining *gangs*, which are of the form (s, T) with s a state and T a set of states. The set of states T represents the set of agents carrying a datum d, at some point in the run. The first component s is the current state of the agent who had datum d initially, while T is the set of states in which some agents with datum d are currently.

If the original owner of this value no longer has it, then $s = \perp$. Note that we define the clique as the set of states q such that *at some point in the run* some agent was in state q with value v. This is because we can use the copycat principle to add a large amount of agents that are in state q with value v, and thus we can assume that there is always one.

In a concrete run of our system, gangs of distinct values may only interact with one another by covering states q which are needed by the other gang (in the form of a broadcast or of a '$\downarrow$' reception from q); therefore our abstraction also keeps track of the set of coverable states, which may only grow. However, it only needs to keep track of one gang at a time.

This leads us to a natural abstract semantics based on gangs. An abstract configuration consists of a set of states S (states covered so far by some agents) and a gang (b, K) (the agent who carried d initially and the states reached by other agents with that datum). If the original owner of d stores a new datum we set $b = \perp$. Abstract transitions are defined by applying transitions of the protocol while assuming that we have unlimited supplies of agents in every state of S and of agents with datum d in all states of K. At any time, we can apply a *gang reset*, which maintains S but reinitializes (b, K) to $(q_0, \emptyset)$ (we track a new value). To bound the length of relevant abstract runs, we impose that S should grow between two gang resets (otherwise they reset to the same abstract configuration) and that there may be at most $O(|Q|^2)$ abstract steps between two resets (as K can only increase and there are only $|Q| + 1$ possibilities for b). This means that if there is an abstract run covering a state, there is one of size $O(|Q|^3)$, proving the NP upper bound.

We leave the decidability and complexity of the SYNCHRO problem in the 1BGR case as an open problem.

Open problem 3.54

Is SYNCHRO decidable for 1BNRA? If so, what is its complexity?

We conjecture that we can use an abstraction similar to the one used for COVER to show that the problem is NP-complete.

3.8.2 Strategy synthesis

We investigate the complexity of safe strategy synthesis in the case of 1BGR. In this case, a record transition essentially resets the memory of the process. This allows to split the invariant game used in the general case into simpler games: the output game and the echo games. The output game concerns processes that still have their original datum in their register. In that game, we show that Controller always has an optimal positional strategy. The echo game starts when a process records a new datum in their register. There, we show that Environment always has an optimal positional strategy.

Those facts allow us to reduce the class of invariants we have to consider: We show that we can restrict ourselves to invariants whose basis only contain words of length polynomial in the size of the system. This means that we can guess in NExpTime an invariant and check that Controller wins all associated games. The last part of the upper bound proof justifies that we can verify that Controller wins all those games in NExpTime. Finally, we show the lower bound by a reduction from the exponential grid tiling problem.

Theorem 3.55

SafeStrat is NExpTime-complete on 1BGR.

We start with the upper bound. In what follows we will rely on the existence of memoryless, or positional, strategies in some games. In particular, we will rely on the following criterion, which can be used to show that some player can win with a positional strategy.

We say that an objective $\mathcal{L}$ is *submixing* (sometimes called *concave*) if whenever we have words $u_0, u_1, \dots$ and $v_0, v_1, \cdots$ such that

$$u_0 u_1 \cdots \notin \mathcal{L}$$
$$v_0 v_1 \cdots \notin \mathcal{L}$$

then $u_0 v_0 u_1 v_1 \cdots \notin \mathcal{L}$.

Proposition 3.56 ▶ [Kopczynski06]

If an objective is submixing then player P_0 has a positional optimal strategy in all games with this objective.

Proposition 3.57

SafeStrat is in NExpTime on 1BGR.

To prove this statement, we rely on the characterisation of winning control strategies by the invariant game, as stated in Lemma 3.39 and 3.40. It turns out that for 1BGR, the invariant game can be split into several simpler games. Essentially, we consider the recording of a new value in the register as a reset of the game.

We define two different games: in the output game the players build the part of the local run before the first record transition. In the echo game the players build an interval of the local run between two record transitions.

We show that in the first game Controller can always use a positional strategy while in the second one it is Environment who can stick to positional strategies. In both cases we use the submixing property of their objectives to prove it.

We show that the winners of those games determine the winner of the 1BGR. The positionality of Environment's strategy in the echo game then lets us bound the size of the invariants necessary to witness the existence of a winning control strategy for Controller.

Finally, we exhibit an NExpTime algorithm, in which the non-deterministic guess is the invariant and a positional strategy for Controller in the output game.

For the rest of this section we fix a 1BGR $\mathcal{G} = (\mathcal{R}, Q_{\mathsf{ctrl}}, Q_{\mathsf{env}}, q_{err})$.

The *output game* is played on $\mathcal{R}$, with players picking transitions from their respective states. It has two parameters: an invariant $(I, (J_m)_{m\in\mathcal{M}})$, and a set of record transitions $T \subseteq \Delta$. We will use the term $(I, (J_m)_{m\in\mathcal{M}}, T)$*-output game* for the output game with those parameters.

The winning condition is defined as follows:

(O1) If at some point the play is not compatible with any decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$, then Controller wins.

(O2) If at some point we see a disequality transition receiving a letter $m \notin I$ then Controller wins.

(O3) If the output of the play is not in I then Environment wins.

(O4) If we reach a record transition then Controller wins if it is in T and Environment wins otherwise.

(O5) If the play goes on forever without any of the previous things happening then Controller wins.

Lemma 3.58

If Controller wins an output game then she has a positional winning strategy.

Proof. We show that Controller's objective in an output game is submixing. This proves the lemma by applying [**Kopczynski06**].

Consider two losing plays for Controller π and π', and a third play $\bar{\pi} = \pi_0\pi'_0\pi_1\cdots$ obtained by shuffling the two. We show that $\bar{\pi}$ is also losing for Controller. As π and π' are losing for Controller, we can consider them as finite: the victory of Environment is witnessed by a finite prefix. We can cut π and π' into $\pi = \pi_0\pi_1\cdots\pi_m$ and $\pi' = \pi'_0\pi'_1\cdots\pi'_m$ so that $\bar{\pi} = \pi_0\pi'_0\pi_1\cdots\pi_m\pi'_m$ (note that π'_m can be empty).

Clearly no transition of T is seen in π or π', thus not in $\bar{\pi}$ as well. Let $\tilde{\pi}$ a prefix of $\bar{\pi}$, we show that it is compatible with some decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$. Let $\tilde{m}_1, \dots, \tilde{m}_k$ be the set of letters received along $\tilde{\pi}$, in that order. Let $\tilde{\pi} = \tilde{\pi}_0\cdots\tilde{\pi}_k$ so that for each i the first step of $\tilde{\pi}_i$ is the first reception of m_i. Let $\tilde{v}_i$ be the sequence of letters broadcast in $\tilde{\pi}_i$, for all i. Let $\tilde{\mathsf{dec}} = (\tilde{v}_0, \tilde{m}_1, \dots, \tilde{v}_k)$. Clearly $\tilde{\pi}$ is compatible with dec.

It remains to show that $\tilde{\mathsf{dec}} \in \mathcal{D}((J_m)_{m\in\mathcal{M}})$. Let $i \in [1, k]$, we need to find a word in $\mathcal{L}_{(\tilde{v}_0,\tilde{m}_1,\dots,\tilde{v}_{i-1})} \cap J_{m_i}$. For that, we observe that the reception of $\tilde{m}_i$ happens in either a segment from π or from π'. We assume that it is from π, the other case is symmetric. Since every prefix of π is compatible with some decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$, in particular the prefix of π up to that reception of $\tilde{m}_i$ is compatible with one. Thus there exists $\mathsf{dec} = (v_0, m_1, \dots, v_\ell)$ such that $\mathcal{L}_{\mathsf{dec}} \cap J_{\tilde{m}_i} \neq \emptyset$ and with which π is compatible. Let $w \in \mathcal{L}_{\mathsf{dec}} \cap J_{\tilde{m}_i}$, w can be obtained from $v_0\cdots v_\ell$ by adding letters from $\{m_1, \dots, m_j\}$ to each v_j.

As this prefix of π is fully contained in $\tilde{\pi}$, we can find the same sequence of broadcast $v_0\cdots v_\ell$ in $\tilde{\pi}$. Moreover, for each j, the first reception of m_j can only be earlier in $\tilde{\pi}$ than in π, hence $\tilde{\mathsf{dec}}$ allows us to find $v_0\cdots v_\ell$ and to add the same letters at the same places. As a consequence, $w \in \mathcal{L}_{(\tilde{v}_0,\tilde{m}_1,\dots,\tilde{v}_{i-1})}$.

It follows that every prefix of $\bar{\pi}$ is compatible with some decomposition of $\mathcal{D}((J_m)_{m\in\mathcal{M}})$. As a consequence, Controller does not win at any point in $\bar{\pi}$.

If some record transition outside of T is seen in π or π' then in $\bar{\pi}$ as well. Otherwise, it means the output of some prefix of π is not in I. As the output of that prefix must be a subword of the output of $\bar{\pi}$, and I is downward-closed, we obtain that the output of $\bar{\pi}$ is not in I.

In conclusion, Controller does not win at any point in $\bar{\pi}$ while Environment does. As a consequence, Controller's objective is submixing and thus if Controller wins she can win with a positional strategy. □

The *echo game* is also played on $\mathcal{R}$, with players picking transitions from their respective states. It has as parameters an invariant $(I, (J_m)_{m\in\mathcal{M}})$, a set of record transitions $T \subseteq \Delta$ and a record transition $t = q \xrightarrow{\mathbf{rec}(m,\downarrow 1)} q'$. The play starts by taking transition t, and continues from q'.

(E1) If at some point the recent input on 1 is not in I, Controller wins.

(E2) If at some point we see a disequality transition receiving a letter $m \notin I$ then Controller wins.

(E3) If at some point we make a broadcast with letter m while the recent input on 1 is not in J_m, then Environment wins.

(E4) If we reach a record transition the game stops: If that transition is in T then Controller wins, otherwise Environment does.

(E5) If the play goes on forever without any of those things happening then Controller wins.

Lemma 3.59

If Environment wins an echo game then he has a positional winning strategy.

Proof. We show that Environment's objective in an echo game has the submixing property, and again apply [**Kopczynski06**].

Consider two losing plays for Environment π and π', and $\bar{\pi}$ a submixing of the two. We can cut π and π' into $\pi = \pi_0\pi_1\cdots$ and $\pi' = \pi'_0\pi'_1\cdots$ so that $\bar{\pi} = \pi_0\pi'_0\pi_1\cdots$.

At all times in $\bar{\pi}$ if we make a broadcast with letter m while the recent input is w, then that broadcast was made in π or π' with a recent input that is a subword of w. As Environment loses in π and π', and as J_m is upward-closed, $w \in J_m$.

Every record transition seen in $\bar{\pi}$ must be seen in π or π', hence must be in T.

As a consequence, Environment cannot win $\bar{\pi}$, hence Controller wins. The objective of Environment is therefore submixing, and thus if Environment wins he can win with a positional strategy. □

Lemma 3.60

Controller wins the $(I, (J_m)_{m\in\mathcal{M}})$-invariant game if and only if there is a set of record transitions T such that she wins the $(I, (J_m), T)$-output game and the $(I, (J_m), T, t)$-echo game for all $t \in T$.

Proof. Suppose Controller wins the $(I, (J_m)_{m \in \mathcal{M}})$-invariant game with a strategy σ. Let T be the set of record transitions taken in a σ-play in which no player has won yet.

We start with the $(I, (J_m), T)$-output game : let Controller apply the same strategy σ in that game.

Claim 3.60.1. *Let π be a play such that no record transition has been seen yet.*

Then π is winning for a player in the invariant game if and only if it is winning for that player in the output game.

> *Proof of the claim.* Take a look at the winning conditions in the invariant game. Condition (A), (C) and (D) are the same as (O1), (O2) and (O3). Condition (B) and (E) cannot happen: as we have not seen any record transition, $\mathtt{reg}(\pi') = \{1\}$ for all prefixes π' of π. ■

As a consequence, a σ-play can only be winning for Environment if we reach a record transition $t \notin T$ while Controller has not won. However, this means that the play obtained before reaching t is not winning for Controller in the invariant game either, by the previous claim. This contradicts the definition of T. Hence σ is winning for Controller in the $(I, (J_m), T)$-output game.

Let $t \in T$. We now show that we have a winning strategy for Controller in the $(I, (J_m), T, t)$-echo game.

Claim 3.60.2. *Let $t\pi_+$ be a play in the $(I, (J_m), T, t)$-echo game such that no record transition has been seen yet (apart from the first step). Let $\pi_- t$ be a play ending with t in the $(I, (J_m))$-invariant game such that no player wins in it.*

Then π_+ is winning for a player in the echo game if and only if $\pi_- t\pi_+$ is winning for that player in the output game.

> *Proof of the claim.* First of all note that $\mathtt{reg}(\pi_- t) = \emptyset$ as t updates the only register. As $\pi_- t$ is not winning for either player, no prefix of it fulfils either (A) or (D). We can then conclude that there is no play starting with $\pi_- t$ that fulfils either of those conditions.
> Furthermore, since no player wins in π_- and t updates the only register, conditions (B), (C), (E) are satisfied by $\pi_- t\pi_+$ if and only if they are satisfied by $t\pi_+$ if and only if $t\pi_+$ satisfies (E1), (E2), (E3) respectively.
> This proves the claim. ■

By definition of t there exists a play reaching t in the invariant game in which no player has won yet. Let π_- be the prefix of that play before reaching t. We define the strategy σ_E as $\sigma_E(\pi) = \sigma(\pi_- \pi)$.

Let us consider a σ_E-play $t\pi_+$ in the echo game and show that it cannot be winning for Environment.

By the claim above, a σ_E-play can only be winning for Environment if we reach a record transition $t' \notin T$ while Controller has not won. However, this means that the play π obtained before reaching t' is such that $\pi_- \pi$ is not winning for Controller in the invariant game either, by the previous claim. This contradicts the definition of T.

We have established that a winning strategy in the $(I, (J_m)_{m \in \mathcal{M}})$-invariant game yields a set of record transitions T and winning strategies in the $(I, (J_m), T)$-output game and the $(I, (J_m), T, t)$-echo game for all $t \in T$.

For the reverse direction, let us consider a set of record transitions T, σ_O a winning strategy in the $(I, (J_m), T)$-output game and, for all $t \in T$, σ_t a winning strategy in the

$(I, (J_m), T, t)$-echo game.

We define a strategy σ in the invariant game as follows: If π does not contain any record transition then $\sigma(\pi) = \sigma_O(\pi)$. Otherwise, let π' be the largest suffix of π with no record transition and t the record transition just before π'. We set $\sigma(\pi) = \sigma_t(t\pi')$.

It remains to show that σ is a winning strategy in the invariant game. Suppose by contradiction that there exists a finite σ-play winning for Environment. Let π be such a σ-play of minimal size. If π contains no record transition then by the first claim it is also winning for Environment in the output game. As σ mimics σ_O while no record transition has been seen, this is a contradiction with the fact that σ_O is winning.

On the other hand, if π contains a record transition, then we can decompose it as $\pi = \pi_- t \pi_+$ with t a record transition and π_+ the maximal suffix of π with no record transition.

Then by minimality of π, no player wins in π_-. As a result, by the second claim, $t\pi_+$ is winning for Environment in the $(I, (J_m), T, t)$-echo game. This is a contradiction as by definition of π, $t\pi_+$ is a σ_t-play, and σ_t is winning for Controller.

This concludes our proof. □

Lemma 3.61

If there exists a winning control strategy for a BGR then there exist I such that every word in the basis of I^c is of length $\leq |\mathcal{R}|(|\mathcal{M}|+1)$ and $(J_m)_{m\in\mathcal{M}}$ in which every word in the basis has length $\leq |\mathcal{R}|$ for all m and T a set of record transitions such that Controller wins the $I, (J_m), T$-output game and the $I, (J_m), T, t$-echo game for all $t \in T$.

Proof. Suppose there exists a winning control strategy σ, then we have some I, (J_m) such that Controller wins $I, (J_m), T$-output game and the $I, (J_m), T, t$-echo game for all $t \in T$. We can assume that the sum of the lengths in the basis of the J_m is minimal.

We remove a word w from the basis of J_m. By minimality of J_m the resulting invariant is not sufficient. Hence Environment wins one of the games. Since I has not changed but (J_m) has decreased, Controller still wins the output game. As a consequence, Environment wins the $I, (J_m), T, t$-echo game for some $t \in T$. Let σ_{echo} be a positional winning strategy for Environment in the new instance of that game. There must be a σ_{echo}-play that is losing for him in the previous instance. As we have decreased $\mathcal{L}(I, (J_m))\downarrow$, the only possibility is that there is a play in which we broadcast m while the recent input is not in J_m. As J_m is upward-closed and σ_{echo} is positional, we can cut all cycles from this play: We obtain a σ_{echo}-play whose recent input is not in J_m, of length at most $|\mathcal{R}|$. As a consequence, $|w| \leq |\mathcal{R}|$.

We have shown that all words in the basis of all J_m have length at most $|\mathcal{R}|$. Let us now bound the words in the basis of I^c.

Consider I with a basis of minimal size such that $I, (J_m)$ is a sufficient invariant for σ. We remove a word w from the basis of I^c, thus increasing I. By minimality of I^c, Environment wins one of the games. It cannot be the output game as I has increased and the J_m are the same. If it is an echo game then let σ_{echo} be a positional winning strategy for Environment in the new instance of that game. There must be a play whose recent input was previously out of $\mathcal{L}(I, (J_m))\downarrow$ but is now in it. We can once again cut cycles on that play. Once we do so, we obtain a play of length $\leq |\mathcal{R}|$ whose recent input w_{in} is in $\mathcal{L}(I, (J_m))\downarrow$ but was not previously. As a consequence, there exists $\mathsf{dec} = (v_0, m_1, \ldots, v_k)$ such that $w_{in} \in \mathcal{L}_{\mathsf{dec}}\downarrow$ and for all i, $\mathcal{L}_{(v_0, m_1, \ldots, v_{i-1})} \cap J_{m_i} \neq \emptyset$. As we have bounded the lengths of words in the basis of each J_m by $|\mathcal{R}|$, there exist $u_1, \dots, u_k$, all of size at most $|\mathcal{R}|$, such that $u_i \in J_{m_i}$ and $u_i \in \mathcal{L}_{(v_0, m_1, \dots, v_{i-1})}\downarrow$.

For each u_i and for w_{in} at most $|\mathcal{R}|$ letters from $v_0, \cdots, v_k$ suffice to maintain these properties. We define $v'_0, \dots, v'_k$ as the words obtained by removing all other letters. Let $\mathsf{dec}' = (v'_0, m_1, \dots, v'_k)$. We therefore have $|v'_0 \cdots v'_k| \leq |\mathcal{R}|(|\mathcal{M}|+1)$, and $(v'_0, m_1, \dots, v'_k) \in \mathcal{D}(I, (J_m))$ and $w_{in} \in \mathcal{L}_{\mathsf{dec}'}\downarrow$.

As a consequence, we must have that $v'_0 \cdots v'_k$ is in I but was previously not. As a result, $w \sqsubseteq v'_0 \cdots v'_k$ and thus $|w| \leq |\mathcal{R}|(|\mathcal{M}|+1)$. □

Proposition 3.62

SafeStrat is in NExpTime for 1BGR.

Proof. We guess a positional strategy σ for Controller in the output game. We also guess a set of record transitions T, a set of words B of length $\leq |\mathcal{R}|(|\mathcal{M}|+1)$ and a family of sets of words $(B_m)_{m \in \mathcal{M}}$, where all words have length at most $|\mathcal{R}|$.

We then try to check if Environment has a winning strategy in one of the games. For the output game, we enumerate all positional strategies for Controller. As the size of words in the basis of I is bounded by $|\mathcal{R}|(|\mathcal{M}|+1)$, if such a strategy allows a losing play, it allows one of length at most $|\mathcal{R}|(|\mathcal{R}|+1)(|\mathcal{M}|+1)$. As a consequence, we can check in exponential time whether one of those strategies is winning.

For the echo games, we enumerate all positional strategies for Environment.

Claim 3.62.1. *We can check that a positional strategy σ_{echo} is not winning for Environment in an echo game in non-deterministic exponential time.*

Proof of the claim. Let π be a play won by Controller, at all times if we make a broadcast with letter m, the recent input on 1 is in J_m. For each m broadcast in the play, we can select a sequence of at most $|\mathcal{R}|$ preceding receptions forming an element of the basis of J_m. Those elements witness the fact that the recent input is in J_m.
We can thus easily construct an NFA of exponential size recognising finite plays in which Environment does not win.
Since σ_{echo} is positional, there is an automaton with $|\mathcal{R}|$ states recognising the set of σ_{echo}-plays. We first check whether there is an infinite word whose prefixes are all accepted by the NFA.
If not, we check whether the NFA accepts a play ending with a transition of T.
Finally, we project it to obtain an NFA $\mathcal{A}$ recognising the recent inputs of σ_{echo}-plays not won by Environment. We also build an exponential-size NFA $\mathcal{B}$ recognising $\mathcal{L}(I, (J_m)_{m \in \mathcal{M}})$. As shown in [**BachmeierLS15**], if there is a word in $\mathcal{L}(\mathcal{A})\downarrow$ that is not in $\mathcal{L}(\mathcal{B})\downarrow$, then there is one of polynomial size in $|\mathcal{A}|$ and $|\mathcal{B}|$.
As a consequence, we can check that non-inclusion in non-deterministic exponential time.
In sum, we can check in non-deterministic exponential time that the given strategy is not winning for Environment. ■

This lets us decide in non-deterministic exponential time if there exist $I, (J_m), T$ such that Controller wins the output game and all the echo games. As a result, SafeStrat is in NExpTime. □

It remains to show the lower bound.

Proposition 3.63

SafeStrat is NExpTime-hard on 1BGR.

Proof. We reduce from the exponential grid tiling problem.

Let C be a set of colours containing a border colour B, let $T = \{t_1, \dots, t_k\}$ be a set of tiles and N an integer in unary. We use the alphabet of letters $\mathcal{M} = \{0, 1, \bar{0}, \bar{1}\} \cup T \cup \bar{T}$, where $\bar{T} = \{\bar{t}_1, \dots, \bar{t}_k\}$ is a copy of T.

We design a 1BGR in which Controller wins if and only if there is a valid tiling of the $2^N \times 2^N$ grid with those tiles.

Essentially, Environment may use some agents to broadcast coordinates (x, y) and $(\bar{x}, \bar{y})$ in the grid, respectively using letters $\{0, 1\}$ and $\{\bar{0}, \bar{1}\}$. Environment can also make an agent receive coordinates (x, y) (resp. $(\bar{x}, \bar{y})$), while checking that they all have the same datum. He then makes Controller choose a tile t (resp. $\bar{t}$), which is broadcast with that same identifier. A strategy for Controller amounts to two functions $\tau, \bar{\tau} : [0, 2^N - 1] \times [0, 2^N - 1] \to T$.

The agents that broadcast coordinates (x, y) and $(\bar{x}, \bar{y})$ can then receive tiles t and $\bar{t}$ with their own identifier, and check that:

- If $x = \bar{x}$ and $y = \bar{y}$ then $t = \bar{t}$
- If $x + 1 = \bar{x}$ and $y = \bar{y}$ then $t.right = \bar{t}.left$
- If $x = \bar{x}$ and $y + 1 = \bar{y}$ then $t.up = \bar{t}.down$
- If $x = 0$ (resp. $y = 0$, $x = 2^N - 1$, $y = 2^N - 1$) then $t.left = B$ (resp. *down*, *left*, *right*).

The first item forces Controller to choose $\tau = \bar{\tau}$. The other items make sure that she picks a valid tiling of the grid.

From the initial state Environment chooses between three modes:

- He can receive a sequence of $2N$ bits in $\{0, 1\}$ with the same datum and then let Controller broadcast a letter of T with that same identifier.
- He can receive a sequence of $2N$ bits in $\{\bar{0}, \bar{1}\}$ with the same datum and then let Controller broadcast a letter of $\bar{T}$ with that same identifier.
- He can broadcast a sequence of letters of the form $x_1\bar{x}_1 \cdots x_N\bar{x}_N y_1\bar{y}_1 \cdots y_N\bar{y}_N$ with $x_1, y_1, \dots, x_N, y_N \in \{0, 1\}$, all with his initial datum. He then receives one letter t' of T and one letter $\bar{t}$ of $\bar{T}$ with his initial datum. If $t = t'$ then he stops, otherwise he goes to q_{err}.
- He can broadcast a sequence of letters of the form $x_1\bar{x}_1 \cdots x_N\bar{x}_N y'_1\bar{y}_1 \cdots y'_N\bar{y}_N$ with $x_1, y_1, y'_1, \dots, x_N, y_N, y'_N \in \{0, 1\}$, all with his initial datum. He makes sure that $\langle y_1 \cdots y_N \rangle_2 = \langle y'_1 \cdots y'_N \rangle_2 + 1$. He then receives one letter t' of T and one letter $\bar{t}$ of $\bar{T}$ with his initial datum. If $up(t') \neq down(t)$ or $\langle y'_1 \cdots y'_N \rangle_2 = 0$ and $down(t') \neq B$ or $\langle y_1 \cdots y_N \rangle_2 = 2^N - 1$ and $up(t) \neq B$, he goes to q_{err}. Otherwise he stops without going to q_{err}.

- Similarly, he can broadcast a sequence of letters of the form $x'_1\bar{x}_1 \cdots x'_N\bar{x}_N y_1\bar{y}_1 \cdots y_N\bar{y}_N$ with $x_1, x'_1, y_1, \dots, x_N, x'_N, y_N \in \{0,1\}$, all with his initial datum. He makes sure that $\langle x_1 \cdots x_N\rangle_2 = \langle x'_1 \cdots x'_N\rangle_2 + 1$. He then receives one letter t' of T and one letter $\bar{t}$ of $\bar{T}$ with his initial datum. If $right(t') \neq left(t)$ or $\langle x'_1 \cdots x'_N\rangle_2 = 0$ and $left(t') \neq B$ or $\langle x_1 \cdots x_N\rangle_2 = 2^N - 1$ and $right(t) \neq B$, he goes to q_{err}. Otherwise he stops without going to q_{err}.

If there is a valid tiling, Controller can play the corresponding strategy. In order to reach q_{err}, Environment must make an agent a broadcast coordinates with its initial datum, and then receive two tiles that do not satisfy the conditions mentioned above. The agents that send those tiles must receive exactly $2N$ letters from a, as they are signed by its initial datum. Thus their broadcasts are the tiles of the valid tiling at those coordinates, and the agent will not be able to reach q_{err}, as they match all the conditions.

If there is no valid tiling, Controller's strategy will either induce two different tilings or two identical invalid ones. In both cases Environment can detect the mistake by making an agent a broadcast the coordinates corresponding to the mistake, making two agents answer with the faulty tiles, and make a reach q_{err} by observing the mistake. □

3.8.3 Connection to population protocols with data

Population protocols are a popular model of distributed computation. In a population protocol, an arbitrary number of finite-state agents interact by rendezvous. When two agents meet, they exchange information about their states and update their states accordingly. The agents collectively compute whether their input configuration, i. e., the initial distribution of agents in each state, satisfies a certain predicate. When every fair computation reaches a consensus eventually, and fair runs from the same initial configuration produce the same answer, we say that the protocol is well-specified. It was shown that population protocols compute exactly the predicates of Presburger arithmetic [**AngluinAER07**]. Moreover, well-specification is known to be decidable but as hard as the reachability problem for Petri nets [**EsparzaGLM17**].

Population protocols with unordered data (PPUD) were introduced by Blondin and Ladouceur as a means to compute predicates over arbitrarily large domains [**BlondinL23**]. In this setting, each agent holds a single datum from a set $\mathbb{D}$. When interacting, agents may check (dis)equality of their data. While PP can compute properties like "there are more than 5 agents in state q1", PPUD can express, e. g., "there are more than 2 data with 5 agents each in state q1". In [**BlondinL23**], the authors construct a PPUD computing the absolute majority predicate, i. e., whether a datum is held by more than half of the agents.

For population protocols, the most prominent problem is the design of protocols realising some predicates. This can be seen as a *closed* synthesis problem, where we try to build a system satisfying a specification without any adversarial interactions with the environment. Formally, such a problem is solved by the constructive characterisation of realisable predicates. In the case of population protocols, an important challenge is to find small (few states) and efficient (converging quickly) protocols for predicates.

However, for population protocols with data, the first step is not yet understood: it is an open question to characterise predicates computable by PPUD. In [**BergeremGKMWW24**] it was shown that well-specification is undecidable for this model, but this does not imply anything about potential characterisations.

By contrast, in [**BlondinL23**] the authors characterise the expressive power of immediate observation PPUD (IOPPUD), a subclass of interest in which interactions are restricted to observations. That is, in every interaction, one of the two agents is passive and does not change its state. Interestingly, this mode of communication can be simulated by rendezvous (one of the two agents stays in the same state) and by broadcasts (agents broadcast their current state at all times, and move by receiving those broadcasts). This restriction was already defined on population protocols without data, and it was shown that the complexity of verifying protocols decreases spectacularly when restricted to this case [**EsparzaJRW21**]. Furthermore, in this subcase the well-specification problem is decidable. In fact, it is shown in [**BergeremGKMWW24**] that a much more general problem is decidable, namely the satisfaction of a *generalised reachability expressions.*

Let us define those expressions on 1BNRA.

Let S be a finite set. A simple interval predicate over S is a formula ψ of the form $\dot{\exists}\mathtt{d}_1, \dots, \mathtt{d}_m, \bigwedge_{q \in S} \bigwedge_{j=1}^{m} \#(q, \mathtt{d}_j) \in [A_{q,j}, B_{q,j}]$ where, for all $q \in S$ and $j \in [1,m]$, we have $A_{q,j} \in \mathbb{N}$ and $B_{q,j} \in \mathbb{N} \cup \{+\infty\}$. The dotted quantifiers quantify over *pairwise distinct* data. Formally, given a protocol $\mathcal{R}$ with set of states Q such that $S \subseteq Q$, the predicate ψ is satisfied by a configuration γ if there exist pairwise distinct data $d_1, \dots, d_m \in \mathbb{D}$ such that for all $q \in S$ and $j \in [1,m]$, the number of agents with datum d_j in state q is in $[A_{q,j}, B_{q,j}]$ (resp. $[A_{q,j}, +\infty[$ in the case that $B_{q,j} = +\infty$). An interval predicate over S is a Boolean combination φ of simple interval predicates over S; we define that φ is satisfied by a configuration γ if the simple interval predicates satisfied by γ satisfy the Boolean combination.

Let $\mathcal{R} = (Q, \mathcal{M}, \Delta, q_{init})$ be a protocol with one register. Given a set C of configurations, $\mathtt{Pre}^*(C)$ is the set of configurations from which C is reachable, and $\mathtt{Post}^*(C)$ the set of configurations reachable from C.

Generalised Reachability Expressions (GRE) over $\mathcal{R}$ are produced by the grammar

$$E ::= \varphi \mid E \cup E \mid E^{\mathsf{c}} \mid \mathtt{Post}^*(E) \mid \mathtt{Pre}^*(E),$$

where φ ranges over interval predicates over Q.

Given a GRE E, we define the set of configurations defined by E, denoted $[\![E]\!]_{\mathcal{R}}$, as the set containing all configurations of $\mathcal{R}$ that satisfy the formula, where the predicates are interpreted as above and the other operators are interpreted naturally.

It is then natural to wonder if we can extend the positive results obtained on IOPPUD to 1BNRA. More precisely, the question is whether generalised reachability expressions is decidable on 1BNRA.

Open problem 3.64

Is it decidable, given a 1BNRA $\mathcal{R}$ and a GRE E over $\mathcal{R}$, whether $[\![E]\!]_{\mathcal{R}} = \emptyset$?

Note that the set of initial configurations of a BNRA and the set of configurations with all agents in a given state are expressible with interval predicates. It is therefore not difficult to express the SYNCHRO problem as a particular case of this one. A positive answer to this open problem would thus imply a positive answer to Open problem 3.54.

3.9 Conclusion

In this chapter we presented a powerful model of computation, and showed decidability of verification and strategy synthesis with respect to state reachability. Here are the main takeaways.

First, we showcased our method for distributed controller synthesis by using it for increasingly complex versions of the model. We are always able to match the resulting complexity class with a lower bound, which tends to show that this method makes sense for this model. Then, we found two interesting decidability barriers in systems with lossy broadcasts. We showed that the parameterisation is in a sense forced, as the synthesis problem is undecidable for a fixed set of agents. We also exhibited two seemingly close problems, COVER and SYNCHRO, and proved that one is decidable on BNRA but not the other. We also tested the robustness of our understanding of the model by exploring several natural extensions. Finally, we investigated the case of 1BGR, by relying on the characterisation by invariants obtained in Section 3.5. We also made a connection with existing literature.

In conclusion, this chapter presents a compelling argument in favour of our method. It advances our understanding of networks with unreliable broadcasts by introducing a new perspective, based on the invariants presented throughout the chapter.

Chapter 4

Lock-Sharing Systems

I have six locks on my door all in a row. When I go out, I lock every other one. I figure no matter how long somebody stands there picking the locks, they are always locking three.

Elayne Boosler

Contents

4.1 Introduction

In this chapter we study lock-sharing systems (LSS for short), which are a simple model of concurrent programs whose only means of communication is locks. Processes have access to a pool of locks. Each process is represented by a finite-state automaton whose transitions acquire and release locks. Locks restrict the behaviours of the system, as a process cannot take a lock already held by another process. It is then an interesting challenge to understand the shape of the set of runs of such a system. A similar model was introduced by Kahlon, Ivancić and Gupta in [**KahIvaGup05**], with only two processes, each being a pushdown system. They proved that the verification of regular constraints relating local runs was undecidable, and provided a fine-grained analysis of the decidable cases in that paper and later ones [**Kahlon09**; **KahGup06lics**]. They also showed that detecting deadlocks is decidable under some restrictions. This exact approach contrasts with other ones, which tackle more general systems but use approximations of the set of possible runs.

Here we will focus on finite-state processes. We study the complexity of verifying if a given LSS has a run satisfying a given property. To express those properties, we define regular objectives, which define boolean combinations of local regular constraints. In particular, regular objectives let us express deadlock properties. In the case of finite-state processes, the number of configurations is at most exponential in the size of the system. Consequently, verifying regular objectives is in PSPACE. We will see that even basic decision problems on LSS are PSPACE-complete, (Proposition 4.11).

We present an analysis of restrictions on lock-sharing systems that suffice in order to obtain more tractable complexities than PSPACE for verification. The goal of this part of the chapter is to give a complete picture of complexities for the verification of regular objective and two natural subproblems, which we will present later. The systems we consider are LSS with various restrictions which all prove useful to make some of the problems more tractable.

We mainly focus on two restrictions, 2LSS and nested LSS. The first one demands that each process only accesses two different locks, the second one that each process takes and releases locks as if they were stored in a stack: the process can only release the lock taken latest. Nested locking is assumed in most papers on verification of dynamic systems with synchronization over locks, see for instance [**Brotherson21**; **KahIvaGup05**]. It is also considered as good programming practice, sometimes even enforced syntactically, as in Java through synchronized blocks. The contribution of [**Brotherson21**] consists in an NP algorithm for detecting deadlocks (more specifically, configurations where some subset of processes is blocked as they all try to acquire locks held by other processes of that subset) in concurrent programs. They use a syntax for programs that can be translated to what we call sound nested exclusive LSS in this chapter. As for the systems with two locks per process, they can already exhibit a variety of behaviours. Dijkstra's famous dining philosophers problem matches this constraint.

The 2LSS and nested restrictions have a common point: local runs can be summarised in short descriptions, called *patterns*. Patterns contain enough information to determine whether local runs can be interleaved to form a global run. Some form of patterns for finite runs of nested systems, called acquisition history, was already considered in [**KahIvaGup05**], but was only focused on systems with two processes, on finite runs, and with no considerations of complexity. We show that we can extend the techniques to handle much larger classes of specifications, in the framework of verification.

In order to do this, we define the notion of *patterns* for finite and infinite local runs. We show that we can check if a set of local runs can be interleaved to form a (fair) global run simply by checking a short list of conditions on their patterns (Lemma 4.28 for 2LSS). This allows us to verify the system against local specifications with the following steps:

- First we guess for each process a pattern, and check that the process has a bad local run with that pattern.
- Then we check that those local runs can be scheduled into a global run by checking the aforementioned conditions on their patterns.

Thus we avoid exploring the product of all processes.

This approach yields NP algorithms for the verification of (boolean combinations of) local specifications for 2LSS and nested LSS. For 2LSS, with additional constraints, we even obtain PTIME algorithms for some specific objectives. More precisely, we show that we can check in polynomial time if there is a run in which a given process p gets blocked indefinitely (process deadlock problem), on 2LSS where whenever a process may acquire a lock, that is the only thing it can do (exclusive). We also show that we can decide if there is a run in which every process gets blocked (global deadlock problem) in polynomial time for 2LSS where processes always have an available action (locally live). The complexity results for verification are summarised in Table 4.1. We provide matching lower bounds for all complexities above P.

We also tackle controller synthesis for this model. We show that for general LSS this is an undecidable problem. On the other hand, for 2LSS and nested LSS, we use once again the patterns defined in the verification part of the chapter. This time the procedure follows those lines:

- First we guess for each process a set of patterns (called a *behaviour*) and check that there is a local strategy guaranteeing that the patterns of all possible bad local runs are in this set. The existence of that strategy is checked using a two-player regular game.
- Then we make a universal guess of a pattern in the behaviour of each process.
- Finally, we check that those patterns represent local runs that can be scheduled into a global run.

This yields an NEXPTIME algorithm in general for regular objectives. This bound is unfortunately tight. In front of this high complexity, we present some subcases which are more tractable. We show that the complexity can be improved to Σ_2^P for 2LSS for some subproblems, and even to P when we consider the global deadlock problem for locally live exclusive 2LSS.

4.1.1 Structure of the chapter

In Section 4.2, we will start by defining the central model of this chapter, lock-sharing systems. We will also introduce the specifications considered in this chapter, which we call regular objectives. We also introduce two central sub-problems, the global deadlock problem and the process deadlock problem. The first one asks whether all processes may end up blocking each other, and the second one whether some give process p may get blocked forever. Finally, we define several restrictions on LSS, which we will use to mitigate the complexity of the problems at hand.

In Section 4.3 we study the verification of regular objectives on LSS. We show that in general the problem is PSPACE-complete. Then, we restrict ourselves to LSS in which each process uses at most two locks (2LSS). We first give a representative example of the method we use in this case, using the dining philosophers. We then define a central tool of this chapter, called patterns. Those are a classification of local runs into finitely many classes. We show that knowing the patterns of some local runs is enough to decide whether they can be combined into a global run. We use this fact to prove an NP upper bound on the verification of regular objectives for 2LSS.

Sections 4.4 and 4.5 are respectively dedicated to the study of the process deadlock problem and the global deadlock problem in the case of 2LSS. Both problems are NP-complete, but we can find subcases with a polynomial-time complexity.

In Section 4.6 we focus on a different restriction on LSS, called nested locking. In this case also, we can define stair patterns with which we characterise schedulability of local runs. We conclude that the verification of regular objectives is NP-complete in that case. We also show that the restrictions considered for the 2LSS case do not improve this complexity for nested LSS.

Finally, in Section 4.7 we lift our results from verification to the synthesis problem, where some states are controllable and we try to find local strategies ensuring that no global run has an undesired behaviour. We show that this problem is undecidable for LSS. By contrast, for 2LSS and nested LSS, we can use sets of patterns as local specifications to guarantee the global objective. We reduce the synthesis problem to a two-player game.

Our results concerning verification are summarised in Table 4.1.

	Regular objectives	**Global deadlock**	**Process deadlock**
LSS	PSPACE (Prop. 4.11)	PSPACE (Prop. 4.11)	PSPACE (Prop. 4.11)
Nested	NP (Thm. 4.53)	NP (Thm. 4.53)	NP (Thm. 4.53)
2LSS	NP (Thm. 4.18)	NP (Thm. 4.29)	NP (Thm. 4.19)
Locally live 2LSS	NP (Thm. 4.18)	P (Thm. 4.30)	NP (Thm. 4.19)
Exclusive 2LSS	NP (Thm. 4.18)	NP (Thm. 4.29)	P (Prop. 4.20)

Table 4.1: A summary of complexities of verifying LSS in the cases we consider in this work. All problems are proven complete for the indicated complexity class, except the ones solvable in polynomial time.

The PSPACE complexity in the first column comes from the fact that three locks suffice to allow processes to synchronise and progress in lock-step manner. In the case of 2LSS and nested LSS we can classify runs with respect to their patterns, and characterise schedulability using those patterns. This lets us verify those systems by guessing short patterns describing the local runs, and then check that they represent local runs that can be tangled into a global run satisfying the specification, Moreover, with some extra assumptions, we can represent patterns of 2LSS as a graph and characterise the existence of deadlocks on that graph, giving us polynomial-time algorithms.

4.2 Definitions

In this section we define the model at hand in this chapter, the decision problems we will study, and some restrictions that we will consider in order to lower the complexities.

Given a set $\mathbf{L}$ of locks, we define the set of operations on $\mathbf{L}$ as

$$\mathsf{Op}(\mathbf{L}) = \{\mathtt{acq}_\ell, \mathtt{rel}_\ell, \mathtt{nop} \mid \ell \in \mathbf{L}\}.$$

Definition 4.1 ▶ Lock-sharing system

A *lock-sharing system* (LSS for short) is a tuple $\mathcal{S} = (\mathsf{Proc}, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathbf{L}, \mathsf{op})$, made of

- Proc a finite set of processes,
- $(\mathcal{A}_p)_{p\in\mathsf{Proc}}$ a family of deterministic automata,
- $\mathbf{L}$ a set of locks,
- $\mathsf{op} : \Sigma \to \mathsf{Op}(\mathbf{L})$ a function mapping letters to operations, with Σ the set of letters used by the automata $(\mathcal{A}_p)_{p\in\mathsf{Proc}}$.

Each transition system $\mathcal{A}_p$ is given as a tuple $(S_p, \Sigma_p, \delta_p, init_p)$ with S_p a finite set of states, $init_p$ the initial state and $\delta_p : S_p \times \Sigma_p \to S_p$ the transition function.

For all $p \in \mathsf{Proc}$ we define $\mathbf{L}_p = \{\ell \in \mathbf{L} \mid \exists a \in \Sigma_p, \mathsf{op}(a) = \mathtt{acq}_\ell\}$ the set of locks p may acquire. We assume that the Σ_p are disjoint for convenience, and we define $\Sigma = \bigsqcup_{p\in\mathsf{Proc}} \Sigma_p$. A *2LSS* is an LSS where every $\mathbf{L}_p$ has two elements.

We fix an LSS $\mathcal{S} = (\mathsf{Proc}, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathbf{L})$ for the rest of this section.

A *local configuration* of process p is a state from S_p together with the locks p currently owns: $(s, B) \in S_p \times 2^{\mathbf{L}_p}$. The initial configuration of p is $(init_p, \emptyset)$, namely the initial state with no locks. A transition between configurations $(s, B) \xrightarrow{a} (s', B')$ exists when $\delta_p(s, a) = s'$ and one of the following holds:

- $\mathsf{op}(a) = \mathtt{nop}$ and $B = B'$;
- $\mathsf{op}(a) = \mathtt{acq}_\ell$, $\ell \notin B$ and $B' = B \cup \{\ell\}$;
- $\mathsf{op}(a) = \mathtt{rel}_\ell$, $\ell \in B$, and $B' = B \setminus \{\ell\}$.

A *local run* of $\mathcal{A}_p$ is defined as a finite or infinite sequence of transitions of the form $\varrho_p = (s_0, B_0) \xrightarrow{a_1}_p (s_1, B_1) \xrightarrow{a_2}_p \cdots$. It is *initial* if $(s_0, B_0) = (init_p, \emptyset)$. Because a local run is determined by its sequence of letters, we will identify local runs of p and words over Σ_p. A local run is *neutral* if it is finite and the locks held at the end and at the beginning are the same, and are not released during the run.

For instance, in the LSS from Figure 4.1, the local run $(B, \{1\}) \to (C, \{1, 2\}) \to (B, \{1\}) \to (D, \{1, 3\}) \to (F, \{1\})$ is neutral. However, $(C, \{1, 2\}) \to (B, \{1\}) \to (C, \{1, 2\})$ is not.

A *global configuration* is a family of local configurations $C = (s_p, B_p)_{p\in\mathsf{Proc}}$ provided the sets B_p are pairwise disjoint: $B_p \cap B_q = \emptyset$ for $p \neq q$. This is because **a lock can be held by at most one process at a time**. The initial configuration is the tuple of initial configurations of all processes $(init_p, \emptyset)_{p\in\mathsf{Proc}}$.

Runs of such systems are *asynchronous*, with transitions between two consecutive configurations done by a single process: $C \xrightarrow{(p,a)} C'$ if we have $(s_p, B_p) \xrightarrow{a}_p (s'_p, B'_p)$ and $(s_q, B_q) = (s'_q, B'_q)$ for every $q \neq p$. A *global run* (or just run) is a sequence of transitions between global configurations. The projection of a global run ϱ on process p is the sequence of transitions taken by p in ϱ, written ϱ_p or $\varrho|_p$.

In what follows we will assume that each process keeps track in its state of the set of locks it owns. Note that this assumption does not compromise the complexity results provided there is a bound on the number of locks a process can access: the number of states is then multiplied by a constant factor.

Definition 4.2

A process of an LSS is *sound* if its transition system $\mathcal{A}_p$ keeps track of the set of locks it has in its states. Formally, let $\mathcal{A}_p = (S_p, \delta_p, init_p)$, p is sound if there exists a function $\texttt{holds}_p : S_p \to 2^{\mathbf{L}_p}$ such that:

- for all local run $\varrho_p = a_1 a_2 \cdots a_n$ ending in a state s, we have $(init_p, \emptyset) \xrightarrow{a_1} \cdots \xrightarrow{a_n} (s, \texttt{holds}_p(s))$.
- for all states $s \in S_p$, there is no outgoing transition of s that acquires a lock in $\texttt{holds}_p(s)$ or releases a lock that is not in $\texttt{holds}_p(s)$.

An LSS is sound if all its processes are.

Remark 4.2.1. *Soundness can be easily checked in polynomial time on a given LSS: it suffices to set* $\texttt{holds}(init_p)$ *to* $\emptyset$*, apply a DFS to compute candidates for* $\texttt{holds}(s)$ *for all states, and then check consistency of* $\texttt{holds}$ *with respect to each transition.*

Example 4.2.1. *Take a look at the LSS displayed in Figure 4.1. This LSS is sound as both its processes are. However, it is not nested, as the upper process is not: for instance the local run* $(A1, \emptyset) \to (B1, \{1\}) \to (C1, \{1,2\}) \to (E1, \{1,2,3\}) \to (D1, \{1,3\})$ *does not satisfy the nested discipline.*

We want to be able to define deadlocks in terms of languages of runs. To this end, we have to restrict our attention to *process-fair runs*, in which every process is either blocked after some point or executes an action infinitely many times. This is often called strong fairness in the literature. This way if a process stops doing anything after some point in a run, it means it is blocked.

Definition 4.3 ▶ Process-fairness

A run ϱ is called *process-fair* if for all $p \in \mathsf{Proc}$, either ϱ contains infinitely many actions of $\mathcal{S}_p$, or there is a point after which no action of p can ever be executed at any moment in the run.

Definition 4.4 ▶ Deadlocks

A process-fair run yields a *global deadlock* if it is finite, i.e., at some point there are no actions that can be executed in any of the processes, and the system cannot advance any more.

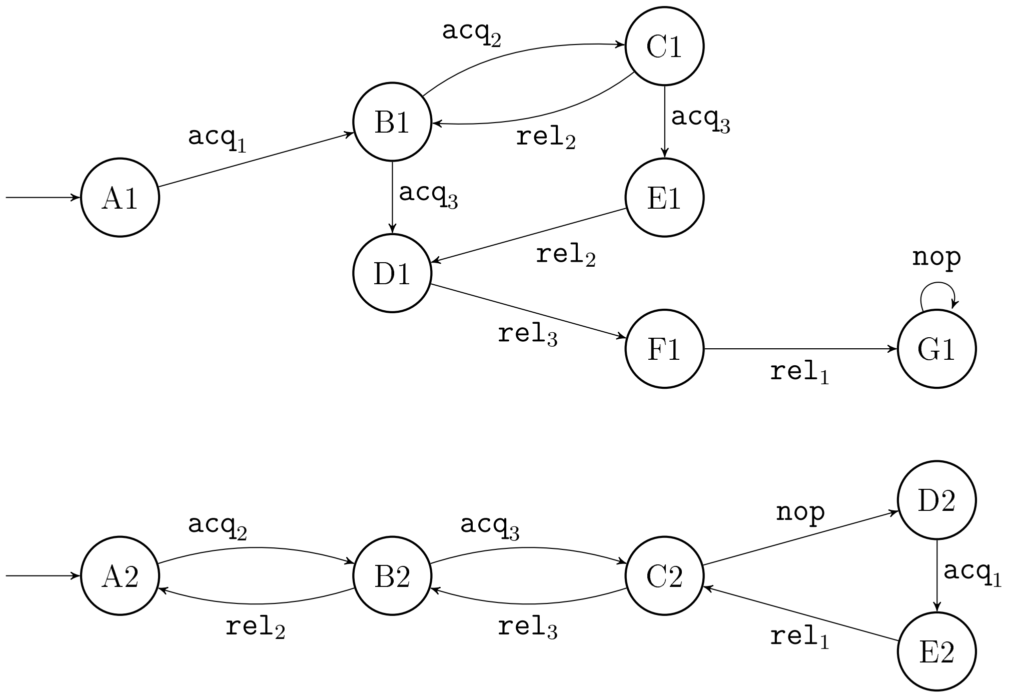


Figure 4.1: Example of an LSS made of two processes. For readability the transitions are not labelled with letters of Σ but with the associated operations.

Example 4.2.2. *In the LSS from Figure 4.1, we can obtain a global deadlock as follows: First make the upper process go to $B1$ while taking lock 1, then make the other process go to $D2$ while taking locks 2 and 3. After this, no process can take any action and thus we have a global deadlock.*

Adding a transition from $B1$ to $A1$ releasing lock 1 would prevent this: The upper process could then always release a lock or do nop *from any state except $A1$. Furthermore, in a process-fair run, after acquiring 1 and getting to $E2$ the lower process would have to eventually release 1 by going to $C2$. As a consequence, the upper process could not get stuck in $A1$ either.*

In all that follows we will have to work with finite and infinite words simultaneously as LSS executions may be finite or infinite. To lighten notations, we will consider that we work with automata on infinite words. We implicitly assume that we use a dummy letter $\square$, and that finite words are padded with an infinite suffix $\square^\omega$, so that we can express objectives as languages of infinite words.

We will now define the set of properties we aim to verify. This class of objectives is inspired by deterministic Emerson-Lei automata, introduced in [**EmersonL1987**], which we will use in several proofs. These automata have advantageous properties, such as the ability to be easily intersected and complemented in polynomial time.

Our objectives are defined with one automaton per process and a single formula expressing a condition on which states (among the ones of all automata) are seen infinitely often.

Definition 4.5

A *regular objective* is a pair $((\mathcal{B}_p)_{p \in \mathsf{Proc}}, \varphi)$ such that each $\mathcal{B}_p$ is a deterministic automaton with a set of states Q_p over the alphabet Σ_p, and φ is a boolean formula over the set of variables $\{\inf_{p,q} \mid p \in \mathsf{Proc}, q \in Q_p\}$.
Let ϱ be a process-fair run, and for each p let ϱ_p be its projection on $\mathcal{S}_p$. We say that ϱ satisfies a regular objective $((\mathcal{B}_p)_{p \in \mathsf{Proc}}, \varphi)$ if φ is satisfied by the valuation evaluating $\inf_{p,q}$ to $\top$ if and only if the unique run of $\mathcal{L}(\mathcal{B}_p)$ on ϱ_p goes through q infinitely many times.

We argue that these specifications are quite expressive and at the same time allow us to stay in reasonably low complexity classes.

Regular objectives are expressive. They can express properties such as reachability (with local or global configurations) or safety, as well as properties related to deadlocks, such as global deadlock: As we focus on process-fair runs, a local projection of a run is finite if and only if the corresponding process is blocked at some point and has no available action for the rest of the run. Hence, we can express for instance a global deadlock with an objective requiring the local run of every process p to be finite. Moreover, the flexibility of boolean formulas allows us to relate configurations between processes: say each process has to decide between 0 and 1, then we can express agreement by demanding that they all select 0 or all 1. Regular objectives are furthermore closed under boolean combinations. They can be complemented by simply taking the negation of the formula φ, and intersected in polynomial time by taking the product automaton for each process and adapting the formula.

Complexity blows up quickly with more expressive objectives Regular objectives only restrict the shape of local runs without any requirement on their interleaving. Restrictions on interleavings would lead to PSPACE-hardness very quickly. For instance, an objective expressing that processes should operate in rounds in which they all use the same letter would let us reduce the problem of emptiness of the intersection of a family of DFAs easily.

We study the problem of finding a run satisfying some given regular objective.

Definition 4.6 ▶ ***Regular verification problem***

Input: a sound LSS $\mathcal{S}$ and a regular objective $((\mathcal{B}_p)_{p \in \mathsf{Proc}}, \varphi)$
Output: Is there a process-fair run of $\mathcal{S}$ satisfying the objective?

Note that we define the problem existentially: we are looking for a bad run, hence the given objective should express the set of runs that we want to avoid. We use this formulation as it simplifies a bit our proofs, and as regular objectives are easy to complement.

We will also study two specific cases of this problem: global deadlocks and process deadlocks. We consider those as the simplest interesting objectives in this framework. We will demonstrate that these problems can be decided in PTIME in some subcases. Additionally, we will use these cases for complexity lower bounds, showing that the complexities do not arise from complicated specifications.

Definition 4.7 ▶ *Process deadlock problem*

Input: a sound LSS $\mathcal{S}$ and a process p.
Output: Is there a process-fair run of $\mathcal{S}$ whose projection on p is finite?

Definition 4.8 ▶ *Global deadlock problem*

Input: a sound LSS $\mathcal{S}$.
Output: Is there a process-fair run of $\mathcal{S}$ whose projection on every process is finite?

In addition to restrictions on the decision problem, we will consider two assumptions on the model, which will also let us improve the complexity. The first one is locally live processes. It forbids processes to block by themselves because they are in a state with no outgoing transition. This makes sense as a deadlock occurring only because some process is in a state with no outgoing transitions can be considered a degenerate case. However, it can be interesting to study the model without this restriction as we can then easily model processes that can crash during the execution.

Definition 4.9 ▶ *Locally live*

An LSS is *locally live* if it is sound and all states have an outgoing transition. In other words, the process cannot be blocked by itself: the only cause of blockage is waiting for locks that are never released.

As our last definition in this part, we introduce *exclusive* LSS, in which a process that can acquire a lock cannot do any other operation from the same state.

Definition 4.10 ▶ *Exclusive*

A process is *exclusive* if its transition system $\mathcal{A}_p$ is such that for all states s, if s has an outgoing transition acquiring some lock ℓ, then all other outgoing transitions from s acquire that same lock ℓ. An LSS is exclusive if all its processes are.

This restriction can be satisfied easily when translating programs into LSS: it suffices to replace every $\mathtt{acq}_\ell$ transition from a state s with a `nop` transition to an intermediate fresh state and a $\mathtt{acq}_\ell$ transition from there. The drawback is that this transformation makes deadlocks more likely to appear in the model. Note that the locally live and exclusive restrictions are orthogonal, in the sense that neither of them implies the other.

4.3 Verification of regular objectives

In this section we address the general problem of verification of 2LSS against regular objectives.

4.3.1 An introductory case study

To understand the interest of 2LSS, let us have a look at the Dining philosophers problem.

Example 4.3.1. *The dining philosophers problem can be formulated as control problem for a lock-sharing system. We set* $\mathsf{Proc} = \{1, \dots, n\}$ *and* $\mathcal{L}ocks = \{\ell_1, \dots, \ell_n\}$ *as the set of locks. For every process* $p \in \mathsf{Proc}$*, process* $\mathcal{A}_p$ *is as in Fig. 5.1, with the convention that* $\ell_{n+1} = \ell_1$*. When a philosopher p is hungry, she has to get both the left* (ℓ_p) *and the right* (ℓ_{p+1}) *fork to eat. She may however take them in two ; actions* $left$ *and* $right$ *are controllable.*

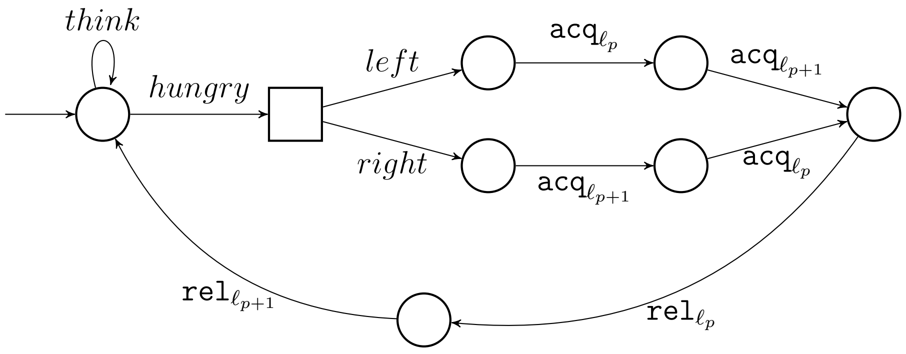


Figure 4.2: A dining philosopher p.

Since each process uses only two locks, we can draw the lock distribution as a graph, where locks are vertices and process label edges. In this example we obtain a cycle.

Let us take the following specification: Every philosopher that takes the action *hungry* should later come back to the initial state (meaning that she managed to eat). Let us furthermore consider only fair runs: a philosopher can stay idle indefinitely only if she has no available action from some point on.

We can show that the satisfaction of that specification by a strategy over fair runs depends only on one thing: which local strategies only take $left$, which ones only take $right$, and which ones may take both depending on the run seen so far. We draw this as a graph, as in Figure 4.3.

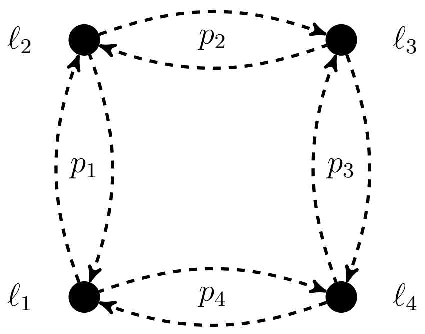


Figure 4.3: A graph representing the choices of each philosopher, with $n = 4$. Each p_i chooses either one or both edges, which indicate orders in which locks are taken.

Once we do that, we translate the previous problem into a graph problem: We want to choose for each p_i either one or two edges so that a cycle can be formed with the chosen edges. This is easy to do, for instance by making each p_i select the edge from ℓ_i to ℓ_{i+1}.

We can thus conclude that there is a run leading to a deadlock.

Let us now illustrate an idea for controller synthesis. Now suppose that the square state in Figure 5.1 is controllable. Then the graph in Figure 4.3 is interpreted differently.

Each process may always take the left lock first, always the right one, or sometimes one and sometimes the other. In the graph this is represented by a subset of edges between the two locks. Each process picks either just the edge from its left to its right lock, just the edge from right to left, or both. If the edges that were picked contain a cycle, then we can create a deadlock.

To get a winning strategy, we must select a subset of edges for each process such that no cycle is formed. This is easily done by making one process go clockwise, another anticlockwise, and the rest in any direction. So a winning strategy is for instance to make one of the processes always take its left then right locks, another one always take its right lock first, and pick any strategy for the others. This guarantees the absence of deadlocks.

4.3.2 Verification of general LSS

In order to justify the restriction to 2LSS, we prove that the general verification of LSS against regular objectives is PSPACE-complete, even for locally live exclusive LSS. This proof also shows how we can use locks to communicate between processes. A similar technique was used in [**KahIvaGup05**] to show that the verification of systems made of two pushdown processes communicating with locks is undecidable. We will use this mechanism again in other proofs.

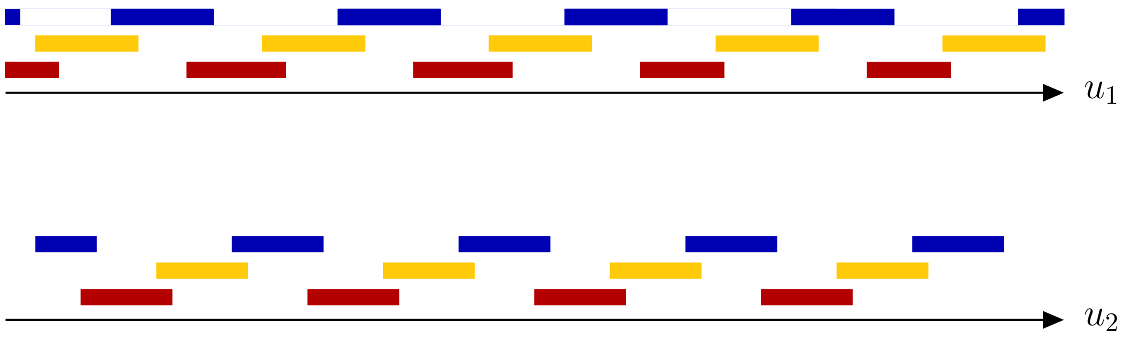


Figure 4.4: A way to have two local runs with only one possible interleaving.

Example 4.3.2. *Take a look at the two local runs in Figure 4.4. They use the same three locks, each corresponding to one colour.*

There is only one way to interleave them, assuming the upper one starts while already holding the red and blue locks (we need a few more details if all runs start with no locks).

In the following we develop this idea to force processes to choose a common sequence of letters $\{a, b\}^*$ during the execution.

We use two extra locks a and b, initially held by the upper process. After each round of lock acquisitions, the first process releases and acquires either a or b. Similarly, after each round the second process acquires and releases a or b.

As the two processes are forced to make rounds in a lockstep manner, in order to continue the run, the two processes must agree on the sequence of letters they choose.

Proposition 4.11

The regular verification problem is PSPACE-complete for LSS in general. PSPACE-hardness already holds for the process deadlock problem for sound locally live exclusive LSS even with a fixed number of locks per process.

Proof. The PSPACE upper bound is easy to obtain: It suffices to guess a state s_p in each $\mathcal{A}_p$ and s'_p in each $\mathcal{B}_p$, and then guess a sequence of letters in $\mathcal{S}$ while keeping track of the states reached by that sequence in the $\mathcal{A}_p$ and $\mathcal{B}_p$.

If we reach a configuration with each $\mathcal{A}_p$ in state s_p and each $\mathcal{B}_p$ in s'_p, we start memorising the set of visited states in each $\mathcal{B}_p$. If we reach that configuration again, we stop and accept if and only if the set of visited states in the $\mathcal{B}_p$ satisfies φ.

The difficulty is to obtain the PSPACE-hardness with a fixed number of locks per process. To do so we reduce the emptiness problem for the intersection of a set of deterministic automata. Without loss of generality we will assume that there are at least two automata and that they are all over the alphabet $\{a, b\}$. We will furthermore assume that all languages are subsets of $aa(ba + bb)^*$. The problem stay PSPACEcomplete with this restriction: take any family of DFAs, replace every transition reading a (resp. b) by a sequence of two transitions reading ba (resp. bb), and then make all automata read aa at the start. The intersection of the languages of the resulting automata is empty if and only if the intersection of the initial languages was.

Let $\mathcal{A}_1, \cdots, \mathcal{A}_n$ (with $n \geq 2$) be automata, with, for each $1 \leq i \leq n$, $\mathcal{A}_i = (S_i, \{a, b\}, \delta_i, init_i, F_i)$. We construct a sound locally live exclusive LSS $\mathcal{S}$ as follows:

For each $1 \leq i \leq n$ we have a process p_i which is in charge of simulating $\mathcal{A}_i$, plus an extra process q. The set of locks is $\mathbf{L} = \{a, b\} \cup \{x_i, y_i, z_i key_i \mid 1 \leq i \leq n\} \cup \{t_i \mid 1 \leq i \leq n+1\}$. Locks a and b will be used to transmit sequences of letters, the x_i and y_i to synchronise all processes in a lockstep manner, and the key_i to handle the initialisation of the system. The locks t_i are used to enable a global deadlock when all processes have reached a final state.

For all i, p_i accesses locks $a, b, x_i, y_i, z_i, key_i, t_i$, as well as $x_{i+1}, y_{i+1}, z_{i+1}, key_{i+1}, t_{i+1}$ if $i < n$ and x_1, y_1, z_1, t_{n+1} if $i = n$. Process q only uses locks t_{n+1} and t_1. Thus a process uses at most 12 locks in total [1].

For all $1 \leq i \leq n$ and ℓ accessed by p_i, we have two actions $\mathtt{acq}^i_\ell$ and $\mathtt{rel}^i_\ell$, with which p_i acquires and releases lock ℓ.

For all i we define the following two sequences of actions. For $i = n$ the $i + 1$ indices are replaced by 1.

$$\text{SEND}_i = \mathtt{rel}^i_{x_i}\mathtt{acq}^i_{z_i}\mathtt{rel}^i_{y_i}\mathtt{acq}^n_{x_i}\mathtt{rel}^i_{z_i}\mathtt{acq}^i_{y_i}$$

$$\text{REC}_i = \mathtt{acq}^i_{x_{i+1}}\mathtt{rel}^i_{z_{i+1}}\mathtt{acq}^i_{y_{i+1}}\mathtt{rel}^i_{x_{i+1}}\mathtt{acq}^i_{z_{i+1}}\mathtt{rel}^i_{y_{i+1}}$$

They allow us to synchronise processes in a lockstep manner.

The transition system of each process p_i is designed as follows: For each $i < n$, we start with $\mathcal{A}_i$, and we replace every transition labelled a with a sequence of transitions labelled by the sequence of actions $\text{REC}_i\text{SEND}_i\mathtt{acq}^i_a\mathtt{rel}^i_a$. Similarly, we replace each b transition with the sequence $\text{REC}_i\text{SEND}_i\mathtt{acq}^i_b\mathtt{rel}^i_b$. Furthermore we add transitions so that process p_i

[1] This can be improved: a proof using 6 locks per process can be found in [**Mascle23**], but we allow a couple more locks here to simplify the proof.

starts by executing $\textsc{start}_i = \mathtt{acq}^i_{key_{i+1}}\mathtt{acq}^i_{x_i}\mathtt{acq}^i_{y_i}\mathtt{acq}^i_{z_{i+1}}\mathtt{rel}^i_{key_{i+1}}\mathtt{acq}^i_{key_i}$ before entering the initial state of $\mathcal{A}_i$.

If $i = n$, we proceed similarly, but we replace a transitions with $\textsc{send}_i\textsc{rec}_i\mathtt{rel}^i_a\mathtt{acq}^i_a$ (and similarly for b) and add the sequence $\textsc{start}_n = \mathtt{acq}^n_{x_n}\mathtt{acq}^n_{y_n}\mathtt{acq}_{z_1}\mathtt{acq}_a\mathtt{acq}_b\mathtt{acq}^n_{key_n}$.

Finally, for each i we add an extra state s_i and put a transition labelled $\mathtt{nop}$ from each state of F_i to s_i. From s_i we add transitions taking t_i and then t_{i+1}, and then releasing both. Process q simply has an initial state s_q, from which it takes t_{n+1} and then t_1 and then releases both.

This system is sound and locally live, but not exclusive as it is: if there is a state of F_i in $\mathcal{A}_i$ with both a and b transitions, the corresponding state of p_i will have transitions acquiring x_{i+1} and another transition $\mathtt{nop}$ going to s_i. We can solve this by adding an intermediate transition with no effect before each acquisition. This does not affect the following reduction.

Let us start by observing that in this system we can reach a global deadlock if and only if we can block process q. This will allow us to use a single reduction for the PSPACE-completeness of the global deadlock problem and the process deadlock problem.

Claim 4.11.1. *There is a process-fair run leading to a global deadlock if and only if there is a process-fair run with a finite projection on q.*

Proof of the claim. Clearly a process-fair run yielding a global deadlock must have a finite projection on q.
On the other hand, if we have a process-fair run ϱ with a finite projection on q, then q must be unable to take t_{n+1} or t_1. In the first case, it means that p_n holds t_{n+1}, but since the run is process-fair and p_n is able to release this lock, it must release it eventually. Therefore q must be holding t_{n+1} and waiting for t_1. Hence p_1 must be holding t_1 forever. The only way this can happen is if t_2 is held forever by p_2, and so on. We conclude that all p_i hold t_i and wait for t_{i+1}: this is a global deadlock. ■

Now we claim that there is a word accepted by all $\mathcal{A}_i$ if and only if there is a run of this LSS with a finite projection on q.

One direction is easy. Say there is a word $u = c_1c_2\cdots c_m$ in the intersection of the languages of the $\mathcal{A}_i$. Then we apply the following steps:

- we start by executing all $\textsc{start}_i$ sequences for all i in increasing order.
- After that, for each $1 \leq j \leq m$ in increasing order, we do the following: we make p_1 execute $\textsc{rec}_1$ while p_n executes $\textsc{send}_n$. This is feasible by alternating one step of each process, starting with p_n.
- Then, for each $1 \leq i \leq n-1$ (in increasing order), we make p_i execute $\textsc{send}_i$ and p_{i+1} execute $\textsc{rec}_i$. Like before, this can be done by making them alternate.
- Then, p_n releases c_j and all other p_i acquire and release c_j. Finally, p_n acquires back c_j.

This run projects on p_i as $\textsc{start}_i\textsc{rec}_i\textsc{send}_i\mathtt{acq}_{c_1}\mathtt{rel}_{c_1}\cdots\textsc{rec}_i\textsc{send}_i\mathtt{acq}_{c_m}\mathtt{rel}_{c_m}$ if $i < n$ and $\textsc{start}_n\textsc{send}_n\textsc{rec}_n\mathtt{rel}_{c_1}\mathtt{acq}_{c_1}\cdots\textsc{send}_n\textsc{rec}_n\mathtt{rel}_{c_m}\mathtt{acq}_{c_m}$ if $i = n$. For all i, as u is in the language of $\mathcal{A}_i$, p_i can execute this run locally. It can be easily checked that this sequence of operations is a valid run, hence we have a run with no process blocked.

As u is accepted by all $\mathcal{A}_i$, after executing the sequence above each process p_i ends up in a state of F_i, and thus they can all go to s_i one after the other.

We make each p_i acquire t_i and q acquire t_{n+1}. Then all p_i are blocked as they can only acquire t_{i+1}, which is held by p_{i+1} (resp. q if $i = n$). Process q is also blocked as it tries to acquire t_1, held by p_1.

For the other direction, suppose there is a process-fair run ϱ with a finite projection on q. As we saw in the proof of Claim 4.11.1, this is only possible if all p_i hold t_i and wait for t_{i+1}.

As a consequence, for all $i < n$ the projection $\varrho|_{p_i}$ must be of the form

$$\text{START}_i\text{REC}_i\text{SEND}_i\mathtt{acq}_{c_1^i}\mathtt{rel}_{c_1^i}\cdots\text{REC}_i\text{SEND}_i\mathtt{acq}_{c_{m_i}^i}\mathtt{rel}_{c_{m_i}^i}\mathtt{nop}\ \mathtt{acq}_{t_i}$$

with $c_1^i \cdots c_{m_i}^i \in \mathcal{L}(\mathcal{A}_i)$.

Meanwhile, the projection $\varrho|_{p_n}$ must be of the form

$$\text{START}_n\text{SEND}_n\text{REC}_n\mathtt{rel}_{c_1^n}\mathtt{acq}_{c_1^n}\cdots\text{SEND}_n\text{REC}_n\mathtt{rel}_{c_{m_n}^n}\mathtt{acq}_{c_{m_n}^n}\mathtt{nop}\ \mathtt{acq}_{t_n}$$

with $c_1^n \cdots c_{m_n}^n \in \mathcal{L}(\mathcal{A}_n)$.

We prove the following claim:

Claim 4.11.2. *For all i, $c_1^i \cdots c_{m_i}^i = c_1^n \cdots c_{m_n}^n$.*

Proof of the claim. First of all note that for all $i \leq n-1$ if p_{i+1} has finished executing START_{i+1} then it holds key_{i+1} and will never release it, hence p_i either has executed START_i in full or has not begun executing it. In the second case, p_i will never be able to advance, which is impossible as $\varrho|_p$ is not empty. As a result, after p_{i+1} has executed START_{i+1}, p_i must have executed START_i.

Consider the moment of the run when p_n finishes executing START_n. Then all other processes p_i have executed their START_i. In particular, from this point on each p_i always holds one of x_i, y_i, z_i and at least one of $x_{i+1}, y_{i+1}, z_{i+1}$.

In consequence, it is clear that whenever p_{i+1} executes SEND_{i+1}, p_i must execute REC_i and whenever p_i executes REC_i, p_{i+1} must execute SEND_{i+1}

Whenever p_n executes $\text{SEND}_n\text{REC}_n$, each other p_i is forced to execute $\text{REC}_i\text{SEND}_i$. This means that the sequence of locks $c_1^n \cdots c_{m_n}^n$ released by p_n between rounds of $\text{SEND}_n\text{REC}_n$ must be a suffix of $c_1^i \cdots c_{m_i}^i$ for all i.

Recall that we assumed that the languages of all automata are included in $aa(ba+bb)^*$. This means that the only way a word from those languages can be a suffix of another one is if they are equal.

As a consequence, $c_1^i \cdots c_{m_i}^i = c_1^n \cdots c_{m_n}^n$ for all i. ■

We have shown that this system had a run in which each q is blocked indefinitely if and only if there is a word accepted by all $\mathcal{A}_i$. This proves the PSPACE-completeness of the process deadlock problem. Furthermore, by Claim 4.11.1, the same reduction shows that the global deadlock problem is PSPACE-complete.

□

4.3.3 Verification of 2LSS using locking patterns

In this section we define patterns for 2LSS. These are summaries of bounded size of the operations executed during a run, which contain enough information to tell if local runs can be combined into a global one.

Definitions for locking patterns

A finite run is *risky* if it ends in a state where all outgoing transitions acquire a lock.

Definition 4.12 ▶ Main functions

Given a finite local run ϱ_p of a process p, $\textsc{Holds}(\varrho_p)$ is the set of locks held after executing ϱ_p.
If ϱ_p is risky we define $\textsc{Blocks}(\varrho_p)$ as the set of locks that can be acquired from the last state reached by ϱ_p. Formally, if ϱ_p ends in state s_p, then we define $\textsc{Blocks}_p = \{\ell \mid \exists a, s', \delta(s_p, a) = s' \text{ and } \mathtt{op}(a) = \mathtt{acq}_\ell\}$.
We extend $\textsc{Holds}$ to infinite runs by setting $\textsc{Holds}(a_1a_2\cdots)$ as the set of locks kept indefinitely by p after some point. Formally, we define $\textsc{Holds}(a_1a_2\cdots) = \bigcup_{i\in\mathbb{N}}\bigcap_{j>i}\textsc{Holds}(a_0\cdots a_j)$.
We also define $\textsc{Inf}(\varrho)$ as the set of sets of locks that p owns infinitely often when executing ϱ_p:

$$\textsc{Inf}(a_1a_2\cdots) = \{A \subseteq \mathbf{L}_p \mid A = \textsc{Holds}(a_1\cdots a_i) \text{ for infinitely many } i\}$$

We also define the *trace* of a local run $\varrho = a_1a_2\cdots$ as the sequence of sets of locks held by p in this run, i.e., $tr(\varrho) = \textsc{Holds}(\varepsilon)\textsc{Holds}(a_0)\textsc{Holds}(a_0a_1)\cdots$.

We start by defining patterns of *finite runs*.

Definition 4.13 ▶ *Finitary patterns*

Let p be a process, $\mathbf{L}_p = \{\ell_1, \ell_2\}$ its locks. Let $\varrho_p = a_1a_2\cdots a_n$ be a finite local run of p. The *pattern* of ϱ_p is defined as one of the following expressions:

$* \ \emptyset$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^*\emptyset$
$* \ \{\ell_1, \ell_2\}$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^*\{\ell_1, \ell_2\}$
$* \ \emptyset\{\ell_i\}$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^*\emptyset\{\ell_i\}^*$
$* \ \{\ell_1, \ell_2\}\{\ell_i\}$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^*\{\ell_1, \ell_2\}\{\ell_i\}^*$

In the last cases we say that ϱ_p has a *strong pattern*, otherwise we say that it has a *weak pattern*.

We also define patterns of infinite runs, which summarize the local runs of processes that do not get blocked.

Definition 4.14 ▶ *Infinitary patterns*

Let p be a process, $\mathbf{L}_p = \{\ell_1, \ell_2\}$ its locks. Let $\varrho_p = a_1 a_2 \cdots a_n$ be a finite local run of p. The *pattern* of ϱ_p is defined as one of the following expressions:

$*\,\emptyset^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* \emptyset^\omega$
$*\,\emptyset\{\ell_i\}^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* \emptyset\{\ell_i\}^\omega$
$*\,\{\ell_1,\ell_2\}\{\ell_i\}^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* \{\ell_1,\ell_2\}\{\ell_i\}^\omega$
$*\,\{\ell_1,\ell_2\}^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* \{\ell_1,\ell_2\}^\omega$
$*\,(\emptyset\{\ell_i\})^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* (\emptyset^+\{\ell_i\}^+)^\omega$
$*\,(\{\ell_1,\ell_2\}\{\ell_i\})^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* (\{\ell_1,\ell_2\}^+\{\ell_i\}^+)^\omega$
$*\,(\{\ell_1,\ell_2\}\emptyset)^\omega$	if $tr(\varrho_p) \in ((2^{\mathbf{L}_p})^* \{\ell_1,\ell_2\}(2^{\mathbf{L}_p})^*\emptyset)^\omega$
$*\,(\{\ell_1,\ell_2\}\{\ell_1\}\{\ell_2\})^\omega$	if $tr(\varrho_p) \in (2^{\mathbf{L}_p})^* (\{\ell_1,\ell_2\}^*\{\ell_1\}^*\{\ell_1,\ell_2\}^*\{\ell_2\}^*)^\omega$

In the third case we say that ϱ_p has a *strong pattern*, otherwise we say it has a *weak pattern*. In the last case we say that ϱ_p is *switching*: after some time, p never releases both locks, but releases each one infinitely often.

Remark 4.3.1. *Note that for both finite and infinite local runs, the set* HOLDS(ϱ_p) *is determined by the pattern of* ϱ_p. *We will therefore use the notation* HOLDS(π_p) *with* π_p *a pattern to mean the set of locks held indefinitely by a run with pattern* π_p.

Given a finite or infinite local run ϱ_p, its pattern is written $\pi(\varrho_p)$. If ϱ_p is finite risky, we define its *blocking pattern* as the pair $(\pi(\varrho_p), \text{BLOCKS}(\varrho_p))$.

A *behaviour* is a set of blocking patterns and infinitary patterns. The behaviour of a process is the set of blocking patterns of its risky local runs and infinitary patterns of its infinite local runs. The behaviour of a 2LSS is the family $(\Pi_p)_{p\in\mathsf{Proc}}$ of behaviours of its processes.

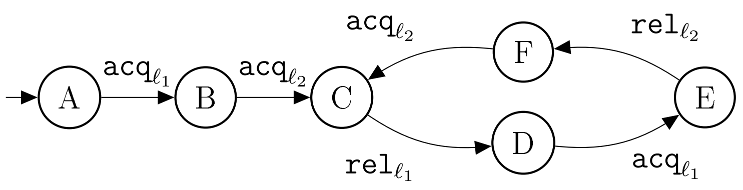


Figure 4.5: Example of a process of a 2LSS.

Example 4.3.3. *The process in Figure 4.5 has a single infinite run whose pattern is* $*(\{\ell_1,\ell_2\}\{\ell_1\}\{\ell_2\})^\omega$ *(it is switching). It also has finite runs of patterns* $*\emptyset$, $*\emptyset\{\ell_1\}$, $*\{\ell_1,\ell_2\}$, $\{\ell_1,\ell_2\}\{\ell_2\}$ *and* $*\{\ell_1,\ell_2\}\{\ell_1\}$, *but not all of them are risky. Its behaviour is* $\{*(\{\ell_1,\ell_2\}\{\ell_1\}\{\ell_2\})^\omega, (*\emptyset, \{\ell_1\}), (*\emptyset\{\ell_1\}, \{\ell_2\}), (\{\ell_1,\ell_2\}\{\ell_2\}, \{\ell_1\}), (*\{\ell_1,\ell_2\}\{\ell_1\}, \{\ell_2\})\}$.

Note that for each of the patterns defined above, the set of runs matching that pattern is a regular language.

Lemma 4.15

Let $p \in \mathsf{Proc}$ be a process and $\mathcal{A}_p$ its transition system. Let Π_p be a behaviour. We can define a deterministic parity automaton with $O(1)$ priorities and $O(|\mathcal{A}_p|)$ states over Σ_p recognizing local runs ϱ whose pattern is in Π_p[a].

[a] Again, we implicitly assume that finite runs are padded with a dummy letter to form infinite words, so that the desired set of runs can be expressed as a language of infinite words

Proof. We define a transition system which keeps track of the pattern of the finite run read so far. It is described in Figure 4.6 with $\mathbf{L}_p = \{\ell_1, \ell_2\}$. We label edges with operations instead of actions as in this automaton the transitions of an action a depend only on the sequence of operations applied. For finite runs, the last visited state indicates the pattern of the run. We can thus define a DFA for the desired pattern by taking a product of this structure with the transition system of p. This immediately gives a DFA for any blocking pattern, as the second component is given by the last state visited in the transition system of p.

For infinite runs, it suffices to observe that the set of states seen infinitely often while reading such a run determines its pattern.

As a result, the acceptance condition can be expressed as an Emerson-Lei condition. The acceptance condition associated to Π_p is the disjunction of the acceptance conditions associated to all of its patterns. This yields an Emerson-Lei automaton with bounded numbers of states and colours, which can be converted into a deterministic parity automaton with bounded states and priorities (by Proposition 2.3). The product of this automaton with the transition system of the process gives an automaton for the desired property. $\square$

Corollary 4.16

Given a 2LSS, we can compute the behaviour of each process in polynomial time.

We now present the key[2] proposition on locking patterns for 2LSS. It provides necessary and sufficient list of conditions for a set of local runs to be schedulable into a global run. Note that the criterion depends only on patterns of local runs and last states reached by finite local runs. This will be the crucial ingredient in the proof that the regular verification problem is in NP for 2LSS.

[2] Pun intended

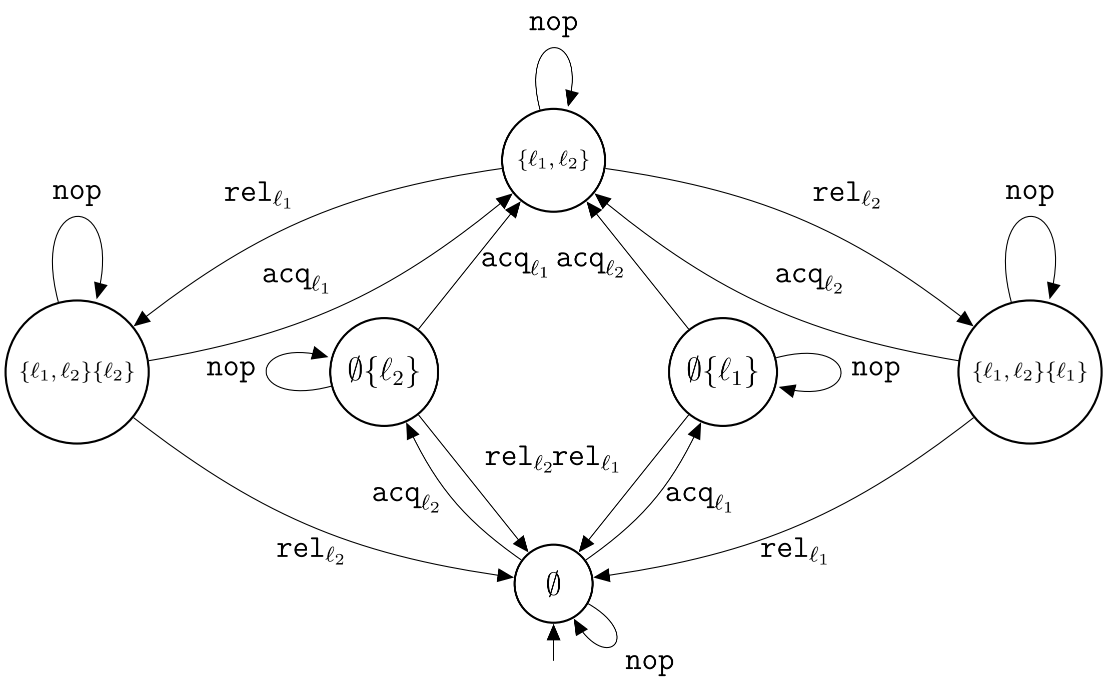


Figure 4.6: The automaton structure for pattern recognition.

Characterisation of schedulability using patterns

Proposition 4.17 ▶ Characterisation of schedulable local runs

Consider a family of finite or infinite local runs $(\varrho_p)_{p\in\mathsf{Proc}}$ of a sound 2LSS.
We write G_{Inf} for the undirected graph whose vertices are locks and with a p-labelled edge between ℓ_1 and ℓ_2 whenever p uses locks ℓ_1 and ℓ_2 and ϱ_p is switching.
The local runs $(\varrho_p)_{p\in\mathsf{Proc}}$ can be scheduled into a process-fair global run if and only if the following conditions are all satisfied.

1. Every finite ϱ_p is risky.
2. All sets $\textsc{Holds}(\varrho_p)$ are disjoint.
3. For all finite ϱ_p, $\textsc{Blocks}(\varrho_p)$ is included in $\bigcup_{q\in\mathsf{Proc}}\textsc{Holds}(\varrho_q)$.
4. The set $\left[\textsc{Holds}(\varrho_p)\cap\bigcup_{S\in\textsc{Inf}(\varrho_q)} S\right]$ is empty for all pairs of processes $p\neq q$ such that ϱ_q is infinite.
5. There is a total order $\leq$ on locks such that for all p whose run ϱ_p has a strong pattern $*\{\ell_1,\ell_2\}\{\ell_1\}$ (for finite runs) or $*\{\ell_1,\ell_2\}\{\ell_1\}^\omega$ (for infinite runs) we have $\ell_1\leq\ell_2$; where ℓ_2 is the other lock used by p.
6. There is no process p such that (1) $\{\ell_1,\ell_2\}\in\textsc{Inf}(\varrho_p)$ and (2) there is a path in G_{Inf} between ℓ_1 and ℓ_2 not using a p-labelled edge.

Proof. $\Rightarrow$ We start with the left-to-right implication. Let $(\varrho_p)_{p\in\mathsf{Proc}}$ be a family of local runs, suppose they can be scheduled into a process-fair global run ϱ.

- For all finite local runs ϱ_p, since ϱ is process-fair, there exists a point after which process p is permanently unable to execute any action. As a consequence, the state reached after executing ϱ_p cannot have any outgoing transition executing `rel` or `nop`, as those can always be executed (given that the 2LSS is sound). Furthermore, the locks of $\textsc{Blocks}_p$ are never released after some point; otherwise, process p would be able to execute the action of acquiring them infinitely often during the run, which would contradict the condition of process-fairness. This shows condition 1.

- All finite runs ϱ_p stop while holding the locks in $\textsc{Holds}(\varrho_p)$. All infinite ϱ_p eventually acquire and never release the locks of their $\textsc{Holds}(\varrho_p)$. Hence the $\textsc{Holds}(\varrho_p)$ sets need to be pairwise disjoint, proving condition 2.
- Furthermore, if a lock is not in some $\textsc{Holds}(\varrho_p)$ then it is free infinitely often, and thus cannot be in $\textsc{Blocks}(\varrho_p)$ for any p, as ϱ is process-fair. This proves condition 3.
- Let $p \neq q$ be two processes. All locks of $\bigcup \textsc{Inf}(\varrho_q)$ are held by q infinitely often, while locks of $\textsc{Holds}(\varrho_p)$ are held by p indefinitely after some point, hence those sets must be disjoint. This shows condition 4.
- If a run ϱ_p of a process p using locks ℓ_1, ℓ_2 has a strong pattern $*\{\ell_1, \ell_2\}\{\ell_1\}$ or $*\{\ell_1, \ell_2\}\{\ell_1\}^\omega$ then the last operation on ℓ_1 (when p acquires it for the last time) is followed by at least one operation on ℓ_2 in the run ϱ. We satisfy condition 5 by setting $\leq$ as an order on locks such that $\ell \leq \ell'$ whenever there is an operation on ℓ' after the last operation on ℓ in ϱ.
- We demonstrate condition 6 by contradiction. Say there exist such locks and processes, i.e., there exist $\ell = \ell_1, \dots, \ell_n = \ell'$ and $p_1, \dots, p_{n-1}$ without p such that for all $1 \leq i < n$, p_i has pattern $*(\{\ell_i, \ell_{i+1}\}\{\ell_i\}\{\ell_{i+1}\})^\omega$. Then all p_i are always holding a lock after some point, hence $\emptyset \notin \textsc{Inf}(\varrho_{p_i})$ for all i.

  Furthermore, as $\{\ell_n, \ell_1\} \in \textsc{Inf}(\varrho_p)$, this means that p holds ℓ_n and ℓ_1 simultaneously infinitely often. Whenever that happens, processes $p_1, \dots, p_{n-1}$ have to share the remaining $(n-2)$ locks, hence one of them holds no lock, contradicting the fact that $\emptyset \notin \textsc{Inf}(\varrho_{p_i})$ for all i.

$\Leftarrow$ For the other direction, suppose $(\varrho_p)_{p \in \mathsf{Proc}}$ satisfies all the conditions of the list. We construct a process-fair global run whose local projections are the ϱ_p.

To do so, we will construct a sequence of finite runs $v_0, v_1, \dots$ such that $v_0 v_1 \cdots$ is such a global run. We will ensure that the following property is satisfied for all $i \in \mathbb{N}$:

$$\begin{aligned}&\text{For all processes } p, \text{ after executing } v_0 \cdots v_i,\\ &\text{if } \textsc{Holds}(\varrho_p) \in \textsc{Inf}(\varrho_p) \text{ then } \textsc{Holds}((v_0 \cdots v_i)|_p) = \textsc{Holds}(\varrho_p)\\ &\text{otherwise } \varrho_p \text{ is switching and } p \text{ holds one lock.}\end{aligned} \tag{4.1}$$

We will also make sure that all p with an infinite ϱ_p execute an action in infinitely many v_i.

The first run v_0 has to be constructed separately as we require it to satisfy some extra conditions. We construct v_0 such that for all p:

- If ϱ_p is finite then $v_0|_p = \varrho_p$.

- If ϱ_p is infinite then $\varrho_p = v_0|_p u_p$ with u_p such that for every prefix u'_p of u_p, $\textsc{Holds}(v_0|_p u'_p) \in \textsc{Inf}(\varrho_p)$. Furthermore if $\emptyset \in \textsc{Inf}(\varrho_p)$ then $\textsc{Holds}(v_0|_p) = \emptyset$.

  In other words, we execute a prefix of each infinite run such that what follows matches its asymptotic behaviour.

Construction of v_0

- First, for all infinite ϱ_p such that $\emptyset \in \textsc{Inf}(\varrho_p)$, there exist arbitrarily large finite prefixes of ϱ_p ending with p having no lock. Hence we can select one of those prefixes $v_0|_p$, large enough for p to never hold a set of locks not in $\textsc{Inf}(\varrho_p)$ later in the run. We execute all such $v_0|_p$ at the start. All locks are free afterwards.

- We then execute for all other ϱ_p, their maximal prefix ending with p having no lock. All locks are still free.

- Then we consider all ϱ_p with strong patterns. For each one of them we have a lock ℓ_p such that $\textsc{Holds}(\varrho_p) = \{\ell_p\}$. We consider the total order $\leq$ given by condition 5. We take those runs one by one in increasing order of ℓ_p. We execute in full the finite ones, while for the infinite ones we execute a prefix $v_0|_p$ such that in the end p owns only ℓ_p and never acquires the other lock afterwards.

  Say we executed some of those local runs, let p be a process with a strong pattern accessing locks $\ell_1 \leq \ell_2$. By condition 2, all $\textsc{Holds}(\varrho_p)$ are disjoint, hence there is no other process q with $\textsc{Holds}(\varrho_q) = \{\ell_1\}$. The only locks that are not free at that point are the ℓ such that $\ell < \ell_1$ and $\textsc{Holds}(\varrho_q) = \{\ell\}$ for some q with a strong pattern. Therefore, both ℓ_1 and ℓ_2 are free, and the run can be executed. In the end the locks held indefinitely by processes with strong patterns are taken and all others are free.

- Then we consider the finite ϱ_p with weak patterns of the form $*\{\ell_1, \ell_2\}$ or $*\emptyset\{\ell_i\}$. For those, we can execute the rest of the run (we already executed the maximal prefix leading to them holding no lock), as all they do is take the locks in $\textsc{Holds}(\varrho_p)$, which are free by conditions 2 and 4.

- We then consider the infinite ϱ_p with non-empty $\textsc{Holds}(\varrho_p)$ and weak patterns of the form $*\emptyset\{\ell_i\}^\omega$ or $*\{\ell_1, \ell_2\}^\omega$. Clearly we can just execute the run until we reach a point after which p only ever holds locks of $\textsc{Holds}(\varrho_p)$ forever. We can do this as all locks taken so far are in $\textsc{Holds}(\varrho_q)$ for some q. Thus all locks from those $\textsc{Holds}(\varrho_p)$ are free by conditions 2.

- Then, we look at runs ϱ_p with patterns of the form $*(\{\ell_1, \ell_2\}\{\ell_i\})^\omega$. At that point all locks that are taken are in some $\textsc{Holds}(\varrho_q)$, thus by condition 4 both locks of p are free. Hence we can execute enough steps of ϱ_p to reach a point at which p holds $\textsc{Holds}(\varrho_p)$ and will only hold sets of locks of $\textsc{Inf}(\varrho_p)$ afterwards.

- Finally we consider the infinite switching runs ϱ_p. All those processes must have $\mathbf{L}_p \in \textsc{Inf}(\varrho_p)$, hence by condition 4 all their locks are free, given that all locks held before belong to some $\textsc{Holds}(\varrho_q)$. Note that condition 6 implies that G_{Inf} is acyclic: suppose there is a cycle, pick some edge between two locks ℓ_1, ℓ_2 along that cycle, let p be its label. Then p would witness a violation of condition 6.

  As G_{Inf} is acyclic, we can apply the following algorithm.

- We define a graph G initially equal to G_{Inf}. We first remove from G all isolated locks.
- While G has an edge, we pick a lock ℓ with exactly one neighbour in G. Let p be the label of the edge connecting it to its neighbour.

  We execute a prefix of ϱ_p such that p only owns ℓ at the end and p only owns sets of locks of $\text{Inf}(\varrho_p)$ afterwards. Then we remove the p-labelled edge and ℓ from G.

As we only remove edges from G, it stays acyclic, hence that lock ℓ always exists. As we remove all isolated locks from G at the start, the remaining locks are all used by a switching local run, and are thus free as all locks taken so far are in some $\text{Holds}(\varrho_q)$ and by condition 4. In the loop we maintain the fact that all locks still in G are free. As a consequence, the prefix of ϱ_p considered can always be executed.

In the end we have executed a suitable prefix of all ϱ_p. We have constructed a finite run v_0 whose projection $v_0|_p$ on $\mathcal{S}_p$ is such that:

- for all finite ϱ_p, $\varrho_p = v_0|_p$,
- for all infinite ϱ_p, $v_0|_p$ is a prefix of ϱ_p such that all local configurations seen later in the run are in $\text{Inf}(\varrho_p)$,
- v_0 satisfies property 4.1.

We now construct the remaining parts of the run. If all ϱ_p are finite then v_0 proves the lemma (we can set all other v_i as ε). Otherwise we must describe the rest of the process-fair global run whose projections are the ϱ_p. We start with a small construction that will help us define the v_i.

Suppose we constructed $v_0, \dots, v_i$ so that property 4.1 is satisfied for all $j \leq i$. Now suppose some lock ℓ_0 is not in any $\text{Holds}(\varrho_p)$ and is not free after executing $v_0 \cdots v_i$. Then there exists a switching ϱ_{p_1} with $\ell_0 \in \mathbf{L}_{p_1}$.

Let ℓ_1 be the other lock of p_1, say it is not free. By property 4.1, p_1 does not hold ℓ_1. By condition 4 ℓ_1 cannot be in some $\text{Holds}(\varrho_p)$, thus there exists a switching ϱ_{p_2} such that $\ell_1 \in \mathbf{L}_{p_2}$. Let ℓ_2 be the other lock of p_2.

We construct this way a sequence of processes $p_1, p_2, \dots$ and of locks $\ell_0, \ell_1, \dots$ such that $\mathbf{L}_{p_j} = \{\ell_{j-1}, \ell_j\}$ and ϱ_{p_j} is switching for all j. This sequence cannot be infinite as each p_j labels an edge in G_{Inf}, which is finite and acyclic.

Hence there exists k such that ℓ_k is free. We can therefore execute ϱ_{p_k} until p_k holds ℓ_k and not ℓ_{k-1}, then execute $\varrho_{p_{k-1}}$ until p_{k-1} holds ℓ_{k-1} and not ℓ_{k-2}, and so on until ℓ_1 is free.

Hence if a lock ℓ is not in any $\text{Holds}(\varrho_p)$ but is not free after executing $v_0 \cdots v_i$ then we can prolong the prefix run so that ℓ is free and some lock from the same connected component in G_{Inf} is not. For all such ℓ and i we name this prolongation of the run $\pi_{\ell,i}$.

Now that we have this construction, let us assume that we already defined $v_0, \dots, v_i$, and that property 4.1 is satisfied for all $j \leq i$. We construct v_{i+1}. Let p be either a process that never executed an action, or if there are no such processes, the process whose last action in $v_0 \cdots v_i$ is the earliest.

We extend the current run to include the execution of some actions from ϱ_p. If the next action in ϱ_p is a `nop` operation, it can be executed immediately. The next action cannot execute a `rel` operation: after executing v_0, all processes with infinite ϱ_p only hold sets of locks from $\textsc{Inf}(\varrho_p)$. As a consequence, by property 4.1, if ϱ_p is switching then after executing $v_0 \cdots v_i$ the process p holds one lock and will not release it as it would be left with no lock and $\emptyset \notin \textsc{Inf}(\varrho_p)$. On the other hand, if p is not switching then after executing $v_0 \cdots v_i$ it holds $\textsc{Holds}(\varrho_p)$ and cannot release any lock as it keeps those forever.

We are left with the case where the next action of p acquires a lock ℓ. If ℓ is not free we apply $\pi_{\ell,i}$ to free it (and block another lock of the same connected component of G_{Inf}). Note that after executing $\pi_{\ell,i}$ all processes with switching runs still hold one lock, and the others have not moved.

- If ϱ_p is switching then p was already holding a lock ℓ', and it can then take ℓ and then run ϱ_p until it holds only one lock again, thus respecting property 4.1.
- Otherwise p was holding $\textsc{Holds}(\varrho_p)$ (by Property 4.1) and we have to run ϱ_p to let him take ℓ and then continue until p holds exactly $\textsc{Holds}(\varrho_p)$ again.
  - If we can do it right away we do so.
  - Otherwise it means that p needs its other lock ℓ' to reach that next step, and that this lock is taken. More precisely, it means that $\textsc{Holds}(\varrho_p) = \emptyset$ and the pattern of ϱ_p is $*(\{\ell, \ell'\}\emptyset)^\omega$.

    As $\{\ell, \ell'\} \in \textsc{Inf}(\varrho_p)$, by condition 6, ℓ and ℓ' are not in the same connected component of G_{Inf}. Hence we can execute $\pi_{\ell',i}$, without locking ℓ back, as $\pi_{\ell,i}$ and $\pi_{\ell',i}$ use disjoint sets of locks and processes.

    This ensures that both ℓ and ℓ' are free, which allows p to take ℓ and proceed to the next point at which it holds $\emptyset$.

  In both cases we end up in a configuration where p owns $\textsc{Holds}_p$, all processes with switching runs hold exactly one lock, and the other processes did not move, thus respecting property 4.1.

We have constructed v_{i+1}, ensuring that property 4.1 is satisfied for $i+1$. Furthermore $v_{i+1}|_p$ is non-empty for p a process with infinite ϱ_p which either never executed anything before or executed its last action the earliest. This ensures that all p with infinite ϱ_p execute infinitely many actions in $v_0 v_1 \cdots$. Hence we obtain a global run $v = v_0 v_1 \cdots$ such that for all p we have $v|_p = \varrho_p$.

Furthermore we ensured that v is process-fair as all p with finite runs are blocked: all such ϱ_p lead to a state from which only locks of $\textsc{Blocks}_p$ can be taken, by condition 1, and by condition 3 all $\textsc{Blocks}_p$ are included in $\bigcup_{p \in \mathsf{Proc}} \textsc{Holds}(\varrho_p)$, the set of locks that are never free from some point on.

As a result, there exists a process-fair run whose local projections are the ϱ_p, proving the right-to-left implication. $\square$

An NP upper bound for 2LSS

Then we prove that the complexity falls to NP when we demand that each process uses at most two locks.

Theorem 4.18

The regular verification problem is NP-complete for 2LSS. The lower bound already holds for locally live exclusive 2LSS.

Proof. We start with the upper bound. Let $\mathcal{S} = ((\mathcal{A}_p)_{p \in \mathsf{Proc}}, \mathbf{L}, op)$ be a 2LSS and $((\mathcal{B}_p)_{p \in \mathsf{Proc}}, \varphi)$ a regular objective. Our NP algorithm goes as follows: we guess a pattern π_p for each process p, as well as a valuation ν of the $(\inf_{p,q})_{p \in \mathsf{Proc}, q \in Q_p}$. For each p let $\textsc{Holds}_p$ be the set of locks kept indefinitely by a run respecting π_p. Then we check that those patterns satisfy the conditions of Proposition 4.17 and that this valuation satisfies the formula φ.

For each p we construct the product $\mathcal{C}$ of $\mathcal{A}_p$, $\mathcal{B}_p$ and $\mathcal{A}_{\pi_p}$ (from Lemma 4.15) to obtain a DELA recognising runs of p that match pattern π_p and are in the language of $\mathcal{B}_p$. We guess an ultimately periodic run of the form uv^ω with u and v of polynomial size in the number of states of $\mathcal{C}$ and check that it is accepting (otherwise we stop). It is well-known that a DELA either has an empty language or accepts a run of that form. Then we accept.

We accept if and only if there is a valuation ν satisfying φ and a family of patterns $(\pi_p)_{p \in \mathsf{Proc}}$ such that there exist local runs $(\varrho_p)_{p \in \mathsf{Proc}}$ of the processes matching those patterns and producing words whose runs in the $(\mathcal{B}_p)_{p \in \mathsf{Proc}}$ match ν, and such that the finite ones end in states from which they can only take locks of $\textsc{Holds}_p$. By Proposition 4.17, this is true if and only if there is a global run of the system satisfying the given objective. Hence the problem is in NP.

The lower bound is immediate: For a trivial 2LSS with a single process p with a single state and just a self-loop labelled a with $\mathtt{op}(a) = \mathtt{nop}$, the problem becomes the emptiness of the Emerson-Lei automaton $(\mathcal{B}_p, \varphi)$. The NP-hardness follows from Proposition 2.6. □

We will see that the problem is NP-complete already in some restricted subcases in Theorems 4.19 and 4.29.

4.4 Process deadlocks

Here we are interested in the process deadlock problem. We provide a polynomial-time algorithm based on a key lemma that lists the different ways a process can be blocked, in the case of exclusive 2LSS.

By contrast, the process deadlock problem becomes NP-complete when we remove the exclusive requirement.

Theorem 4.19

The process deadlock problem is NP-complete for 2LSS. The lower bound holds already for locally live 2LSS.

Proof. The upper bound follows from the one on regular objectives given by Theorem 4.18. The lower bound (which already holds for locally live systems) will be shown later, see Corollary 4.63. □

4.4.1 A polynomial-time algorithm for process deadlocks in exclusive 2LSS

This section is dedicated to the proof of the following proposition:

Proposition 4.20

The process deadlock problem is in PTIME for sound exclusive 2LSS.

We will start with a small technical lemma stating that, given some suitable local runs, we can construct a process-fair run in which we execute those local runs in full or up to a deadlock. After this, we will define a graph G with locks as vertices and whose edges describe the possible patterns of each process. Then, we will establish some sufficient conditions for a process deadlock to exist in Lemmas 4.24, 4.25, 4.26. In Lemma 4.27 we will group those conditions to obtain a characterisation of process deadlocks on the graph G. We will conclude by arguing that we can compute G and check the characterisation in polynomial time

Throughout this section we fix a sound exclusive 2LSS $\mathcal{S}$.

Lemma 4.21

Let $(s_p, B_p)_{p \in \mathsf{Proc}}$ be a global configuration and for each process p let u_p be a local run starting in (s_p, B_p) and such that u_p is either infinite or leads to a state with no outgoing transitions.
There exists a process-fair global run ϱ from $(s_p, B_p)_{p \in \mathsf{Proc}}$ such that for all p its projection ϱ_p on $\mathcal{S}_p$ is a prefix of u_p.

Proof. We construct ϱ by iterating the following step: For each p we set $u_p = v_p w_p$ with v_p the prefix of u_p executed so far. We select uniformly at random a process $p \in \mathsf{Proc}$. If it can execute the first action of w_p then we let it do so, otherwise we do nothing.

We iterate this procedure indefinitely. This produces a (possibly finite) global run of the system such that its local projections are prefixes of the u_p. We prove that it is process-fair.

Let $p \in \mathsf{Proc}$, assume that p has an available action at infinitely many steps. As our LSS is exclusive, whenever p has an available action and is in some state s, either all outgoing transitions are executing an operation `nop` or `rel` (and thus can all be executed as the system is sound), or they all acquire the same lock ℓ (as the system is exclusive). Hence if one outgoing transition can be executed , they all can. In particular the next action of u_p is available. As a result, p can execute the next action of u_p at infinitely many steps, and thus will progress infinitely many times in u_p with probability 1.

In conclusion, by following this procedure, with probability 1 we either reach a global deadlock, or we always have an available action. In the second case, at least one process will be able to progress infinitely many times and thus the resulting run ϱ is infinite. Hence, all processes that can execute an action at infinitely many steps of the run will do so, proving that the run is process-fair. As we obtain a process-fair run satisfying the requirement of the lemma with probability 1, in particular such a run must exist. □

Definition 4.22

Define the graph G whose vertices are locks and with an edge $\ell_1 \xrightarrow{p} \ell_2$ if and only if the process p has a local run ϱ_p ending in a state where all outgoing transitions acquire ℓ_2 and such that $\textsc{Holds}(\varrho_p) = \{\ell_1\}$. We say that ϱ_p *witnesses* the edge $\ell_1 \xrightarrow{p} \ell_2$.

Before we start showing necessary conditions on G for the existence of a process deadlock, we make an observation on its structure.

Lemma 4.23

For all $p \in \mathsf{Proc}$, if there is a p-labelled edge $\ell_1 \xrightarrow{p} \ell_2$ in G then either $\ell_1 \xrightarrow{p} \ell_2$ is witnessed by a run with a weak pattern or its reverse $\ell_2 \xrightarrow{p} \ell_1$ is in G and is witnessed by a run with a weak pattern.

Proof. As p has an edge $\ell_1 \xrightarrow{p} \ell_2$ in G, there is a local run which has both locks of p at the same time. Let ϱ_p be such a run of minimal length. The last operation in ϱ_p must be a `acq`, by minimality. Let ϱ'_p be the local run obtained by removing the last step in ϱ_p. Suppose the last operation in ϱ'_p besides `nop` is a `rel`, then there is a previous configuration in ϱ_p in which p holds both of its locks, contradicting the minimality of ϱ_p. Hence ϱ'_p has a weak pattern, and it leads to a state where p may acquire ℓ, thus has to acquire ℓ as the system is exclusive. Furthermore p is then holding its other lock, therefore ϱ'_p witnesses an edge in G. □

We can now show the three main necessary conditions.

Lemma 4.24 ▶ First condition

If p has a reachable transition acquiring some lock ℓ and there is a path from ℓ to a cycle in G then there is a process-fair global run with a finite projection on p.

Proof. Let $\ell = \ell_0 \xrightarrow{p_1} \ell_1 \xrightarrow{p_2} \cdots \xrightarrow{p_k} \ell_k$ be such a path in G and let $\ell_k = \ell'_1 \xrightarrow{p'_1} \cdots \ell'_n \xrightarrow{p'_n} \ell'_{n+1} = \ell'_1 = \ell_k$ be such a cycle.

For all $1 \leq i \leq k$ we choose a run ϱ_i witnessing $\ell_{i-1} \xrightarrow{p_i} \ell_i$. Similarly for all $1 \leq j \leq n$ we choose a run ϱ'_j witnessing $\ell'_j \xrightarrow{p'_i} \ell'_{j+1}$, and we choose it so that it has a weak pattern whenever possible.

If there exists j such that ϱ'_j has a weak pattern, then we proceed as follows: Let $\varrho_j = \varrho_j^- \varrho_j^+$ so that ϱ_j^- is the maximal neutral prefix of ϱ'_j. We execute ϱ_j^-. Let m be the maximal index such that $\ell_m \in \{\ell'_1, \cdots, \ell'_n\}$. We execute all ϱ_i in increasing order for $1 \leq i \leq m$.

Then we execute $\varrho'_{j+1} \cdots \varrho'_n \varrho'_1 \cdots \varrho'_{j-1}$ and then ϱ''_j. Then we end up in a configuration where all p_i with $i \leq m$ are holding ℓ_{i-1} and need ℓ_i to advance, while all p'_i are holding ℓ'_i and need ℓ'_{i+1} to advance. As $\ell_j \in \{\ell'_1, \ldots, \ell'_n\}$, all those processes are blocked, and in particular $\ell = \ell_0$ is held by a process which will never release it.

As p has a reachable transition taking ℓ, we can define ϱ_p as a run that ends in a state where some outgoing transitions takes a lock of $\{\ell_0, \ldots, \ell_m, \ell'_1, \ldots, \ell'_n\}$. By minimality this run can be executed, as all other locks are free. By exclusiveness, it reaches a state where all transitions take the same non-free lock.

By Lemma 4.21 we can extend this run into a process-fair one, whose projection on p can only be ϱ_p, as p will never be able to advance further.

Now suppose there is no j such that ϱ'_j has a weak pattern, then as we took all ϱ'_j with weak patterns whenever possible, it means there is no local run with a weak pattern witnessing any of the $\ell'_j \xrightarrow{p'_j} \ell'_{j+1}$. We can then apply Lemma 4.23 to show that the reverse cycle $\ell'_1 = \ell'_{n+1} \xrightarrow{p'_n} \ell'_n \cdots \xrightarrow{p'_1} \ell'_1$ exists in G and all its edges are witnessed by runs with weak patterns. Hence we can apply the arguments from the previous case using this cycle to conclude. □

Lemma 4.25 ▶ Second condition

If p has a reachable transition acquiring some lock ℓ and

- there is a path in G from ℓ to some ℓ' and
- there is a process q with an infinite local run ϱ_q acquiring ℓ' and never releasing it,

then there is a process-fair global run with a finite projection on p.

Proof. Let $\ell = \ell_0 \xrightarrow{p_1} \ell_1 \xrightarrow{p_2} \cdots \xrightarrow{p_k} \ell_k = \ell'$ be the shortest path from ℓ to ℓ'. Let ϱ_p be a local run of p acquiring ℓ at some point, either infinite or leading to a state with no outgoing transition. For each $1 \leq i \leq k$ we select a local run ϱ_i of p_i witnessing $\ell_{i-1} \xrightarrow{p_i} \ell_i$. Furthermore we select those ϱ_i with weak patterns whenever possible. Let ℓ_q be the other lock used by q besides ℓ', and let ϱ_q be an infinite run of q in which ℓ' is eventually taken and never released. We can decompose ϱ_q as $\varrho'_q\varrho''_q$ where ϱ'_q is the largest prefix of ϱ_q such that q holds no lock at the end. We distinguish several cases:

Case 1: $\ell_q \notin \{\ell_0, \dots, \ell_k\}$, or ℓ_q is not used in ϱ''_q. Then we can execute ϱ'_q, leaving all locks free, then $\varrho_1 \cdots \varrho_k$, which can be done as the execution of $\varrho_1 \cdots \varrho_i$ leaves $\ell_i, \dots, \ell_k$ free and thus ϱ_{i+1} can be executed. Then as no ℓ_i is used in ϱ''_q, we can execute a prefix of ϱ''_q large enough so that ℓ' is held by q and never released later. Let ϱ be the run constructed so far. Then by Lemma 4.21 we can construct a process-fair run ϱ' starting in the last configuration of ϱ whose projection on q is a prefix of ϱ''_q (thus ℓ_k is never released and thus neither are $\ell_0, \dots, \ell_{k-1}$) and whose projection on p is a prefix of ϱ_p (and thus finite as ϱ_p tries to acquire ℓ, which is never free). As a consequence, $\varrho\varrho'$ is a process-fair run whose projection on p is finite.

Case 2: $\ell_q = \ell_j$ for some $0 \leq j \leq k$ and ϱ_q acquires ℓ_j at some point and never releases it. Then we apply the same reasoning as in the previous case for the path $\ell = \ell_0 \xrightarrow{p_1} \cdots \xrightarrow{p_j} \ell_j$.

Case 3: $\ell_q = \ell_j$ for some $0 \leq j \leq k$ and ℓ_j is used in ϱ''_q but not kept indefinitely.

Subcase 3.1: there is an edge $\ell' \xrightarrow{q} \ell_j$. Then we have a path from ℓ to a cycle $\ell_j \xrightarrow{p_{j+1}} \cdots \xrightarrow{p_k} \ell' \xrightarrow{q} \ell_j$. Hence by Lemma 4.24, there is a process-fair global run with a finite projection on p.

Subcase 3.2: One of the runs ϱ_i has a weak pattern $*\emptyset\ell_i$. We decompose ϱ_i as $\varrho_i^- \varrho_i'$ with ϱ_i^- its largest neutral prefix. We execute ϱ_i^-, and then $\varrho_{i+1} \cdots \varrho_k$. After that we execute a prefix ϱ_q' of ϱ_q such that at the end q holds only ℓ', and does not release it later. This prefix exists as q never keeps ℓ_j indefinitely in ϱ_q. We decompose ϱ_q as $\varrho_q'\varrho_q''$. Then we execute $\varrho_1 \cdots \varrho_{i-1}\varrho_i'$. All those runs can be executed as before executing each $\varrho_{i'}$ both locks of $p_{i'}$ are free, and before executing ϱ_i', ℓ_{i-1} is free, which is all that is needed to execute ϱ_i' as ϱ_i has a weak pattern. Let ϱ be the run constructed so far. Then by Lemma 4.21 we can construct a process-fair run from the configuration reached by ϱ whose projection on q is a prefix of ϱ_q'' and whose projection on p is a prefix of ϱ_p. As a consequence, $\ell' = \ell_k$ is never released in ϱ_q'' and thus neither are $\ell_0, \ldots, \ell_{k-1}$. As ϱ_p tries to take $\ell = \ell_0$ at some point, its prefix executed in ϱ' is finite. Hence $\varrho\varrho'$ is a process-fair run with a finite projection on p.

Subcase 3.3: There is no edge $\ell' \xrightarrow{q} \ell_j$ and all ϱ_i have strong patterns. We once again decompose ϱ_q as $\varrho_q'\varrho_q''$, with ϱ_q' such that at the end q holds only ℓ', and does not release it later. When executing the ϱ_q'' part of ϱ_q, q holds a lock at all times, and holds ℓ_j at some point and ℓ' at some point, hence it has to have both at the same time at some moment. Hence there is a moment at which q holds one of the locks and is about to get the other. As the system is exclusive, it means all its available transitions take that lock. Hence there is an edge $\ell' \xrightarrow{q} \ell_j$ or $\ell_j \xrightarrow{q} \ell'$ in the graph. As we assumed that there is no edge $\ell' \xrightarrow{q} \ell_j$, there is one $\ell_j \xrightarrow{q} \ell'$. Furthermore, as we selected the ϱ_i so that they had weak patterns whenever possible, it means that for all i there is no run with a weak pattern witnessing $\ell_{i-1} \xrightarrow{p_i} \ell_i$. By Lemma 4.23 this means that there are edges $\ell_k \xrightarrow{p_k} \ell_{k-1} \xrightarrow{p_{k-1}} \cdots \xrightarrow{p_{j+1}} \ell_j$. With the edge $\ell_j \xrightarrow{q} \ell'$, we obtain a cycle in G with a path from ℓ to it. By Lemma 4.24, there is a process-fair global run with a finite projection on p.

This concludes our case distinction, proving the lemma. □

Lemma 4.26 ▶ Third condition

If p has a reachable transition acquiring some lock ℓ and

- there is a path in G from ℓ to some ℓ' and
- there is a process q with a local run ϱ_q with $\ell' \in \textsc{Holds}(\varrho_q)$ and going to a state with no outgoing transitions,

then there is a process-fair global run with a finite projection on p.

Proof. Let s_q be the state reached by ϱ_q, we add a self-loop on it with a fresh letter $\#$. Then $\varrho_q\#^\omega$ is an infinite run acquiring ℓ' and never releasing it.

Hence by Lemma 4.25, there is a process-fair run ϱ in this new system whose projection on p is finite. Let h be the morphism such that $h(\#) = \varepsilon$ and $h(a) = a$ for all other letters a. Then $h(\varrho)$ is a process-fair run of the original system: it is a run as $\#$ does not change the configuration, meaning that all actions of $h(\varrho)$ can be executed. For the same reason, if a process p' other than q only has finitely many actions in $h(\varrho)$, then the same is true in ϱ, thus there is a point after which no configuration allows p' to move in ϱ, and thus in $h(\varrho)$ as well. As for q, either it only executes $\#$ from some point on, meaning it has reached s_q and will be immobilised in $h(\varrho)$, or it never executes any $\#$, in which case $h(\varrho) = \varrho$ and it follows the same configurations in both. □

We gather the conditions listed above to characterise process deadlocks in exclusive 2LSS.

Lemma 4.27 ▶ Characterisation of process deadlocks

There is a process-fair run whose projection on p is finite if and only if there is a local run ϱ_p of p leading to a state where all outgoing transitions take some lock ℓ and either

1. p has a local run leading to a state with no outgoing transitions.
2. or there is a path from ℓ to a cycle in G
3. or there is a path in G from ℓ to some lock ℓ' and there is a process q with a local run ϱ_q with an infinitary pattern with $\ell' \in J$ for all J in $\textsc{Inf}(\varrho_q)$.
4. or there is a path in G from ℓ to some lock ℓ' and there is a process q with a local run ϱ_q such that $\ell' \in \textsc{Holds}(\varrho_q)$ and leading to a state with no outgoing transitions.

Proof. We start with the left-to-right implication: Say there is a run ϱ whose projection on p is finite. For each process $p' \in \mathsf{Proc}$ let $\varrho_{p'}$ be its local run.

Then ϱ_p has to end in a state where all available transitions acquire a lock ℓ. If there are no transitions at all, condition 1 is satisfied. If there is at least one such transition, then ℓ is held forever by some other process p_1.

We construct a path $\ell = \ell_0 \xrightarrow{p_0} \ell_1 \xrightarrow{p_1} \cdots$ in G so that all ℓ_i are held indefinitely by some process after some point in the run. Say we already constructed those up to i.

There is a process p_i holding ℓ_i indefinitely. If ϱ_{p_i} is infinite, then condition 3 is satisfied. Otherwise, ϱ_{p_i} is finite, and with a finitary pattern such that $\ell_i \in \textsc{Holds}(\varrho_{p_i})$.

If this local run ends up in a state with no outgoing transition then condition 2 is satisfied, otherwise it must have no choice but to acquire some lock ℓ'_{i+1}. Hence we construct an infinite path $\ell'_0 \xrightarrow{p'_0} \ell'_1 \xrightarrow{p'_1} \cdots$ in G.

The set of processes is finite, hence there exist $i < j$ such that $\ell_i = \ell_j$, meaning we have reached a cycle. Thus condition 4 is satisfied.

For the other direction, suppose there exists ℓ as in the statement of the lemma, so that one of the conditions is satisfied.

If condition 1 is satisfied, then we have a finite run ϱ_p leading to a state with no outgoing transition. We execute it and then prolong it into a global process-fair run by choosing a process uniformly at random and executing one of its available actions if there is any (similarly to the proof of 4.21). We obtain a process-fair run in which p only has finitely many actions. If condition 2 is satisfied then we have the result by Lemma 4.24. If condition 3 is satisfied then we have the result by Lemma 4.26. If condition 4 is satisfied then we have the result by Lemma 4.25. □

To conclude, by Lemma 4.27, we only have to check the conditions listed in its statement.

We start by looking, in the transition system of process p, for a reachable local state with no outgoing transition. If there is one, we accept. Otherwise, we compute the behaviour of each process (which is computable in polynomial time by Lemma 4.16), and compute G from those behaviours. The conditions listed in the lemma above are then straightforward to check in polynomial time, proving Proposition 4.20.

4.5 Global deadlocks

We describe how to solve the global deadlock problem for 2LSS. As a first step, we characterise global deadlocks for 2LSS in terms of patterns. This characterisation is simpler than previous ones as we only have to deal with finite local runs. .1 In this section we will show that the global deadlock problem is NP-complete on 2LSS, but falls in PTIMEover locally live 2LSS, a mild assumption that prevents processes from blocking by themselves.

The next lemma gives a characterization of systems with reachable global deadlocks in terms of patterns.

Lemma 4.28

Let $\mathcal{S} = (\mathsf{Proc}, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathbf{L})$ be a 2LSS and $\Pi = (\Pi_p)_{p\in\mathsf{Proc}}$ its behaviour. There is a run leading to a global deadlock if and only if there exists a family of blocking patterns $(\pi_p, B_p) \in \Pi_p$ for each process p, such that all conditions below hold.

- for all p, $\bigcup_{p\in\mathsf{Proc}} B_p \subseteq \bigcup_{p\in\mathsf{Proc}} \textsc{Holds}(\pi_p)$,
- the sets $\textsc{Holds}(\pi_p)$ are pairwise disjoint,
- there exists a total order $<$ on $\mathbf{L}$ such that for all p, if π_p is a strong pattern $*\{\ell_1, \ell_2\}\{\ell_1\}$ then $\ell_1 < \ell_2$.

Proof. For each p, let ϱ_p be a risky local run such that $(\pi_p, B_p) = (\pi(\varrho_p), \textsc{Blocks}(\varrho_p))$. We simply apply Proposition 4.17 □

4.5.1 Global deadlocks for general 2LSS

In this section we fix a 2LSS $\mathcal{S} = (\mathsf{Proc}, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathbf{L})$ over the set of processes Proc. We assume that the 2LSS is sound. This is not a restriction on our results as the number of locks per process is bounded: we can make the system sound with a constant blow-up in the number of states.

Thanks to Lemma 4.28, in order to decide if there is a global deadlock for a given system it is enough to compute the behaviour Π_p of each process p and show that the sets of patterns Π_p meet the conditions given by Lemma 4.28.

Theorem 4.29

The global deadlock problem for 2LSS is NP-complete. The lower bound holds even for exclusive 2LSS.

Proof. The upper bound follows from Theorem 4.18. The lower bound (which holds already for exclusive systems) will come as a by-product of a later proof, as stated in Corollary 4.65. □

4.5.2 Global deadlocks for locally live systems in polynomial time

We now consider the case of locally live 2LSS. Recall that with this condition, a process can only block if all its available transitions need to acquire a lock, but all these locks are taken. In the last subsection we showed that we could check in NP if the conditions from Lemma 4.28 hold. Here we show that this check can be done in PTIME in the locally live case.

Theorem 4.30

The global deadlock problem for locally live 2LSS is in PTIME.

The argument is unfortunately quite lengthy. We represent a behaviour as a lock graph G_Π, with vertices corresponding to locks and edges to patterns. Then, thanks to local liveness, instead of Lemma 4.28 we get Lemma 4.35 characterizing when a strategy is not winning by the existence of a subgraph of G_Π, called *sufficient deadlock scheme.* The main part of the proof is a polynomial time algorithm for deciding the existence of sufficient deadlock schemes.

Remark 4.5.1. *Let ϱ be a run leading to a global deadlock. In particular, at the end of this run every process is in a state where all available actions take a lock. Furthermore, as we are in a locally live framework, they must all have at least one available action. In consequence, we have* $\text{BLOCKS}(\varrho_p) \neq \emptyset$ *for all p.*

In light of this remark, we have two types of local runs which can result be the projection of a run leading to a global deadlock: Those that hold no lock at the end and those which hold a lock ℓ_1 and are trying to acquire the other one at the end.

We represent those of the second type as a graph. An edge labelled by p from ℓ_1 to ℓ_2 means that p has a local run in which it tries to take ℓ_2 while holding ℓ_1.

Definition 4.31 ▶ Lock graph G_Π

For a behaviour $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$, we define a labelled graph $G_\Pi = (\mathbf{L}, E_\Pi)$, called *lock graph*, whose nodes are locks and whose edges are either weak or strong. Edges are labelled by processes.

There is a *weak edge* $\ell_1 \overset{p}{\dashrightarrow} \ell_2$ in G_Π whenever there is a blocking pattern $(*\emptyset\{\ell_1\}, \{\ell_2\})$ in Π_p. There is a *strong edge* $\ell_1 \overset{p}{\Rightarrow} \ell_2$ whenever there is a strong pattern $(*\{\ell_1, \ell_2\}\{\ell_1\}, \{\ell_2\})$ in Π_p **and** there is no blocking pattern $(*\emptyset\{\ell_1\}, \{\ell_2\})$ in Π_p. We write $\ell_1 \xrightarrow{p} \ell_2$ when the type of the edge is irrelevant.

A path (resp. cycle) in G_Π is *simple* if all its edges are labelled by different processes. A cycle is *weak* if it contains some weak edge, and *strong* otherwise.

Note that the information contained in the graph does not say anything about runs holding no lock at the end. The next definition provides some notions to incorporate them next to our graph representation.

Definition 4.32

For a behaviour $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$, a process p is called *solid* if there is no blocking pattern of the form $(*\emptyset, B)$ in Π_p; otherwise it is called *fragile*.
A process p is Z-*lockable* in Π if there is a blocking pattern $(*\emptyset, B)$ in Π_p with $B \subseteq Z$. Note that a process is fragile if and only if it is Z-lockable for some Z.
A *solid edge* of G_Π is one that is labelled by a solid process. A *solid cycle* is one that only has solid edges.

What the previous definition says is that a solid process needs to take a lock to be blocked, whereas a fragile one can be blocked without owning a lock. So solid processes must be taken into account in the deadlock schemes defined next:

Definition 4.33 ▶ Z-deadlock scheme

Consider a behaviour $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$, and the associated lock graph G_Π. Let $Z \subseteq \mathbf{L}$ be a set of locks. We set $Proc_Z$ as the set of processes whose accessible locks are both in Z.
A *Z-deadlock scheme* for Π is a partial function $\mathtt{ds}_Z : Proc_Z \to E_\Pi \cup \{\perp\}$ assigning an edge of G_Π to some processes of $Proc_Z$ such that:

C1 For all $p \in Proc_Z$, if $\mathtt{ds}_Z(p) \neq \perp$ then $\mathtt{ds}_Z(p)$ is a p-labelled edge of G_Π.

C2 If $p \in Proc_Z$ is solid then $\mathtt{ds}_Z(p) \neq \perp$.

C3 For all $\ell \in Z$ there is a unique $p \in Proc_Z$ such that $\mathtt{ds}_Z(p)$ is an outgoing edge of ℓ.

C4 The subgraph of G_Π restricted to $\mathtt{ds}_Z(Proc_Z)$ does not contain any strong cycle.

The idea of the previous definition is that a Z-deadlock scheme witnesses a way to reach a configuration in which all locks of Z are taken, and all processes using those locks are blocked. Each process from $Proc_Z$ is mapped to an edge telling which lock it holds in the global deadlock configuration and which one it needs in order to advance. A process which is not holding any lock is mapped to $\perp$. For every lock in Z there is a unique outgoing edge in $\mathtt{ds}_Z$, corresponding to the process owning that lock. Note that this implies that the subgraph induced by $\mathtt{ds}_Z$ is a union of cycles, with some trees rooted in these cycles, and all edges towards the roots in those trees.

Definition 4.34 ▶ Sufficient deadlock scheme

A *sufficient deadlock scheme* for a behaviour Π is a Z-deadlock scheme ds_Z for Π such that for every process $p \in \mathsf{Proc}$ either $ds_Z(p)$ is an edge of the lock graph G_Π, or p is Z-lockable in Π.

We now prove a key result: a 2LSS admits a global deadlock if and only if the lock graph of its behaviour admits a sufficient deadlock scheme. We will then show that those conditions can be checked in polynomial time.

Lemma 4.35

Consider a locally live 2LSS and $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$ its behaviour. There is a run leading to a global deadlock if and only if there is a sufficient deadlock scheme for Π.

Proof. Suppose there is a run ϱ leading to a global deadlock. Then there exist blocking patterns $(\pi_p, B_p) \in \Pi_p$ for each p such that the three conditions of Lemma 4.28 are met. Let $Z = \bigcup_{p \in \mathsf{Proc}} \textsc{Holds}(\pi_p)$, and for all $p \in \mathit{Proc}_Z$, define $ds(p)$ as:

- $\perp$ if $\textsc{Holds}(\pi_p) = \emptyset$,

- $\ell_1 \xrightarrow{p} \ell_2$ if $\textsc{Holds}(\pi_p) = \{\ell_1\}$ $B_p = \{\ell_2\}$ (whether π_p is strong or weak is irrelevant for now). This edge exists by definition of G_Π.

Once again note that there are no other possible cases above, as the system is locally live and thus B_p cannot be empty.

We show that ds is a sufficient deadlock scheme for Π by checking the four conditions from Definition 4.33.

- The first condition C1 holds by definition of ds.

- For the second condition C2 let $p \in \mathit{Proc}_Z$ and suppose p is solid with respect to Π. Then, $\textsc{Holds}(\pi_p)$ is not empty and $ds(p) \neq \perp$.

- For the third condition C3 let $\ell \in Z$. As Z is the disjoint union of the sets $\textsc{Holds}(\pi_p)$ there exists a unique $p \in \mathit{Proc}_Z$ such that $\ell \in \textsc{Holds}_p$, so a unique edge $ds(p)$ outgoing from ℓ.

- For the last condition C4 note that for all strong edges $\ell \overset{p}{\Rightarrow} \ell'$, π_p must be a strong pattern, hence $\ell \leq \ell'$. As $\leq$ is a total order on locks, there cannot be any strong cycle.

It remains to show that ds is a sufficient deadlock scheme for Π. Suppose that $p \notin \mathit{Proc}_Z$ or $ds(p) = \perp$. In both cases $\textsc{Holds}(\pi_p) = \emptyset$, thus p is B_p-lockable, and hence Z-lockable as $B_p \subseteq Z$. As a consequence, ds is a sufficient deadlock scheme for Π.

For the other direction, suppose we have a sufficient Z-deadlock scheme ds for Π. As $ds(\mathit{Proc}_Z)$ does not contain any strong cycle (by C4), we can pick a total order $\leq$ on locks such that for all strong edges $\ell_1 \overset{p}{\Rightarrow} \ell_2$ belonging to $ds(\mathit{Proc}_Z)$, we have $\ell_1 \leq \ell_2$.

By definition of sufficient Z-deadlock scheme, for each process $p \in \mathsf{Proc}$ we can find a blocking pattern $(\pi_p, B_p) \in \Pi_p$ with the following properties.

- If $ds(p) = \perp$ or $p \notin \mathit{Proc}_Z$ then p is Z-lockable. Hence we can choose $B_p \subseteq Z$ such that $(*\emptyset, B_p) \in \Pi_p$.

- If $ds(p) = \ell_1 \xrightarrow{p} \ell_2$ then there exists a blocking pattern $(\pi_p, \{\ell_2\}) \in \Pi_p$. If moreover $ds(p)$ is a weak edge then we can take $\pi_p = *\emptyset\{\ell_1\}$ a weak pattern.

As all locks of Z have exactly one outgoing edge in $ds(Proc_Z)$, and as all $\textsc{Holds}(\pi_p)$ with $p \notin Proc_Z$ or $ds(p) = \bot$ are empty, the sets $\textsc{Holds}(\pi_p)$ are pairwise disjoint and $\bigcup_{p\in\mathsf{Proc}} B_p \subseteq Z \subseteq \bigcup_{p\in\mathsf{Proc}} \textsc{Holds}(\pi_p)$.

By definition of $\leq$, for all p such that π_p is a strong pattern $*\{\ell_1,\ell_2\}\{\ell_1\}$ we have $\ell_1 \leq \ell_2$.

By Lemma 4.28, the properties shown above imply that there is a reachable global deadlock. □

From now on we fix a behaviour Π. We will show how to decide if there is a sufficient deadlock scheme for Π in PTIME. Definition 4.33. Our approach will be to eliminate edges from G_Π and construct a Z-deadlock scheme incrementally, on bigger and bigger sets of locks Z. We show that this process either exhibits a set Z that is big enough to provide a sufficient deadlock scheme for Π, or it fails, and then there is no sufficient deadlock scheme for Π.

The next lemma shows that a Z-deadlock scheme can be constructed incrementally. Suppose we already have a set Z on which we know how to construct a Z-deadlock scheme. Then the lemma says that in order to get a sufficient deadlock scheme for Π it is enough to focus on $G_\Pi \setminus Z$.

Lemma 4.36

Let $Z \subseteq \mathbf{L}$ be such that there is no solid edge from Z to $\mathbf{L} \setminus Z$ in G_Π. Suppose that $\mathtt{ds}_Z : Proc_Z \to E \cup \{\bot\}$ is a Z-deadlock scheme for Π. If there exists some sufficient deadlock scheme for Π then there is one which is equal to $\mathtt{ds}_Z$ over $Proc_Z$.

Proof. Suppose $\mathtt{ds}$ is a sufficient deadlock scheme for Π, so $\mathtt{ds}$ is a B-deadlock scheme for some $B \subseteq \mathbf{L}$ such that for every $p \in \mathsf{Proc}$ either $\mathtt{ds}(p)$ is an edge in G_Π or p is B-lockable in Π. We construct a $(B \cup Z)$-deadlock scheme $\mathtt{ds}'$ which is equal to $\mathtt{ds}_Z$ over $Proc_Z$. Then we show that the deadlock scheme is sufficient.

For every process $p \in \mathsf{Proc}$, we define $\mathtt{ds}'(p)$ as:

- $\mathtt{ds}_Z(p)$ if $p \in Proc_Z$,
- $\bot$ if p labels an edge from Z to $\mathbf{L} \setminus Z$,
- $\mathtt{ds}(p)$ otherwise.

We check that $\mathtt{ds}'$ is a $(B\cup Z)$-deadlock scheme. The assumption is that there are no solid edges from Z to $\mathbf{L} \setminus Z$, thus all processes mapped to $\bot$ are fragile. Every lock $\ell \in B \cup Z$ has at most one outgoing edge in $\mathtt{ds}'$, since it can only come from $\mathtt{ds}_Z$, if $\ell \in Z$, or from $\mathtt{ds}$, if $\ell \in B \setminus Z$. We verify that there is at least one outgoing edge. By definition of Z-deadlock scheme there is one outgoing edge from every lock in Z. A lock $\ell \in B \setminus Z$ has exactly one outgoing edge in $\mathtt{ds}(\mathsf{Proc})$, and this edge in conserved in $\mathtt{ds}'$. Finally, there cannot be any strong cycle in $\mathtt{ds}'(\mathsf{Proc})$ as there are none within Z or $B \setminus Z$ and there are no edges from Z to $\mathbf{L} \setminus Z$ in $\mathtt{ds}'$.

It remains to show that $\mathtt{ds}'$ is a sufficient deadlock scheme for Π. Let $p \in \mathsf{Proc}$ be an arbitrary process. We make a case distinction on the locks of p. The first case is when both locks are in Z. If p is solid then $\mathtt{ds}'(p) = \mathtt{ds}_Z(p) \neq \bot$. If p is fragile then it is Z-lockable by definition of fragile. The second case is when one lock is in $B \setminus Z$ and the other in $B \cup Z$. If p is solid then $\mathtt{ds}(p)$ must be defined because $\mathtt{ds}$ is a sufficient deadlock scheme. We must have $\mathtt{ds}'(p) = \mathtt{ds}(p)$ as there are no solid edges from Z to $\mathbf{L} \setminus Z$. If p is fragile then p is $B \cup Z$-lockable. The final case is when one lock of p is not in $B \cup Z$. Since ds is a sufficient deadlock scheme, p is B-lockable, so it is $B \cup Z$-lockable. □

Recall that we have fixed a behaviour Π, and that G_Π is its lock graph. We will describe several polynomial-time algorithms operating on a graph $H = (\mathbf{L}, E_H)$ and a set Z of locks. Graph H will be obtained by erasing some edges from G_Π. We will say that H has a sufficient deadlock scheme to mean that there is a deadlock scheme using only edges in H that is sufficient for Π. Each of those algorithms will either eliminate some edges from H or extend Z, while maintaining the following three invariants.

Invariant 1. *G_Π has a sufficient deadlock scheme if and only if H does.*

Invariant 2. *There are no solid edges from Z to $\mathbf{L} \setminus Z$ in H.*

Invariant 3. *There exists a Z-deadlock scheme in G_Π.*

Invariant 1 expresses that the edges we removed from G_Π to get H were not useful for the deadlock scheme. Invariant 2, along with Lemma 4.36, guarantees that we can always extend a deadlock scheme over Z to a sufficient one if it exists. Invariant 3 maintains the existence of a deadlock scheme over Z.

We extend Z as much as we can while maintaining those. In the end we either obtain a sufficient deadlock scheme (that is, Z is large enough so that all processes outside of Proc_Z are Z-lockable), or a non-sufficient one that we cannot extend anymore.

We may also at some point observe contradictions in the edges of H that forbid any sufficient deadlock scheme, in which case we can conclude thanks to Invariant 1.

We start with H being the given G_Π and $Z = \emptyset$. The invariants are clearly satisfied.

Definition 4.37 ▶ Double and solo solid edges

Consider a solid process p. We say that there is a *double solid edge* $\ell_1 \overset{p}{\leftrightarrow} \ell_2$ in H if both $\ell_1 \xrightarrow{p} \ell_2$ and $\ell_1 \xleftarrow{p} \ell_2$ exist in H. We say that $\ell_1 \xrightarrow{p} \ell_2$ in H is a *solo solid edge* if there is no $\ell_1 \xleftarrow{p} \ell_2$ in H.

Our first algorithm looks for a solo solid edge $\ell_1 \xrightarrow{p} \ell_2$ and erases all other outgoing edges from ℓ_1. It is correct as a deadlock scheme for H has to map p to the edge $\ell_1 \xrightarrow{p} \ell_2$ and there must be at most one outgoing edge from every lock.

We repeat this algorithm until no more edges are removed. If some call of the algorithm fails then there is no sufficient deadlock scheme for H. Otherwise the resulting H satisfies the property:

(Trim) if a lock ℓ in $\mathbf{L} \setminus Z$ has an outgoing solo solid edge then it has no other outgoing edges in H.

We call H *trimmed* if it satisfies property (Trim).

Lemma 4.38

Suppose (H, Z) satisfies Invariants 1 to 3. If Algorithm 2 fails then H has no sufficient deadlock scheme. After a successful execution of the algorithm all the invariants are still satisfied. If a successful execution does not remove an edge from H then H satisfies (Trim).

Algorithm 2 Trimming the graph

```
1: Look for $\ell \in H \setminus Z$ with a solo solid edge $\ell \xrightarrow{p} \ell' \in E_H$ and some other outgoing edges
2: If there is no such edge then stop and report success.
3: for every edge $\ell \xrightarrow{q} \ell'' \in E_H$ from $\ell$ with $q \neq p$ do
4:     if $q$ is solid and $\ell \xleftarrow{q} \ell'' \notin E_H$ then
5:         return “no sufficient deadlock scheme for $H$”
6:     else
7:         Erase $\ell \xrightarrow{q} \ell'$
8:     end if
9: end for
```

Proof. Let H' be the graph after an execution of Algorithm 2. Observe that the algorithm does not change Z. If $H = H'$ then (Trim) holds. If the algorithm fails then there is a lock ℓ with two outgoing solo solid edges, labelled by processes p and q. Thus it is impossible to find a sufficient deadlock scheme for H, as p and q would have to be assigned to those edges (item C1 and C2) and ℓ would then have two outgoing edges, contradicting item C3.

Finally, if the algorithm succeeds but H' is smaller than H, we must show that all the invariants hold. Since the algorithm does not change Z, Invariants 2 and 3 continue to hold. For Invariant 1, suppose $\ell \xrightarrow{p} \ell'$ is the edge found by the algorithm. Observe that if H has a sufficient deadlock scheme $\mathtt{ds}_H$ then $\mathtt{ds}_H(p)$ must be this edge, as it is a solo solid edge (by items C1 and C2). So $\mathtt{ds}_H$ is also a sufficient deadlock scheme for H'. In the other direction, a sufficient deadlock scheme for H' is also sufficient for H, as H' is a subgraph of H and $\mathsf{Proc}_H = \mathsf{Proc}_{H'}$. The latter holds because H' has the same locks as H. □

Our second algorithm searches for cycles formed by solid edges and eventually adds them to Z. If such a cycle is weak then it can be added to Z. If the cycle is strong, it may still be the case that its reversal is weak (see p_1, p_2, p_3 in Figure 4.7). More precisely it may be the case that for every solid edge $\ell_i \xrightarrow{p_i} \ell_{i+1}$ in the cycle there is also a reverse edge $\ell_i \xleftarrow{p_i} \ell_{i+1}$ (which is solid by definition, since p_i is so). If the reversed cycle is also strong then there is no sufficient deadlock scheme for H. Otherwise, it is weak and it can be added to Z. The result still satisfies the invariants thanks to the (Trim) property of H.

Figure 4.7 presents a case where an application of Algorithm 3 allows us to detect an inconsistency in the solid edges, proving the absence of any deadlock scheme.

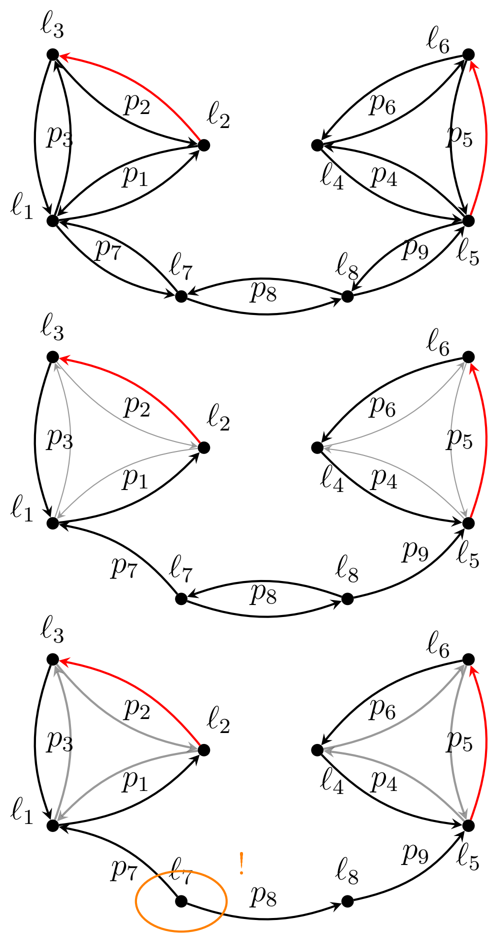
Lemma 4.39

Suppose (H, Z) satisfies the Invariants 1 to 3 and H is trimmed. If the execution of Algorithm 3 does not fail then the resulting H and Z also satisfy the invariants and (Trim). If the execution fails then there is no sufficient deadlock scheme for H.

Proof. Suppose that the algorithm finds a simple cycle $\ell_1 \xrightarrow{p_1} \ell_2 \cdots \xrightarrow{p_k} \ell_{k+1} = \ell_1$ where all p_i are solid processes, and all ℓ_i are distinct. By definition of a simple cycle, all p_i are distinct as well. If there is a sufficient deadlock scheme for H then it should assign either $\ell_i \xrightarrow{p_i} \ell_{i+1}$ or $\ell_i \xleftarrow{p_i} \ell_{i+1}$ to p_i.

Algorithm 3 Find solid cycles and add them to Z if possible.

```
1: Look for a simple cycle of solid edges $\ell_1 \xrightarrow{p_1} \ell_2 \cdots \xrightarrow{p_k} \ell_{k+1} = \ell_1$ not intersecting $Z$
   and all $\ell_i$ distinct
2: If there is no such cycle, stop and report success.
3: if all the edges on the cycle are strong then
4:     if for some $j$ there is no reverse edge $\ell_j \xleftarrow{p_j} \ell_{j+1} \in E_H$ then
5:         return "no sufficient deadlock scheme for $H$"
6:     else if all edges $\ell_j \xleftarrow{p_j} \ell_{j+1}$ are strong then
7:         return "no sufficient deadlock scheme for $H$"
8:     end if
9: end if
10: $Z \leftarrow Z \cup \{\ell_1, \dots, \ell_k\}$
11: For every $\ell_i$ remove from $E_H$ all edges outgoing from $\ell_i$ except for $\ell_i \xleftarrow{p_i} \ell_{i+1}$.
12: if some solid process $p$ has no edge in $H$ then
13:     return "no sufficient deadlock scheme for $H$"
14: end if
15: repeat
16:     Apply Algorithm 2
17: until no more edges are removed from $H$
```



This graph does not have a sufficient deadlock scheme (all processes are solid, weak edges are displayed in red). However a first execution of Algorithm 2 has no effect as all edges are double.

We apply Algorithm 3, which finds solid cycles, erases all other edges going out of those cycles, and makes sure that those cycles are weak.

We now apply Algorithm 2 again. It detects that $\ell_8 \xrightarrow{p_9} \ell_5$ is a solo solid edge and it erases the other outgoing edge $\ell_8 \xrightarrow{p_8} \ell_7$. It then concludes that there is no sufficient deadlock scheme as ℓ_7 has two outgoing solo solid edges.

Figure 4.7: An example of application of Algorithm 2 and Algorithm 3.

We examine the cases when the algorithm fails. The first reason for failure may appear when all the edges on the cycle are strong. If for some j there is no reverse edge $\ell_j \xleftarrow{p_j} \ell_{j+1}$ in E_H then a sufficient deadlock scheme for H, call it $\mathtt{ds}_H$, should assign the edge $\ell_j \xrightarrow{p_j} \ell_{j+1}$ to p_j. In consequence, as $\mathtt{ds}_H$ has to give each ℓ_i at most one outgoing edge, all the edges in the cycle should be in the image of $\mathtt{ds}_H$. This is impossible as the cycle is strong. When there are reverse edges $\ell_i \xleftarrow{p_i} \ell_{i+1} \in E_H$ for all i, the algorithm fails if all of them are strong. Indeed, there cannot be a sufficient deadlock scheme for H in this case. The last reason for failure is when there is some solid process p and all p-labelled edges were removed by the algorithm. These must be edges of the form $\ell_i \xrightarrow{p} t$ that are not on the cycle, for some $i = 1, \dots, k$. Those edges cannot belong to a deadlock scheme as it has to contain the cycle in one direction or the other and thus cannot contain other outgoing edges from that cycle. As a deadlock scheme cannot assign any edge to p, and p is solid, there cannot be a sufficient deadlock scheme in that case.

If the algorithm does not fail then either the cycle $\ell_1 \xrightarrow{p_1} \ell_2 \cdots \xrightarrow{p_k} \ell_{k+1} = \ell_1$ is weak, or its reverse is. Thanks to Lemma 4.38, we only need to show that our three invariants hold after line 11. Let (H', Z') be the values at that point. So $Z' = Z \cup \{\ell_1, \dots, \ell_k\}$, and H' is H after removing edges in line 11. We show that the invariants hold.

For Invariant 2, we observe that thanks to (Trim) for every lock in Z' there is exactly one outgoing edge in H'. So there is no solid edge from Z' to $H \setminus Z'$ as there was none from Z.

For Invariant 3, we extend our Z-deadlock scheme to Z'. We choose the cycle found by the algorithm or its reversal depending on which one is weak. For every p_i we define $ds_{Z'}(p_i)$ to be the edge in the chosen cycle. We set $ds_{Z'}(p) = \perp$ for all $p \in \mathsf{Proc}_{Z'} \setminus Proc_Z$ other than $p_1, \dots, p_k$. We must show that such a p is necessarily fragile. Indeed, in this case p must have one of its locks ℓ in Z, and the other one, ℓ', in $Z' \setminus Z$. By Invariant 2, there is no solid edge from ℓ to ℓ' in H. In H' all edges from ℓ' to ℓ are removed. So p is fragile as the algorithm does not fail at line 12.

For Invariant 1 suppose there is a sufficient deadlock scheme for H'. Then it is also a sufficient deadlock scheme for H, as H' is a subgraph of H over the same set of locks. For the other direction take $\mathtt{ds}_H$, a sufficient deadlock scheme for H. By Lemma 4.36, as we showed that Invariant 2 is maintained, we can assume that $\mathtt{ds}_H$ is equal to $\mathtt{ds}_{Z'}$ on Z'. We define a deadlock scheme $\mathtt{ds}_{H'}$ for H'. If $\mathtt{ds}_H(p) = \perp$ then $\mathtt{ds}_{H'}(p) = \perp$. If the source edge of $\mathtt{ds}_H(p)$ is in $H \setminus Z'$ then $\mathtt{ds}_{H'}(p) = \mathtt{ds}_H(p)$. This edge is guaranteed to exist in H'. If the two locks of p are both in Z' let $\mathtt{ds}_{H'}(p) = \mathtt{ds}_H(p) = \mathtt{ds}_{Z'}(p)$. The remaining case is when $\mathtt{ds}(p)$ is an edge $\ell \xrightarrow{p} \ell'$ with $\ell \in Z'$ and $\ell' \in H \setminus Z'$. In this case p is fragile as Z' has no solid edges leaving it. We can then set $ds_{H'}(p) = \perp$. It can be verified that $\mathtt{ds}_{H'}$ is a sufficient deadlock scheme for H'. $\square$

Lemma 4.40

If Algorithm 3 succeeds but does not increase Z nor decrease H then (H, Z) satisfies three properties:

H1 H is trimmed.

H2 H has no solid cycle that intersects $\mathbf{L} \setminus Z$.

H3 Every solid process has an edge in H.

Proof. Since H was not modified, Algorithm 2 did not find any solo solid edge $\ell \xrightarrow{p} \ell'$ with other outgoing edges from ℓ, hence property H1 is satisfied.

By Lemma 4.39, Invariant 2 is satisfied, hence any solid cycle intersecting $\mathbf{L} \setminus Z$ in H must be entirely in $\mathbf{L} \setminus Z$. However if such a cycle existed then Algorithm 3 would not have stopped in line 2, and thus would have either failed or increased Z. There is therefore no such cycle intersecting $\mathbf{L} \setminus Z$, hence property H2 is also satisfied.

If H3 were not satisfied then Algorithm 3 would have failed on lines 12-13. □

Since in the rest of the algorithm we increase Z but do not modify H, the three properties stated in the previous lemma will continue to hold.

Definition 4.41

For any pair (H, Z) we define an equivalence relation between locks: $\ell_1 \equiv_H \ell_2$ if $\ell_1, \ell_2 \in \mathbf{L} \setminus Z$ and there is a path of double solid edges in H between ℓ_1 and ℓ_2.

Intuitively, once we have trimmed the graph and eliminated simple cycles of solid edges with Algorithm 3, the equivalence classes of $\equiv_H$ are “trees” made of double solid edges (c.f. Lemma 4.43 below) with no outgoing edges (except for singletons, c.f. Lemma 4.42).

Lemma 4.42

If H satisfies property H1 and $\ell_1 \xrightarrow{p} \ell_2$ is in H for a solid process p then either the $\equiv_H$-equivalence class of ℓ_1 is a singleton, or $\ell_1 \xleftarrow{p} \ell_2$ is in H, hence $\ell_1 \equiv_H \ell_2$.

Proof. If the $\equiv_H$-equivalence class of ℓ_1 is not a singleton then there is a double solid edge from ℓ_1. By the (Trim) property, there cannot be any outgoing solo solid edge from ℓ_1, so $\ell_1 \xleftarrow{p} \ell_2$ must be in H, too. □

Lemma 4.43

Suppose that H satisfies properties H1 and H2. Let $\ell_1, \ell_2 \in \mathbf{L} \setminus Z$. If $\ell_1 \equiv_H \ell_2$ then H has a unique simple path of solid edges from ℓ_1 to ℓ_2.

Proof. If $\ell_1 = \ell_2$ then any non-empty simple path of solid edges from ℓ_1 to ℓ_2 would contradict property H2, hence the empty path is the only simple path from ℓ_1 to ℓ_2.

If $\ell_1 \neq \ell_2$ then by definition of $\equiv_H$ there is a path of double solid edges from ℓ_1 to ℓ_2, hence there is a simple path from ℓ_1 to ℓ_2.

Suppose there exist two distinct simple paths from ℓ_1 to ℓ_2, then by Lemma 4.42 all the locks on those paths are in the $\equiv_H$-equivalence class of ℓ_1 and ℓ_2. Hence as $\ell_1 \notin Z$, there is a cycle of double solid edges intersecting $H \setminus Z$, contradicting property H2. □

Our third algorithm looks for an edge $\ell_1 \xrightarrow{p} \ell_2$ with $\ell_1 \notin Z$ and $\ell_2 \in Z$, and adds the full $\equiv_H$-equivalence class C of ℓ_1 to Z. This step is correct, as we can extend a Z-deadlock scheme to $(Z \cup C)$-deadlock scheme by orienting edges in C, as displayed in the example in Figure 4.8.

Algorithm 4 Extending Z by locks that can reach Z

```
1: while there exists $\ell_1 \xrightarrow{p} \ell_2 \in E_H$ with $\ell_1 \notin Z$ and $\ell_2 \in Z$ do
2:     $Z \leftarrow Z \cup \{t \in \mathbf{L} \mid \ell \equiv_H \ell_1\}$
3: end while
```

Lemma 4.44

Suppose H satisfies properties H1, H2 and H3, and (H, Z) satisfies Invariants 1 to 3. After executing Algorithm 4, the new H and Z also satisfy all these properties, and H has no edges from $\mathbf{L} \setminus Z$ to Z.

Proof. Let (H', Z') be the pair obtained after executing Algorithm 4. Observe that $H' = H$, hence Invariant 1 holds. For the same reason H1 and H3 are still satisfied. Furthermore, as Z can only increase, H2 continues to hold.

Let Z_{m+1} be the value of Z at the end of the m-th iteration. So $Z_{m+1} = Z_m \cup \{t \in \mathbf{L} \mid \ell \equiv_H \ell_1\}$, where $\ell_1 \xrightarrow{p} \ell_2$ is the edge found in the guard of the while statement. We verify that Z_{m+1} satisfies Invariants 2 and 3 if Z_m does.

For Invariant 2, Lemma 4.42 says that there are no outgoing solid edges from the $\equiv_H$-equivalence class of ℓ_1, unless that class is a singleton. If it is a singleton, there are no outgoing solid edges from ℓ_1 or $\ell_1 \xrightarrow{p} \ell_2$ is the only outgoing edge of ℓ_1. In both cases, there are no solid edges from Z_{m+1} to $\mathbf{L} \setminus Z_{m+1}$ in H.

For Invariant 3 we extend a Z_m-deadlock scheme to Z_{m+1}. So we are given $\mathtt{ds}_m$ and construct $\mathtt{ds}_{m+1}$. If the two locks of some process q are in Z_m then $ds_{m+1}(q) = ds_m(q)$. We set $ds_{m+1}(p)$ to be the edge $\ell_1 \xrightarrow{p} \ell_2$ found by the algorithm, so here $\ell_1 \in Z_{m+1} \setminus Z_m$ and $\ell_2 \in Z_m$. Let C be the $\equiv_H$-equivalence class of ℓ_1: $C = \{t \in \mathbf{L} \mid \ell \equiv_H \ell_1\}$. By Lemma 4.43 there is a unique simple path from $\ell \in C$ to ℓ_1. Let $\ell \xrightarrow{q} \ell'$ be the first edge on this path. We set $ds_{m+1}(q)$ to be this edge. We set $ds_{m+1}(q) = \bot$ for all remaining processes q.

We verify that ds_{m+1} is a Z_{m+1}-deadlock scheme. By the above definition every lock in C has a unique outgoing edge in ds_{m+1}, hence every lock in Z_{m+1} does. It is also immediate that the image of ds_{m+1} does not contain a strong cycle as it would need to be already in the image of ds_m (every lock has exactly one outgoing edge in ds_{m+1} and the path obtained by following those edges from an element of C leads to Z_m). It is maybe less clear that $\mathtt{ds}_{m+1} \neq \bot$ for every solid $q \in \mathsf{Proc}_{Z_{m+1}}$. Let q be a solid process in $\mathsf{Proc}_{Z_{m+1}}$, and suppose ds_{m+1} is not defined by the procedure from the previous paragraph. If both locks of q are in Z_m then $ds_{m+1}(q)$ must be defined because $ds_m(q)$ is. If $q = p$, the process labeling the transition chosen by the algorithm, then $ds_{m+1}(q)$ is defined. Otherwise both locks of q are in C. Say these are ℓ and ℓ'. If neither $\ell \xrightarrow{q} \ell'$ is on the shortest path from ℓ to ℓ_1, nor is $\ell \xleftarrow{q} \ell'$ on the shortest path from ℓ' to ℓ_1 then there must be a cycle in C. But this is impossible as we assumed that there are no solid cycles intersecting $\mathbf{L} \setminus Z$ (property H2) and $Z \subseteq Z_m$. Hence $ds_{m+1}(q)$ is defined, and ds_{m+1} is a Z_{m+1}-deadlock scheme.

All that is left to prove is that H has no edges from $\mathbf{L} \setminus Z$ to Z, which is immediate as otherwise Algorithm 4 would not have stopped. □

Our last algorithm looks for weak cycles in the remaining graph. If it finds one, it adds to Z not only all locks in the cycle but also their $\equiv_H$-equivalence classes.

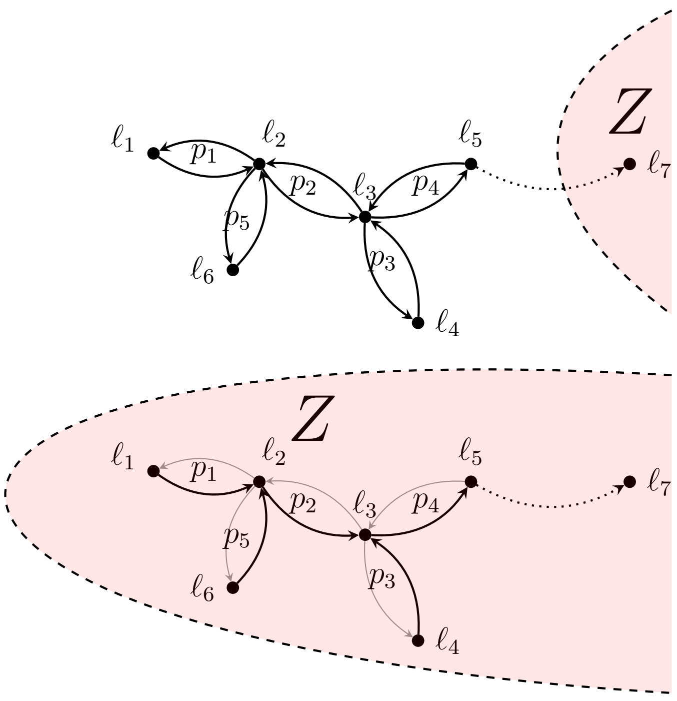


Figure 4.8: Illustration of Algorithm 4. The thick edges in the lower picture are those used in the deadlock scheme.

Algorithm 5 Incorporating weak cycles

```
1: if there exists a weak cycle $\ell_1 \xrightarrow{p_1} \ell_2 \cdots \xrightarrow{p_k} \ell_{k+1} = \ell_1$ with $\ell_k \xrightarrow{p_k} \ell_1$ weak and $\ell_i \notin Z$
   for some $i$, then
2:     $Z \leftarrow Z \cup \bigcup_{i=1}^{k} \{t \mid t \equiv_H \ell_i\}$
3: end if
```

Lemma 4.45

Suppose H satisfies H1, H2 and H3, (H, Z) satisfies Invariants 1 to 3, and moreover there are no edges from $\mathbf{L} \setminus Z$ to Z. After an execution of Algorithm 5, H still satisfies H1, H2 and H3, and the new (H, Z) satisfies Invariants 1 to 3.

Proof. Let (H', Z') be the pair obtained after execution of Algorithm 4. Observe that $H' = H$, hence Invariant 1 holds. For the same reason H1 and H3 are still satisfied. Furthermore, as Z can only increase, so is H2. It remains to verify Invariants 2 and 3.

Consider the weak cycle found by the algorithm $\ell_1 \xrightarrow{p_1} \ell_2 \cdots \xrightarrow{p_k} \ell_{k+1} = \ell_1$, and note that $\ell_i \notin Z$ for all i. Let $Z' = Z \cup \bigcup_{i=1}^{k} \{t \mid t \equiv_H \ell_i\}$ as in line 2.

For Invariant 2, consider some ℓ_i on the cycle. Lemma 4.42 says that there are no outgoing solid edges from the $\equiv_H$-equivalence class of ℓ_i, unless that class is a singleton. If this class is a singleton, there are no outgoing solid edges from ℓ_i or $\ell_i \xrightarrow{p} \ell_{i+1}$ is the only outgoing edge of ℓ_i. In both cases, there are no solid edges from Z' to $\mathbf{L} \setminus Z'$ in H.

For Invariant 3 we extend a Z-deadlock scheme $\mathtt{ds}_Z$ to Z'. For every lock $\ell \in Z' \setminus Z$ let j be the biggest index among $1, \dots, k$ with $\ell \equiv_H \ell_j$. If $\ell = \ell_j$ then set $ds_{Z'}(p_j)$ to be the edge $\ell_j \xrightarrow{p_j} \ell_{j+1}$. Otherwise, take the unique path from ℓ to ℓ_j in the $\equiv_H$-equivalence class of the two locks; this is possible thanks to Lemma 4.43. If the path starts with $\ell \xrightarrow{p} \ell'$ then set $ds_{Z'}(p)$ to this edge. Then set $ds_{Z'}(p) = \bot$ for all remaining processes p.

We claim that $\mathtt{ds}_{Z'}$ is a Z'-deadlock scheme. First, there is an outgoing $\mathtt{ds}_{Z'}$ edge from every lock in Z' because of the definition. Moreover it is unique.

We need to show that $\mathtt{ds}_{Z'}(p)$ is defined for every solid process p. This is clear if the two locks, ℓ and ℓ', of p are in Z. If both locks are not in Z then either $\ell \equiv_H \ell'$ or there is a solo solid edge between the two, say $\ell \xrightarrow{p} \ell'$. In the latter case this is the only edge from ℓ, as H is trimmed. As the $\equiv_H$-equivalence class of ℓ is then a singleton, this must be an edge on the cycle and $\mathtt{ds}_{Z'}(p)$ is defined to be this edge. Suppose $\ell \equiv_H \ell'$ and $\mathtt{ds}_{Z'}(p)$ is not defined. Let j be the biggest index among $1, \dots, k$ such that $\ell \equiv_H \ell_j$. If neither $\ell \xrightarrow{p} \ell'$ is on the shortest path from ℓ to ℓ_j, nor $\ell \xleftarrow{p} \ell'$ is on the shortest path from ℓ' to ℓ_j then there must be a cycle in C. But this is impossible as we assumed that there are no solid cycles intersecting $\mathbf{L} \setminus Z$ in H (Property H2). The remaining case is when one of the locks of p is in Z and another in $Z' \setminus Z$. There is no solid edge leaving Z by Invariant 2. There is no solid edge entering Z by the assumption of the lemma. So p is a solid process labeling no edge in H which contradicts H3.

The last thing to verify for a Z'-deadlock scheme is that there is no strong cycle in $\mathtt{ds}_{Z'}$. We first check that $\mathtt{ds}_{Z'}$ contains $\ell_k \xrightarrow{p_k} \ell_1$. This is because ℓ_k is necessarily the last from its $\equiv_H$-equivalence class. A strong cycle cannot contain locks from Z as there are no edges entering Z in $\mathtt{ds}_{Z'}$. Let $\ell'_1 \xrightarrow{p'_1} \ell'_2 \dots \xrightarrow{p'_l} \ell'_{l+1} = \ell'_1$ be a hypothetical strong cycle in $Z' \setminus Z$ using transitions in $ds_{Z'}$.

Consider x such that $\ell'_1 \equiv_H \ell'_j$ for $j \leq x$ but $\ell'_1 \not\equiv_H \ell'_{x+1}$. By definition of $ds_{Z'}$ we must have that ℓ'_x is the last lock among $\ell_1, \dots, \ell_k$ equivalent to ℓ'_1, say it is ℓ_y. As each lock only has one outgoing transition in the image of $ds_{Z'}$, and as there is a path from ℓ_y to ℓ_k in that image, ℓ_k must be on that cycle, and thus the weak edge $\ell_k \xrightarrow{p_k} \ell_1$ as well, contradicting the assumption that this is a strong cycle. □

We conclude with our complete algorithm (if one of our sub-algorithms returns a result, then the entire algorithm stops):

Algorithm 6 Algorithm to check the existence of a sufficient deadlock scheme

```
1: $H \leftarrow G_\Pi$
2: $Z \leftarrow \emptyset$
3: repeat
4:     Apply Algorithm 2
5: until No more edges are removed from $H$
6: repeat                                         ▷ $H$ is trimmed
7:     Apply Algorithm 3
8: until No more edges are removed from $H$
9: repeat                  ▷ From now on $H$ satisfies properties H1, H2 and H3
10:    Apply Algorithm 4                  ▷ no edges from $\mathbf{L} \setminus Z$ to $Z$
11:    Apply Algorithm 5
12: until $Z$ does not increase anymore
13: if there is a process $p \in \mathsf{Proc} \setminus Proc_Z$ that is not $Z$-lockable then
14:     return "No global deadlock"
15: else
16:     return "Global deadlock"
17: end if
```

Lemma 4.46

Algorithm 6 terminates in polynomial time, and fails if and only if there is no sufficient deadlock scheme for Π.

Proof. Let (H', Z') be the values at the end of the execution of the algorithm.

Suppose the algorithm fails. If it is before line 13 then using the previous lemmas and Invariant 1 we get that G_Π does not have a sufficient deadlock scheme. If the algorithm fails in line 14 then there exists a process p with one of its locks outside of Z and not Z-lockable. Suppose towards a contradiction H has a sufficient deadlock scheme $\mathtt{ds}_H$. It must have an edge from a lock of p that is not in Z, say from ℓ. By definition, every lock with an incoming edge in $\mathtt{ds}_H$ must also have an outgoing edge in $\mathtt{ds}_H$. Following these edges we get a cycle in H. During the last iteration of lines 9-12, Z was not increased, hence by Lemma 4.44 there are no edges from $\mathbf{L} \setminus Z$ to Z. This cycle is therefore outside Z. It has to be a weak cycle by definition of a deadlock scheme, which is a contradiction because Algorithm 5 did not increase Z in its last application.

If the algorithm succeeds then there is a Z-deadlock scheme, say $\mathtt{ds}_Z$ (c.f. Invariant 3). We construct a sufficient deadlock scheme $(Z, \mathtt{ds})$ for G_Π as follows. First, we set $\mathtt{ds}(p) = \mathtt{ds}_Z(p)$ for all $p \in Proc_Z$. Consider $p \in \mathsf{Proc} \setminus Proc_Z$, as the algorithm did not fail in lines 13-14, p is Z-lockable, thus we set $\mathtt{ds}(p) = \bot$.

Finally, this algorithm operates in polynomial time as all steps of all loops in the algorithms either decrease H or increase Z. Furthermore, the condition on line 13 is easily verifiable by checking in the behaviour $(\Pi_p)_{p \in \mathsf{Proc}}$ whether there exists $(*\emptyset, B) \in \Pi_p$ such that $B \subseteq Z$. □

Proof of Theorem 4.30. We start by computing the behaviour $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$ of the system. We can do this in polynomial time by Lemma 4.16.

Then, we compute the lock graph G_Π for Π and check if there is a sufficient deadlock scheme for Π in polynomial time thanks to Lemma 4.46.

By Lemma 4.35, this algorithm answers yes if and only if the system has a reachable global deadlock. □

4.6 Verification of nested LSS

In this section we address the verification problem for systems with a restricted lock acquisition policy, but no bound on the number of locks used by each process. We require that each process acquires and releases locks as if they were stored in a stack. This is a classical restriction, as this way of managing locks is considered to be sound and suitable in many contexts.

An LSS is *nested* if all its runs are such that a process can only release the lock it acquired the latest among the ones it holds.

In [**Brotherson21**] the authors proved an NP upper bound for a type of system which corresponds to our sound nested exclusive LSS on the complexity of the following problem: Is there a reachable configuration where there are some processes $p_1, \ldots, p_k \in \mathsf{Proc}$ and locks $\ell_1, \ldots, \ell_{k+1} = \ell_1 \in \mathbf{L}$ with each p_i holding lock ℓ_i and needing to get ℓ_{i+1} to keep running? We will call such configurations *circular deadlocks*. They leave the question of a matching lower bound open.

We generalise their result by proving an NP upper bound on the regular verification problem for nested LSS (note that the problem above can be solved by guessing a configuration with such a circular deadlock and using our NP algorithm to check reachability of that configuration). We then prove an NP lower bound on the process deadlock problem for sound nested exclusive LSS, and show a matching NP lower bound to the existence of circular deadlocks simultaneously.

This shows that the nested requirement significantly improves the complexity of the regular verification problem. On the other hand, the NP-hardness is difficult to avoid: it holds even for a very restricted class of systems and for very simple objectives.

Lemma 4.47

Every local run of a in a nested LSS can be decomposed as

$$\varrho = \varrho_0 a_1 \varrho_1 a_2 \cdots \varrho_{k-1} a_k \varrho_k \varrho_{k+1} \cdots$$

where $a_1, \ldots, a_k$ are the actions acquiring a lock that is not released later in the run.

Furthermore, all ϱ_i are neutral. Finally, for all $i \geq k+1$, all locks acquired in ϱ_i are acquired infinitely many times in ϱ. If ϱ is finite, all ϱ_i are empty for $i \geq k+1$. We call such a decomposition a *stair decomposition* of ϱ.

Proof. Let ϱ be a local run of some process p. We start by decomposing it as

$$\varrho = \varrho_0 a_1 \varrho_1 a_2 \cdots \varrho_{k-1} a_k \varrho_\infty$$

with $a_1, \ldots, a_k$ the actions getting a lock that is not released later in the run. For all i let ℓ_i be the lock taken while executing a_i.

We check that all $\varrho_0, \ldots, \varrho_{k-1}$ are neutral. Consider some ϱ_i. If a lock ℓ is taken in ϱ_i then it must be released somewhere in the run because a_{i+1} is the next operation that takes a lock and does not release it. But because of the nesting discipline ℓ cannot be released after a_{i+1}. So it must be released in ϱ_i.

Now we look at ϱ_∞. Every lock acquired in it must be released eventually. Thus if the run is finite we can set $\varrho_k = \varrho_\infty$ and $\varrho_i = \varepsilon$ for all $i \geq k+1$.

If the run is infinite then we proceed as follows: Before executing ϱ_∞, p holds $\ell_1, \dots, \ell_k$. We construct a sequence of neutral runs ϱ'_j such that $\varrho_\infty = \varrho'_1 \varrho'_2 \cdots$. Say we constructed $\varrho'_1 \cdots \varrho'_j$. As they are all neutral after executing them p holds $\ell_1, \dots, \ell_k$. The next action a in ϱ_∞ cannot release a lock as none of those locks are ever released. If a is neutral then we can simply set $\varrho_{j+1} = a$ If a acquires lock ℓ then let ϱ_{j+1} be the infix of ϱ_∞ starting with a and ending with the next action releasing ℓ. This run is neutral as the system is nested. Then let j be such that $\varrho'_1 \cdots \varrho'_j$ contains all $\mathtt{acq}_t$ operations with ℓ acquired finitely many times in ϱ_∞. We set $\varrho_k = \varrho'_1 \cdots \varrho'_j$ and for all $i \geq k+1$, $\varrho_i = \varrho'_{j-k+i}$. We obtain our decomposition. □

We now define patterns of local runs in a similar manner as in Section 4.3.3. While the principle of patterns stays the same, the presentation is a little different: as the number of locks per process is not bounded, notations like the ones of Definition 4.13 are not convenient.

Patterns for nested LSS describe three things:

- the set of locks that are held indefinitely in the run after some point
- the order of the last operations on each lock
- the set of locks taken and released infinitely many times

We will see that this information suffices to characterise sets of local runs that can be interleaved into a global one. This can be understood as follows. Clearly we cannot have two processes where one keeps a lock indefinitely after some point while the other one keeps taking and releasing it indefinitely. Hence we need to know which locks each process takes forever and which ones it takes infinitely often. Then, observe that if a process takes lock ℓ and never releases it later, and if it takes ℓ' after taking ℓ, then in the global run the last operation on ℓ' happens after the last operation on ℓ. Every local run thus induces a partial order on locks. To be able to schedule local runs, they must be consistent with each other on that order. This explains the second component in this definition of patterns.

Definition 4.48

Consider a (finite or infinite) local run ϱ of process p, and a stair decomposition $\varrho = \varrho_0 a_1 \cdots \varrho_{k-1} a_k \varrho_k \varrho_{k+1} \cdots$. For each $1 \leq i \leq k$ let ℓ_i be the lock acquired by a_i. We say that ϱ matches a *stair pattern* $(H, \leq, SW)$ if $H = \{\ell_1, \dots, \ell_k\}$, the set of locks acquired infinitely many times is included in SW, and $\leq$ is a total order on $\mathbf{L}$ satisfying two conditions:

- if ℓ is acquired only finitely many times and ℓ' infinitely many times then $\ell \leq \ell'$,
- if $\ell = \ell_i$ for some i and ℓ' is acquired at some point after a_i then $\ell \leq \ell'$.

Note that unlike the patterns defined for 2LSS, here a run may have several different patterns. We could define unique patterns but this would cause some problems for Lemma 4.50, as it would require automata of exponential size to recognise them.

Our next lemma characterises when local runs can be combined into a process-fair global. Once again the characterization uses only patterns and last states of the local runs.

Proposition 4.49

Consider a family of (finite or infinite) local runs $(\varrho_p)_{p\in\mathsf{Proc}}$ of a nested LSS. For each $p \in \mathsf{Proc}$ we consider a stair decomposition of ϱ_p:

$$\varrho_p = \varrho_{p,0}a_{p,1}\cdots\varrho_{p,k_p-1}a_{p,k_p}\varrho_{p,k_p}\varrho_{p,k_p+1}\cdots$$

and for each $a_{p,i}$ let $\ell_{p,i}$ be the lock taken while reading $a_{p,i}$.
Runs $(\varrho_p)_{p\in\mathsf{Proc}}$ can be scheduled into a process-fair global run if and only if there exist for each p a stair pattern $(H_p, \leq_p, SW_p)$ that ϱ_p matches and the following conditions are satisfied.

1. The sets H_p are pairwise disjoint.
2. All $\leq_p$ orders are the same.
3. For all p, if ϱ_p is finite then it leads to a state where all outgoing transitions acquire a lock from $\bigcup_{p\in\mathsf{Proc}} \textsc{Holds}_p$.
4. The set $\bigcup_{p\in\mathsf{Proc}} H_p$ is disjoint from $\bigcup_{p\in\mathsf{Proc}} SW_p$.

Proof. $\Longrightarrow$ Suppose we have a process-fair global run ϱ whose local projections are the $(\varrho_p)_{p\in\mathsf{Proc}}$. For each p let H_p be the set of locks kept indefinitely in ϱ_p and Sw_p the set of locks acquired infinitely often in ϱ_p. Let $\leq$ be a total order on locks such that for all $\ell, \ell' \in \mathbf{L}$, if in ϱ there is an operation on ℓ' after the last operation on ℓ then $\ell \leq \ell'$. In particular, a lock acquired infinitely often is always greater than one acquired finitely many times. Further, for all p, i the action $a_{p,i}$ acquires $\ell_{p,i}$, and it is not released it later. Thus $a_{p,i}$ is the last action with an operation on $\ell_{p,i}$ in ϱ. Hence if another lock ℓ is used after $a_{p,i}$ in ϱ_p, it is also used after $a_{p,i}$ in ϱ, and therefore $\ell_{p,i} \leq \ell$. As a result, $(H_p, \leq, \mathrm{Sw}_p)$ is a pattern of ϱ_p for all p, and condition 2 is immediately satisfied.

As each p eventually holds H_p and keeps those locks forever, the H_p have to be disjoint, thus condition 1 is satisfied.

For condition 3, we use the fact that ϱ is process-fair. For all p, if ϱ_p is finite then it leads to a state where after some point in the run none of the outgoing transitions can be taken. Hence all these transitions all acquire a lock that is never released after some point. This is the case for locks of $\bigcup_{p\in\mathsf{Proc}} H_p$ but not for the others, which are free infinitely often. Hence condition 3 holds.

Finally, as all locks from $\bigcup_{p\in\mathsf{Proc}} H_p$ are eventually never freed while the locks from $\bigcup_{p\in\mathsf{Proc}} \mathrm{Sw}_p$ are free infinitely often, the two sets are necessarily disjoint, proving condition 4.

$\Longleftarrow$ For the other implication, suppose that we have patterns $(H_p, \leq_p, \mathrm{Sw}_p)$ for each p such that all conditions are satisfied. Let $\leq$ be the total order on locks common to all patterns, which exists by condition 2.

We start by executing one by one for each run ϱ_p its prefix $\varrho_{p,0}$. After executing each $\varrho_{p,0}$ all locks are free, therefore we can execute all of them and end up in a configuration will all locks free.

We use the notation H for the set $\bigcup_{p \in \mathsf{Proc}} H_p$. We number the locks of H so that $H = \{\ell_1, \dots, \ell_m\}$ and $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_m$. For each $\ell_i \in H$ there is a pair (p_i, j_i) such that $\mathtt{op}(a_{p_i,j_i}) = \mathtt{acq}_{\ell_i}$. Furthermore that pair is unique as a process p cannot have $a_{p,j}$ take ℓ_i for two different j (by definition of a stair decomposition) and as the H_p are disjoint (by condition 1).

We execute, for all $\ell_i \in H$, in increasing order on i, $a_{p_i,j_i}\varrho_{p_i,j_i}$.

At first all locks are free. Then, for each i, just before we execute $a_{p_i,j_i}\varrho_{p_i,j_i}$, the locks that are not free are exactly $\{\ell_{i'} \mid i' \leq i-1\}$. Hence for every lock $\ell_{i'}$ that is not free, we have $\ell_{i'} \leq \ell_i$ and $\ell_{i'} \neq \ell_i$.

By definition of $\leq_p$, all locks ℓ' acquired in $a_{p_i,j_i}\varrho_{p_i,j_i}$ are such that $\ell_i \leq_p \ell'$, hence $\ell_i \leq \ell'$ by condition 2. As a result, they are all free just before we execute $a_{p_i,j_i}\varrho_{p_i,j_i}$. After we execute it, the set of non-free locks becomes $\{\ell_{i'} \mid i' \leq i\}$.

The projection of the resulting run on each p is $\varrho_{p,0}a_{p,1}\cdots\varrho_{p,k_p-1}a_{p,k_p}\varrho_{p,k_p}$. All that is left to do is to execute the $\varrho_{p,i}$ for $i \geq k_p+1$ for each p. They only contain operations on locks that are acquired infinitely many times which are thus in Sw_p as ϱ_p matches pattern $(\textsc{Holds}_p, \leq_p, \mathrm{Sw}_p)$, and therefore free by condition 4. As furthermore all $\varrho_{p,i}$ are neutral by definition of stair decomposition, we can execute the next $\varrho_{p,i}$ for each p again and again indefinitely, to obtain an infinite global run of the system.

This run is furthermore process-fair as the finite ϱ_p lead to states whose outgoing transitions acquire locks of H, which are eventually all taken forever. Hence those processes do not have an available action infinitely often. $\square$

Before we can present our NP algorithm, we need one last technical lemma to show that we can recognise runs with a given pattern using a small automaton.

Lemma 4.50

Given a process p of a nested LSS and a stair pattern π we can construct a DELA $\mathcal{A}_\pi^p$ such that a local run ϱ_p is accepted if and only if ϱ_p matches pattern π. The automaton $\mathcal{A}_\pi^p$ has at most $|\mathbf{L}_p|+1$ states and a formula of constant size for the accepting condition.

Proof. Let $\pi = (H_p, \leq_p, \mathrm{Sw}_p)$. We set $H_p = \{\ell_1, \dots, \ell_k\}$ so that $\ell_1 \leq_p \cdots \leq_p \ell_k$. We define the automaton $\mathcal{A}_\pi^p = (S_\pi, \Sigma_p, \delta_\pi^p, init_\pi, \varphi_\pi)$ as follows: The states of the automaton are $S_\pi = \{0, \dots, k, \infty\}$, with $init_\pi = 0$.

This automaton keeps track of the largest index i such that the run read so far is of the form $\varrho_0 a_1 \varrho_1 \cdots a_i \varrho_i$ with ϱ_j neutral for all $j < i$, $op(a_j) = \mathtt{acq}_{\ell_j}$ for all a_j and all locks ℓ' used after a_j are such that $\ell_j \leq \ell'$.

For instance, say we have taken ℓ_1, ℓ_2, ℓ_3 in that order, then we are in state 3, but then if we acquire ℓ' with $\ell_1 \leq \ell' \leq \ell_2$ then we go back to state 1.

For each action $a \in \Sigma_p$ we have the following transitions:

- If $\mathtt{op}(a) = \mathtt{nop}$ then $\delta(s,a) = s$ for all $s \in S_\pi \setminus \{k\}$ and $\delta(s,k) = \infty$.
- If $\mathtt{op}(a) = \mathtt{acq}_{\ell_i}$ for some $1 \leq i \leq k$ then $\delta(s,a) = i$ if $s \in \{i-1, \dots, k, \infty\}$ and $\delta(s,a) = s$ otherwise.

- If $\mathtt{op}(a) = \mathtt{rel}_{\ell_i}$ for some $1 \leq i \leq k$ then $\delta(s, a) = \min\{i-1, s\}$.
- If $\mathtt{op}(a) \in \{\mathtt{acq}_\ell, \mathtt{rel}_\ell\}$ for some $\ell \notin \mathrm{Sw}_p \cup H_p$ then let $j = \max\{i \mid \ell_i \leq_p \ell\}$ ($j = 0$ if that set is empty). We set $\delta(s, a) = \min\{j, s\}$.
- If $\mathtt{op}(a) \in \{\mathtt{acq}_t, \mathtt{rel}_t\}$ for some $\ell \in \mathrm{Sw}_p$ then $\delta(k, a) = \delta(\infty, a) = \infty$ and $\delta(s, a) = s$ for all $s \neq k$.

The acceptance condition φ_π is simply to see ∞ infinitely many times and all other states finitely many times.

Let ϱ be a local run of p matching the given stair pattern π, and consider its stair decomposition $\varrho = \varrho_0 a_1 \cdots \varrho_{k-1} a_k \varrho_k \varrho_{k+1} \cdots$. For all $i < k_p$ there is a path in the automaton reading $\varrho_{p,i}$ from state i to itself: every letter acquiring ℓ_{i+1} is followed by one releasing it and going back to state i, letters using a lock lower than ℓ_i for $\leq_p$ cannot appear in $\varrho_{p,i}$ as otherwise ϱ_p would not match π.

As a result, the run $\varrho_0 a_1 \cdots \varrho_{k-1} a_k$ labels a path from 0 to k in the automaton. Then all letters that appear in the ϱ_i for $i \geq k$ are greater than ℓ_k, otherwise ϱ_p would not match π. Hence the rest of ϱ_p is read between states k and ∞. If ϱ_p is finite then we stay in ∞ forever. If ϱ_p is infinite then eventually we only see letters using locks of Sw_p or $\mathtt{nop}$ and we stay in ∞ forever. In both cases the run is accepting.

Now let ϱ be a local run of p such that ϱ is accepted by $\mathcal{A}_\pi$. Then we decompose ϱ as $\varrho = \varrho_0 a_1 \cdots \varrho_{k-1} a_k \varrho_\infty$ with a_i the last letter in the run such that the prefix $\varrho_0 a_1 \cdots \varrho_{i-1} a_i$ labels a path in the automaton from 0 to i. By definition of the automaton, we must have $op(a_i) = \mathtt{acq}_{\ell_i}$ for all i and all locks used after a_i must be greater than ℓ_i for $\leq$. Similarly, all locks used in ϱ_∞ must be greater than ℓ_k.

If ϱ is finite, then so is ϱ_∞ and by the previous arguments we obtain that ϱ matches π. Otherwise, ϱ has a suffix that labels a path that stays in ∞ forever, hence after some point ϱ only contains letters using locks of Sw_p or applying $\mathtt{nop}$. As a result, ϱ matches π. □

We can finally give an NP upper bound for the problem over sound nested LSS.

Proposition 4.51

The regular verification problem is decidable in NP for sound nested LSS.

Proof. Let $\mathcal{S} = ((\mathcal{A}_p)_{p \in \mathsf{Proc}}, \mathbf{L})$ be a sound nested LSS, and $((\mathcal{B}_p)_{p \in \mathsf{Proc}}, \varphi)$ a regular objective.

The algorithm is similar to the one for Theorem 4.18: we guess a pattern $\pi_p = (H_p, \leq_p, \mathrm{Sw}_p)$ for each process p, and a valuation ν of the $(\inf_{p,q})_{p \in \mathsf{Proc}, q \in Q_p}$ (the variables of φ, see Definition 4.5). We check that ν satisfies φ. We also equip each $\mathcal{B}_p$ with an Emerson-Lei accepting condition expressing that the run matches ν.

We then guess, for each process p, a run in the product of $\mathcal{B}_p$, $\mathcal{A}_p$ and $\mathcal{A}_{\pi_p}$ (as described in Lemma 4.50) that matches valuation ν. It is folklore that if an Emerson-Lei automaton has an accepting run then it has one of the form uv^ω with u and v of polynomial size in the number of states of the automaton. Thus we can guess an accepting run within NP. An accepting run is one that respects ν in $\mathcal{B}_p$, and follows a run of $\mathcal{A}_p$ of pattern π_p (and ends in a state with all outgoing transitions getting a lock of $\bigcup_{p \in \mathsf{Proc}} H_p$ if it is finite).

By Proposition 4.49, we accept if and only if there is a process-fair global run of the LSS satisfying the objective. □

We give a matching lower bound, robust to many restrictions. The reduction also solves a question left open in [**Brotherson21**], as explained at the beginning of the section.

Proposition 4.52

The global deadlock problem, the process deadlock problem and the circular deadlock problem are NP-hard for sound nested locally live exclusive LSS.

Proof. We reduce from the independent set problem.

Let $G = (V, E)$ be an undirected graph, and $k \in \mathbb{N}$. We can assume that $V = \{1, \dots, n\}$ for some $n \in \mathbb{N}$. We set $E = \{e_1, \dots, e_m\}$, i.e., we put an arbitrary order on edges in E. Our set of processes is $\mathsf{Proc} = \{p_1, \dots, p_k\}$. For each $1 \leq j \leq m$ we have a lock ℓ_j. We write $\mathbf{L}$ for the set $\{\ell_j \mid 1 \leq j \leq m\}$. Our set of locks is $\mathbf{L} \cup \{t_1, \dots, t_k\}$. For each $v \in V$ we write E_v for the set of edges adjacent to v and $\mathbf{L}_v$ for $\{\ell_j \mid e_j \in E_v\}$. Each process p_i uses locks of $\mathbf{L} \cup \{t_i, t_{i+1}\}$, with the convention $t_{k+1} = \ell_1$.

Each process p_i has n transitions from its initial state, with operation $\mathtt{nop}$, which lead to states $s_1, \dots, s_n$. From each s_v a sequence of transitions (with no choice) acquires all locks $\ell_j \in \mathbf{L}_v$ in increasing order of indices, then acquires t_i, then t_{i+1}, and then releases all those locks in reverse order (thus ensuring the nested property). We end up in a state end_i with a local self-loop. This system is clearly exclusive, as the only state with several outgoing transitions is the initial one, and none of them acquire any lock.

In order to prove that the global deadlock problem, the process deadlock problem and the circular deadlock problem are NP-hard, we first pick an arbitrary process p_i. We prove the two following implications:

- If there is an independent set of size k then there is a run leading to a global deadlock that is also a circular deadlock (and thus also a process deadlock on p_i).
- If there is a process-fair run whose projection on some process is finite, then we can build an independent set of size k.

Proving these two implications allows us to show hardness for all three deadlock problems at once.

Suppose that this LSS has a process-fair run ϱ whose projection on some process is finite. The structure of the LSS imposes that when executing ϱ we eventually stay in the same configuration forever, with some processes blocked because they cannot acquire some lock and some looping indefinitely on their state end_i.

Let C be that configuration. If some p_i is stuck at some point after acquiring t_i, then it cannot have acquired t_{i+1}, as otherwise it could release all of its locks and loop in end_i. Hence some other process holds t_{i+1}, and it can only be p_{i+1} (with $p_{k+1} = p_1$). By iterating this reasoning, we conclude that all processes p_i are blocked while holding ℓ_i, as they cannot acquire t_{i+1}. They must be holding disjoint sets of locks. By construction, each p_i is holding t_i, plus the locks of some $\mathbf{L}_v$, $v \in V$. Hence we have a set of k vertices whose sets of adjacent edges are disjoint, i.e., an independent set of size k.

Now suppose no p_i is stuck after acquiring t_i. Then all p_i that have acquired t_i have reached end_i, and released all their locks, thus all t_i are free. There must be at least one process blocked when trying to acquire an element of some $\mathbf{L}_v$. Let j be the highest index in $\{1, \dots, m\}$ such that there is a process p_i blocked because it cannot acquire ℓ_j. Then there is a process $p_{i'}$ which is holding ℓ_j, and is itself unable to acquire some $\ell_{j'}$ (as all locks ℓ_r are free). However, as all processes acquire elements of $\mathbf{L}$ in increasing order of index, we must have $j' > j$, contradicting the maximality of j. Thus this case cannot happen, concluding the first part of our reduction.

Conversely suppose we have an independent set of vertices $S = \{v_1, \ldots, v_k\} \subseteq V$ of size k. Then we construct the run ϱ in which, one by one, each p_i first goes to s_{v_i} and then acquires $\{\ell_i\} \cup \mathbf{L}_{v_i}$. This is possible as they all acquire disjoint sets of locks. We end up in a configuration where each p_r needs ℓ_{r+1} to advance, but cannot do so as ℓ_{r+1} is held by p_{r+1}. Hence ϱ yields a circular deadlock (and even a global deadlock, which shows that it is process-fair). This ends our reduction. $\square$

Theorem 4.53

The regular verification problem is NP-complete for sound nested. The lower bound holds even for locally live exclusive LSS.

4.7 From verification to synthesis

Now that we have studied the verification problem, we can leverage our previous results to prove complexity bounds for the synthesis problem, defined below.

Definition 4.54 ▶ *Lock-sharing game*

A *lock-sharing game* (LSG for short) $(\mathcal{S}, (S_p^C, S_p^E)_{p \in \mathsf{Proc}}, \mathcal{B})$ is an LSS $\mathcal{S}$ where the set of states of each process p is partitioned in two S_p^C, S_p^E, the Controller states and the Environment states, along with a regular objective $\mathcal{B}$.
We additionally require that all actions from Controller states have operation $\mathtt{nop}$.

Remark 4.7.1. *The assumption that all actions from Controller states have operation* $\mathtt{nop}$ *is purely technical: we need it to keep the notion of deadlock natural. Suppose we had a Controller state s with two outgoing transitions taking locks ℓ_1 and ℓ_2, and the strategy of Controller went for ℓ_1. Then, if ℓ_1 is never freed by other processes, but ℓ_2, it is not clear if we should consider that the process is blocked or not.*

We call *2LSG* and *nested LSG* the LSG where the underlying system is a 2LSS and nested LSS, respectively.

A *control strategy* is a family of functions $\sigma = (\sigma_p)_{p \in \mathsf{Proc}}$ with $\sigma_p : \Sigma_p^* \to \Sigma_p$ for all p. The functions σ_p are called *local strategies*

A *σ-local run* of process p is a local run $\varrho_p = a_1 \cdots a_k$ of p such that for all prefix $a_1 \cdots a_i$ of ϱ_p, if the state reached by ϱ_p' belongs to Controller then $a_{i+1} = \sigma_p(a_1 \cdots a_i)$. A *$\sigma$-run* is a run ϱ whose projection on each process p is a σ-local run. A control strategy is *winning* if no process-fair σ-run is accepted by the regular objective.

Definition 4.55 ▶ *Regular control problem for LSG*

Given an LSG $(\mathcal{S}, (S_p^C, S_p^E)_{p \in \mathsf{Proc}}, \mathcal{B})$, decide if there is a winning control strategy.

We also define two subproblems, analogous to the ones in the verification part. The *global deadlock avoidance problem* asks, given an LSG whether there is a control strategy σ

such that no process-fair σ-run leads to a global deadlock. The *process deadlock avoidance problem* asks, given an LSG and a process p, whether there is a control strategy σ such that no process-fair σ-run has a finite projection on p.

4.7.1 Undecidability in the general case

In an extended version of [**GimMMW22**], available on Arxiv [**GimMMW24**], we showed that the global deadlock avoidance problem is already undecidable on systems where each process accesses at most 4 locks. As the proof is quite long, we present here a lightweight version, where processes use a larger (but still fixed) number of locks.

Theorem 4.56

The global deadlock avoidance problem for LSG is undecidable, even for a fixed number of locks.

Undecidability with initialisation

In a lock-sharing system all locks are assumed to be initially free. We consider now the variant where some of the locks are initially owned by some processes.

The input is a lock-sharing system $\mathcal{S} = ((\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathcal{S}^s, \mathcal{S}^e, \mathbf{L})$ and an initial configuration $C_{init} = (init_p, I_p)_{p\in\mathsf{Proc}}$ with pairwise disjoint sets $I_p \subseteq \mathbf{L}$. The question is whether there exists a control strategy that guarantees that no run from C_{init} deadlocks.

It turns out that this generalization of the deadlock-avoidance control problem is not more difficult than our original problem, as we will later see in Lemma 4.59.

Theorem 4.57

The control problem for LSG with initial configuration and at most 7 locks per process is undecidable.

We reduce from the question whether a PCP instance has an infinite solution. Let $(\alpha_i, \beta_i)_{i=1}^m$ be a PCP instance with $\alpha_i, \beta_i \in \{0,1\}^*$. We construct below a system with three processes $P, \bar{P}, C$, using locks from the set

$$\{c, s_0, s_1, p, \bar{s}_0, \bar{s}_1, \bar{p}\}\,.$$

Process P will use locks from $\{c, s_0, s_1, p\}$, process $\bar{P}$ from $\{c, \bar{s}_0, \bar{s}_1, \bar{p}\}$, and C all seven locks.

Processes $P, \bar{P}$ are supposed to synchronize over a PCP solution with the process C. That is, P and C synchronize over a sequence $\alpha_{i_1}\alpha_{i_2}\dots$, whereas $\bar{P}$ and C synchronize over a sequence $\beta_{j_1}\beta_{j_2}\dots$. The environment tells C at the beginning whether she should check index equality $i_1 i_2 \cdots = j_1 j_2 \dots$ or word equality $\alpha_{i_1}\alpha_{i_2}\cdots = \beta_{j_1}\beta_{j_2}\dots$.

For the initial configuration we assume that P owns p, $\bar{P}$ owns $\bar{p}$ and C owns $c, s_0, s_1, \bar{s}_0, \bar{s}_1$.

We describe now the three processes $P, \bar{P}, C$. Define first for $b = 0, 1$:

$$\begin{aligned}
\varrho_P(b) &= \mathtt{acq}_{s_b}\mathtt{rel}_p\,\mathtt{acq}_c\,\mathtt{rel}_{s_b}\mathtt{acq}_p\mathtt{rel}_c\\
\varrho_{\bar{P}}(b) &= \mathtt{acq}_{\bar{s}_b}\mathtt{rel}_{\bar{p}}\,\mathtt{acq}_c\,\mathtt{rel}_{\bar{s}_b}\mathtt{acq}_{\bar{p}}\mathtt{rel}_c
\end{aligned}$$

The automaton of $\mathcal{A}_P$ ($\mathcal{A}_{\bar{P}}$, resp.) allows all possible action sequences from $(\varrho_P(0) + \varrho_P(1))^\omega$ ($(\varrho_{\bar{P}}(0) + \varrho_{\bar{P}}(1))^\omega$, resp.). If e.g. process P manages to execute a sequence $\varrho_P(b_1)\varrho_P(b_2)\ldots$ then this means that C, P synchronize over the sequence $b_1, b_2, \ldots$.

Process C's behaviour for checking word equality consists in repeating the following procedure: she chooses a bit b through a controllable action, then tries to do $\varrho_C(P, b)\varrho_C(\bar{P}, b)$, with:

$$\begin{aligned}\varrho_C(P, b) &= \mathtt{rel}_{s_b}\,\mathtt{acq}_p\,\mathtt{rel}_c\,\mathtt{acq}_{s_b}\mathtt{rel}_p\,\mathtt{acq}_c\\ \varrho_C(\bar{P}, b) &= \mathtt{rel}_{\bar{s}_b}\mathtt{acq}_{\bar{p}}\,\mathtt{rel}_c\,\mathtt{acq}_{\bar{s}_b}\mathtt{rel}_{\bar{p}}\,\mathtt{acq}_c\end{aligned}$$

For index equality C's behaviour is similar: she chooses an index i and then tries to do $\varrho_C(P, b_1)\ldots\varrho_C(P, b_k)\varrho_C(\bar{P}, b'_1)\ldots\varrho_C(\bar{P}, b'_r)$, where $\alpha_i = b_1\ldots b_k$, $\beta_i = b'_1\ldots b'_r$.

The next lemma is the key property showing that the system has a deadlock-avoiding strategy if and only if the PCP instance has a solution.

Lemma 4.58

Assume that C owns $\{s_0, s_1, c\}$, P owns $\{p\}$, C wants to execute $u_C(P, b)$, and P wants to execute $u_P(b')$. Then the system deadlocks if and only if $b \neq b'$. If $b = b'$ then C and P finish executing $\varrho_C(P, b)$ and $\varrho_P(b)$, respectively, and the lock ownership is the same as before the execution.

Proof. If, say, $b = 0$ and $b' = 1$ then C releases s_0 but P wants to acquire s_1, so that P deadlocks. Since C wants to acquire p as second step, she deadlocks, too. Process $\bar{P}$ will deadlock as well, because he is waiting for either $\bar{s}_0$ or $\bar{s}_1$.

Suppose now that $b = b'$, say with $b = 0$. Then there is only one possible run alternating between steps of $\varrho_C(P, 0)$ and $\varrho_P(0)$, until C finally acquires c. Then both C and P have finished the execution of $\varrho_C(P, 0)$ and $\varrho_P(0)$, respectively. Moreover, C re-owns $\{c, s_0, s_1\}$ and P re-owns $\{p\}$. ☐

Elimination of the initialisation

We aim to prove the following lemma:

Lemma 4.59

There is a polynomial-time reduction from the control problem for lock-sharing systems with initial configuration to the control problem where all locks are initially free. The reduction adds $|\mathsf{Proc}|$ new locks.

Proof. The system $\mathcal{S} = ((\mathcal{A}_p)_{p\in\mathsf{Proc}}, \mathcal{S}^s, \mathcal{S}^e, \mathbf{L})$ with initial ownership $(I_p)_{p\in\mathsf{Proc}}$ is transformed into a new system $\mathcal{S}_\emptyset$ with additional locks. The transformation introduces one extra lock per process, denoted k_p and called *the key of* p. Each process uses in addition to $\mathbf{L}_p$ the $|\mathsf{Proc}|$ extra locks.

The transition system $\mathcal{A}_p$ of process p is extended with new states and transitions, which define a specific finite run called the *init sequence*. The new states and transitions can occur only during the init sequence. When a process p completes his init sequence in $\mathcal{S}_\emptyset$, he owns precisely all locks in I_p, plus the key k_p, and has reached his initial state $init_p$ in $\mathcal{A}_p$. After that, further actions and transitions played in $\mathcal{S}_\emptyset$ are actions and transitions of $\mathcal{S}$, unchanged. All the new actions are uncontrollable, thus there is no strategic decision to make for the controller of a process p until his init sequence is completed.

The init sequence. For process p, the init sequence consists of three steps.

1. First, p takes one by one (in a fixed arbitrary order) all locks in I_p.
2. Second, p takes and releases, one by one (in a fixed arbitrary order) all the keys of the other processes $(k_q)_{q \neq p}$.
3. Finally, p acquires his key k_p and keeps it forever.

After acquiring k_p process p reaches the initial state $init_p$ in $\mathcal{A}_p$.

In order to prevent the init sequence to create extra deadlocks, every state used in the initialisation sequence is equipped with a local self-loop on the $\mathtt{nop}$ operation. This way, a deadlock may only occur if all processes have finally completed their init sequences.

Linking runs in $\mathcal{S}_\emptyset$ and $\mathcal{S}$. When a process completes his init sequence, he has been until that point the sole owner of its initial locks:

Claim 4.59.1. *Let p be a process and $\varrho_\emptyset$ a run of $\mathcal{S}_\emptyset$ such that the last action of $\varrho_\emptyset$ is acq_{k_p} by process p. Let $\ell \in I_p$, then p is the only process to acquire ℓ in $\varrho_\emptyset$.*

Proof. By contradiction, let $\ell \in I_p$ and $q \neq p$ and assume that $\varrho_\emptyset$ factorizes as $\varrho_\emptyset = \varrho_0 \cdot (\mathsf{acq}_t, q) \cdot \varrho_1 \cdot (\mathsf{acq}_{k_p}, p)$ (we abuse the notation and denote (acq_t, q) and (acq_{k_p}, p) the transitions where q and p respectively acquire ℓ and k_p). Process p must take and release k_q before taking k_p, thus the transition $\delta = (\mathsf{acq}_{k_q}, p)$ occurs either in ϱ_0 or in ϱ_1. However δ cannot occur in ϱ_0: the init sequence of p requires that p owns ℓ permanently in the interval between the occurence of δ and the occurence of (acq_{k_p}, p), thus (acq_t, q) cannot occur in the meantime. Hence δ occurs in ϱ_1. But this leads to a contradiction: since ℓ is not an initial lock of q, process q is not allowed to acquire ℓ during his init sequence, hence q has completed his init sequence in ϱ_0. After ϱ_0, q owns permanently k_q, but then it is impossible that $\delta = (\mathsf{acq}_{k_q}, p)$ occurs during ϱ_1. □

There is a tight link between runs in $\mathcal{S}_\emptyset$ and runs in $\mathcal{S}$.

Claim 4.59.2. *Let $\varrho_\emptyset$ be a global run in $\mathcal{S}_\emptyset$ in which all processes have completed their init sequences. There exists a global run ϱ in $\mathcal{S}$ (with initial lock ownership $(I_p)_{p \in \mathsf{Proc}}$) with the same local runs as $\varrho_\emptyset$, except that the init sequences are deleted.*

Proof. The proof is by induction on the number N of transitions in $\varrho_\emptyset$ which are not transitions of the init sequence. In the base case $N = 0$, then $\varrho_\emptyset$ is an interleaving of the init sequences of all processes and ϱ is the empty run. Assume now $N > 0$. Let δ be the last transition played in $\varrho_\emptyset$ which is not part of an init sequence, and $Z \subseteq \mathsf{Proc}$ the set of processes that have not yet completed their init sequence when δ occurs. Then $\varrho_\emptyset$ factorizes as

$$\varrho_\emptyset = \varrho'_\emptyset \cdot \delta \cdot \varrho''_\emptyset$$

where $\varrho''_\emptyset$ is an interleaving of infixes of the init sequences of processes in Z.

Assume first that $\varrho''_\emptyset$ is empty. We apply the inductive hypothesis to $\varrho'_\emptyset$, get a global run ν and set $\varrho = \nu \cdot \delta$. Then ϱ has the same local runs as $\varrho_\emptyset$, after deletion of init sequences.

We now reduce the general case to the special case where $\varrho''_\emptyset$ is empty. Let q be the process operating in δ and (a, op) the corresponding pair of action and operation on locks. Since δ is not part of an init sequence, then $q \notin Z$ and op is not an operation on one of the keys. Moreover, according to Claim 4.59.1, neither is op an operation on one of the initial locks of processes in Z. Thus (a, op) can commute with all transitions in $\varrho''_\emptyset$ and become the last transition of the global run, while leaving the local runs unchanged, and we are back to the case where $\varrho''_\emptyset$ is empty. □

We turn now to the proof of the theorem.

Claim 4.59.3. *The system wins in $\mathcal{S}_\emptyset$ if and only if it wins in $\mathcal{S}$ with initial lock ownership $(I_p)_{p \in \mathsf{Proc}}$.*

Since in $\mathcal{S}_\emptyset$ there is no strategic decision to make during the init sequence, the strategies in $\mathcal{S}_\emptyset$ are in a natural one-to-one correspondence with strategies in $\mathcal{S}$. For a fixed strategy we show that there is some deadlock in $\mathcal{S}$ if and only if there is some deadlock in $\mathcal{S}_\emptyset$.

If there is a deadlock in $\mathcal{S}$ then there is also one in $\mathcal{S}_\emptyset$, by executing first all init sequences, and then the deadlocking run of $\mathcal{S}$. The execution of all init sequences is in two steps: first each process p acquires its initial locks I_p and acquires and releases the keys $k_q, q \neq p$ of other processes. Second, each process p acquires (definitively) its key k_p.

Suppose now that there is a deadlocking run $\varrho_\emptyset$ in $\mathcal{S}_\emptyset$. Observe first that all processes $p \in \mathsf{Proc}$ have completed their init sequences in ϱ, because all states used in this sequence have local $\mathtt{nop}$ self-loops. By Claim 4.59.2 there exists a global run ϱ of $\mathcal{S}$ which has the same local runs as $\varrho_\emptyset$ (apart from the init sequences). Since $\varrho_\emptyset$ is deadlocking, so is ϱ. □

4.7.2 Synthesis for 2LSS and nested LSS

Conceptually, extending our framework from verification to synthesis is not difficult when we consider the cases where we have patterns. Indeed, it suffices to guess a family of behaviours, check that there is a strategy that only allows runs within that behaviour, and finally check that the behaviour guarantees the specification.

Unfortunately, the nature of the regular objectives we defined for verification yield an NExpTime-complete control problem, even without locks.

Proposition 4.60

The regular control problem is NExpTime-complete for 2LSG and nested LSG. In fact, the lower bound does not require the use of locks.

Proof sketch. We only sketch this proof as this is a minor result and we will have several other opportunities to see the encoding of the exponential grid tiling problem in distributed synthesis problems in this manuscript.

For the upper bound, just guess a set of pairs (π, I) with π a pattern and I a set of states, for each process. Then check that there are local strategies σ_p that only allow local runs such that the pair made of the pattern of that run and the set of states it sees infinitely often is in the guessed set. This check comes down to solving a game with an objective given by a deterministic Emerson-Lei automaton with a polynomial number of states and colours and a formula of exponential size, which can be done in exponential time (we can turn the automaton into a deterministic parity automaton with exponentially many states and polynomially many colours by Proposition 2.3, and then apply Proposition 2.9). Finally, check that we can pick in each set a pair such that the resulting patterns satisfy the schedulability conditions given by Proposition 4.17 and Proposition 4.49, and the sets of states seen infinitely often satisfy the formula in the regular objective.

For the lower bound, we can reduce the exponential grid tiling problem. Take two identical processes with sets of states $\{0^x_1, 1^x_1, \dots, 0^x_n, 1^x_n, 0^y_1, 1^y_1, \dots, 0^y_n, 1^y_n\} \cup \mathit{Tiles}$. The states outside of Tiles are called the *binary states*. The transitions are so that Environment goes through a sequence of binary states, then lets Controller pick a tile, and go back to the initial state.

We can express the following conditions as Emerson-Lei conditions of polynomial size. They are listed by priority, the most important is the first.

- For all i and $z \in \{x, y\}$, if a process sees both of $0^z_i, 1^z_i$ infinitely many times, or neither, then Controller wins.
- If a process sees two tiles infinitely often then Environment wins.
- If the set of binary states seen by both processes is the same, they see the same tile infinitely often.
- If the binary states of the two processes describe adjacent tiles, then they match on their shared side.

The first condition forces Environment to pick a single coordinate for each process, the second condition makes Controller answer with a single tile. The other conditions make sure that the two processes must describe the same tiling, and that this tiling has to be consistent.

Of course, Controller cannot know in advance which set of binary states will be seen infinitely often. However, if she has a valid tiling she can simply pick the tile corresponding to the latest coordinates given by Environment. As he has to eventually repeat the same coordinates indefinitely, she will end up picking the correct tile indefinitely. ☐

On the other hand, the simpler problems can be solved in Σ^P_2 as they only depend on patterns and do not yield a combinatorial explosion on the sets of states.

We say that a local strategy σ_p *admits* a blocking pattern (π_p, B_p) if there is a finite σ-local run ϱ_p such that $\pi(\varrho_p) = \pi_p$ and $\sigma_p(\varrho_p) = a$ with $\mathtt{op}(a) = \mathtt{acq}_\ell$ and $B_p = \{\ell\}$. Similarly, σ_p admits an infinitary pattern π_p there is an infinite σ-local run ϱ_p such that $\pi(\varrho_p) = \pi_p$.

Lemma 4.61

Given a 2LSS and a family of behaviours $(\mathbb{P}_p)_{p \in \mathsf{Proc}}$, it is decidable in PTIME whether there exists a control strategy $\sigma = (\sigma_p)_{p \in \mathsf{Proc}}$ such that every blocking pattern and every infinitary pattern admitted by σ_p is in Π_p.

Proof. For each p we check whether there is a strategy σ_p that only allows runs whose patterns are in Π_p.

To do so, we simply observe that by Lemma 4.15, we can construct a parity automaton of polynomial size recognising runs whose patterns are in Π_p. Checking the existence of σ_p thus comes down to computing the winner of a parity game of polynomial size and with a constant number of priorities, which we can do in polynomial time [**Zielonka98**]. ☐

We define the behaviour of a strategy $(\sigma_p)_{p\in\mathsf{Proc}}$ as the family $(\Pi_p)_{p\in\mathsf{Proc}}$ where Π_p is the set of infinitary patterns and blocking patterns admitted by σ_p.

Theorem 4.62

The global deadlock avoidance problem is Σ_2^P-complete on 2LSG.

Proof. A strategy allows a global deadlock if and only if there exist local runs ϱ_q for each $q \in \mathsf{Proc}$ whose patterns satisfy the conditions given by Lemma 4.28.

In order to check the existence of a winning strategy, we can guess a family of behaviours $(\Pi_q)_{q\in\mathsf{Proc}}$ (note that all behaviours are of bounded size), and check that there is a strategy $(\sigma_q)_{q\in\mathsf{Proc}}$ whose local runs on each q all have patterns in Π_q, by Lemma 4.61. Then, we can make a universal guess to check that there do not exist blocking patterns $(\pi_q, B_q) \in \Pi_q$ for each $q \in \mathsf{Proc}$ satisfying the conditions given by Lemma 4.28.

This yields a Σ_2^P algorithm for the problem.

For the lower bound we reduce from the $\exists\forall$ SAT problem. Suppose we are given a formula in 3-disjunctive normal form $\bigvee_{i=1}^k \alpha_i$, so each α_i is a conjunction of three literals $\mathbf{l}_1^i \wedge \mathbf{l}_2^i \wedge \mathbf{l}_3^i$ over a set of variables $\{x_1,\dots,x_n,y_1,\dots,y_m\}$. The question is whether the formula $\varphi = \exists x_1,\dots,x_n \forall y_1,\dots,y_m, \bigvee_{i=1}^k \alpha_i$ is true.

We construct a 2LSG for which there is a winning strategy iff the formula is true. The 2LSG will use locks:

$$\{\ell_i \mid 1 \le i \le k\} \cup \{x_i, \bar{x}_i \mid 1 \le i \le n\} \cup \{y_j, \bar{y}_j \mid 1 \le j \le m\} .$$

For all $1 \le i \le n$ we have a process p_i with four states, as depicted in Fig. 4.9. In that process the system has to take both x_i and $\bar{x}_i$, and then may release one of them before being blocked in a state with no outgoing transitions. Similarly, for all $1 \le j \le m$ we have a process q_j, in which the environment has to take y_j or $\bar{y}_j$, and then is blocked.

For each clause α_i we have a process $p(\alpha_i)$ which just has one transition acquiring lock ℓ_i towards a state with a local loop on it. Hence to block all those processes the environment needs to have all ℓ_i taken by other processes. Those processes are not necessary but help to clarify the proof.

She can do that with our last kind of processes. For each clause α_i and each literal $\mathbf{l}$ of α_i there is a process $p_i(\mathbf{l})$. There the process has to acquire ℓ_i and then $\mathbf{l}$ before entering a state with a self-loop.

In order to block all processes $p(\alpha_i)$, each ℓ_i has to be taken by a process $p_i(\mathbf{l})$ for some literal $\mathbf{l}$ of α_i. For process $p_i(\mathbf{l})$ to be blocked, lock $\mathbf{l}$ has to be taken before, by some p_i or q_j.

A control strategy for the system amounts to choosing whether p_i should release x_i or $\bar{x}_i$, for each $i = 1,\dots,n$. It may also choose to release neither. Since the environment has a global view of the system, it can afterwards choose one of $y_j, \bar{y}_j$ in process q_j, for each $j = 1,\dots,m$. Those choices represent a valuation, the free lock remaining being the satisfied literal.

If the formula φ is true, then the system chooses the valuation of the x_i's in order to make φ true. As soon as processes p_i, q_j have reached their final state, we also have a valuation for the y_j's. At this point there is at least one clause α_i true, so with all its literals $\mathbf{l}_1^i, \mathbf{l}_2^i, \mathbf{l}_3^i$ true. Observe that among the 4 processes $p(\alpha_i)$ and $p_i(\mathbf{l}_j^i)$ at least one can reach its self-loop, namely the one that acquires ℓ_i first. Hence, the system does not deadlock.

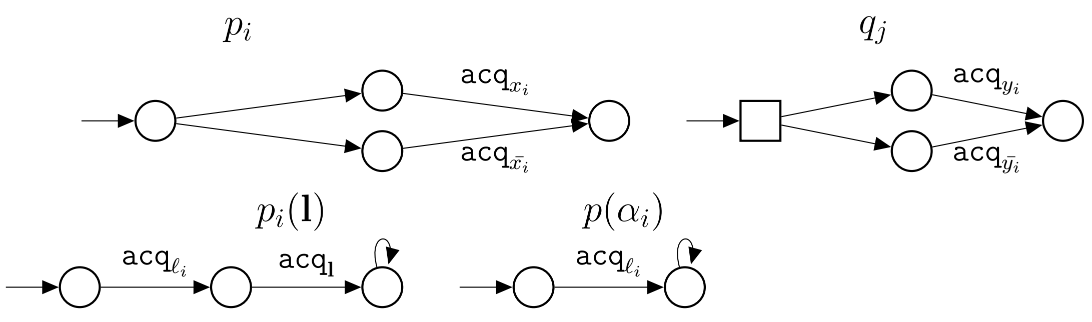


Figure 4.9: The processes used in the reduction in Theorem 4.62. Transitions of the system are dashed.

Otherwise, if the formula φ is not true, then for each choice of the system for the x_i's, the environment can chose afterwards a suitable valuation of the y_j's that falsifies φ ("afterwards" means that we look at a suitable scheduling of the acquire actions). For such a valuation, for every α_i there is some literal $\mathbf{l}_i$ of α_i that is false. Consider the scheduling that lets $p_i(\mathbf{l}_i)$ acquire ℓ_i first. Since $\mathbf{l}_i$ is taken, this implies that $p_i(\mathbf{l}_i)$ is blocked. Also, $p(\alpha_i)$ is blocked because of ℓ_i. The other two processes $p_i(\mathbf{l})$ with $\mathbf{l} \neq \mathbf{l}_i$ are also blocked because of ℓ_i. So overall the entire system is blocked. □

Corollary 4.63

The reduction above can be applied on formulas with no x_i variables. This becomes a reduction from the 3SAT problem to the global deadlock problem for exclusive 2LSS.

Theorem 4.64

The process deadlock avoidance problem is Σ_2^P-complete on locally live 2LSG.

Proof. We start with the upper bound: A strategy allows a process deadlock on p if and only if there exist local runs ϱ_q for each $q \in \mathsf{Proc}$ whose patterns satisfy the conditions given in Lemma 4.27.

To check the existence of a winning strategy, we can guess a family of behaviours $(\Pi_q)_{q\in\mathsf{Proc}}$, and check that there is a strategy $(\sigma_q)_{q\in\mathsf{Proc}}$ whose local runs on each q all have patterns in Π_q, thanks to Lemma 4.61. Then, we can make a universal guess to check that there do not exist patterns $\pi_q \in \Pi_q$ for each $q \in \mathsf{Proc}$ satisfying the conditions given by Lemma 4.27). This yields a Σ_2^P algorithm.

For the lower bound, we reduce once again from the $\exists\forall$ SAT problem. Let $\varphi = \bigvee_{i=1}^k \alpha_i$, so each α_i is a conjunction of three literals $\mathbf{l}_1^i \wedge \mathbf{l}_2^i \wedge \mathbf{l}_3^i$ over a set of variables $\{x_1, \dots, x_n, y_1, \dots, y_m\}$.

We construct a system with processes $\mathsf{Proc} = \{p, p'\} \cup \{p_i^C \mid 1 \leq i \leq m\} \cup \{p_{i,j}^{\mathbf{l}} \mid 1 \leq i \leq m, 1 \leq j \leq 3\} \cup \{p_k^x \mid 1 \leq k \leq n\} \cup \{p_k^y \mid 1 \leq k \leq m\}$, and locks $\mathbf{L} = \{t, \ell\} \cup \{t(C_i) \mid 1 \leq i \leq m\} \cup \{t(x_k), t(\neg x_k) \mid 1 \leq k \leq n\} \cup \{t(y_k), t(\neg y_k) \mid 1 \leq k \leq m\}$. The transition systems of these process are described in Figure 4.10.

In order to block process p Environment needs to block it in its first state by having another process keep ℓ forever. The only other process accessing ℓ is p'. As a consequence, a process-fair run blocks p if and only if p' eventually keeps ℓ forever.

Consider such a run ϱ. Then eventually p' has to stop visiting its state 1. Furthermore, as ϱ is process-fair, p' can never stay indefinitely in one of the other two states as it is always possible to execute a `rel` action. Hence p' goes through states 2 and 3 infinitely many times, meaning it takes and releases ℓ infinitely often.

This implies that none of the p_i^C keep ℓ indefinitely, which is only possible if all the $\ell(C_i)$ are taken and never released by other processes (if some $\ell(C_i)$ is free infinitely often, as ϱ is process-fair p_i^C has to take $\ell(C_i)$ at some point, and then ℓ cannot be free infinitely often as p_i^C would have to take it eventually).

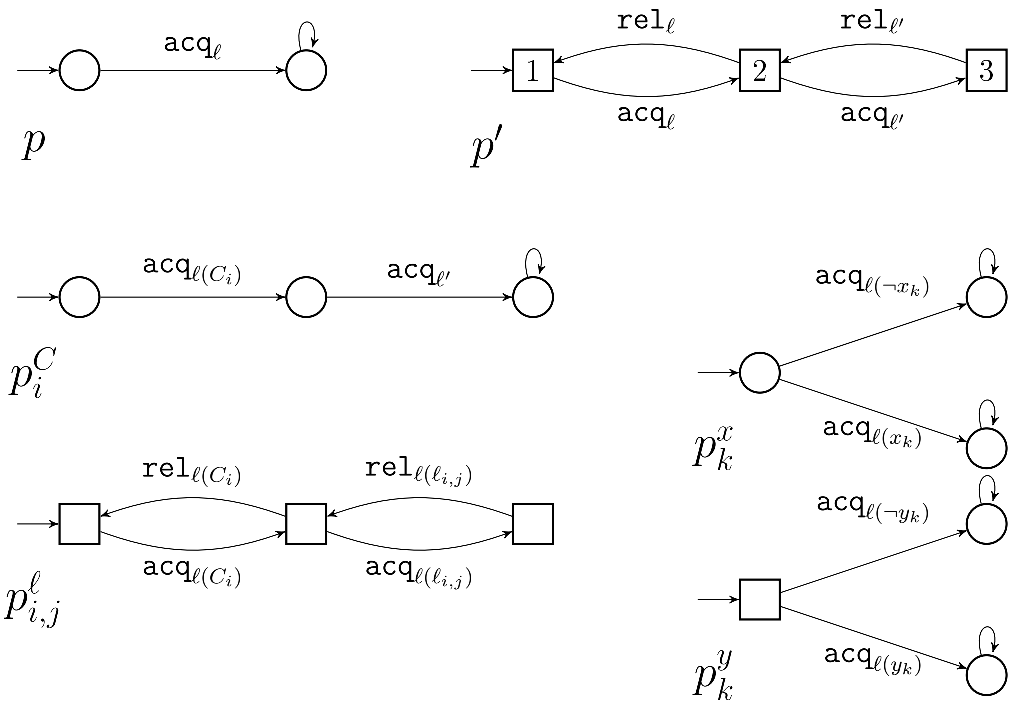
As a consequence, for each C_i there has to be a $\mathbf{l}_i^j$ such that $p_{i,j}^{\mathbf{l}}$ keeps $\ell(C_i)$ forever, which is only possible if $\ell(\mathbf{l}_{i,j})$ is free infinitely often.

This means that the process p_k^x (with x_k the variable appearing in $\mathbf{l}_{i,j}$) must have taken the lock associated with the negation of $\mathbf{l}_i^j$ (it cannot stay in its initial state as the run is process-fair and $\mathbf{l}_i^j$ is free infinitely often).

In conclusion, Environment can find a run in which p is blocked if and only if he can make sure that for all clause C_i there is a literal in C_i whose lock is free infinitely often.

The strategy of Controller comes down to choosing a valuation mapping each x_k to $\top$ if x_k is taken by p_k^x and $\bot$ otherwise. As we saw above, such a strategy is losing for Controller if and only if Environment can answer with a valuation of the y_k such that for all clause, one of the literals is mapped to $\bot$.

This corresponds exactly to the satisfaction of the input $\exists\forall$ formula. As a result, the problem is Σ_2^P-complete. $\square$



$\mathtt{acq}_\ell$
p
$\mathtt{rel}_\ell$
$\mathtt{rel}_{\ell'}$
1
2
3
$\mathtt{acq}_\ell$
$\mathtt{acq}_{\ell'}$
p'
$\mathtt{acq}_{\ell(C_i)}$
$\mathtt{acq}_{\ell'}$
p_i^C
$\mathtt{acq}_{\ell(\neg x_k)}$
$\mathtt{acq}_{\ell(x_k)}$
p_k^x
$\mathtt{rel}_{\ell(C_i)}$
$\mathtt{rel}_{\ell(\ell_{i,j})}$
$\mathtt{acq}_{\ell(C_i)}$
$\mathtt{acq}_{\ell(\ell_{i,j})}$
$p_{i,j}^\ell$
$\mathtt{acq}_{\ell(\neg y_k)}$
$\mathtt{acq}_{\ell(y_k)}$
p_k^y

Figure 4.10: Processes for the reduction in Theorem 4.64

Once again, the lower bound transfers to verification by simply considering formulas without x_k.

Corollary 4.65

The reduction above can be applied on formulas with no x_i variables. This becomes a reduction from the 3SAT problem to the global deadlock problem for exclusive 2LSS.

4.7.3 A polynomial-time algorithm for locally live exclusive 2LSS

Finally, we consider the restriction of the global deadlock avoidance problem to locally live exclusive 2LSG. We give this as an example of distributed synthesis problem solvable in polynomial time. The exclusive condition means that whenever a process can execute an action acquiring a lock it is the only thing it can do. This means that a process gets blocked whenever it tries to get a lock and the lock is not free.

In an exclusive system, if a state has an outgoing $\mathtt{acq}_\ell$ transition, then all its outgoing transitions are labelled with $\mathtt{acq}_\ell$. So in such a state the process is necessarily blocked until ℓ becomes available.

Behaviours of exclusive systems have some special properties, see Lemma 4.68. First, whenever a control strategy allows a strong edge $\{\ell_1\}\overset{p}{\Rightarrow}\{\ell_2\}$ for a process p, it also allows a reverse weak edge $\{\ell_2\}\overset{p}{\dashrightarrow}\{\ell_1\}$. This will imply that the strong cycle condition in our deadlock schemes can be satisfied automatically, because any strong cycle can be replaced by a reverse cycle of weak edges. Second, all processes that have some pattern are fragile.

The above observations simplify the analysis of the lock graph. First, we get a much simplified NP argument (Proposition 4.69). This allows us to eliminate guessing and obtain a PTIME algorithm (Proposition 4.73).

Throughout this section we fix a locally live exclusive 2LSG. We will use the tools of Section 4.5.2. We define the *behaviour* of a local strategy σ_p as the set of blocking patterns of risky σ_p-local runs and the patterns of infinite σ_p-local runs.

Lemma 4.66

Let $\sigma = (\sigma_p)_{p\in\mathsf{Proc}}$ be a control strategy, and $\Pi = (\Pi_p)_{p\in\mathsf{Proc}}$ its behaviour.
Then there is a σ-run leading to a global deadlock if and only if there is a sufficient deadlock scheme for Π.

Proof. We can see each process p with σ_p as a (possibly infinite) process. Lemma 4.35 makes no finiteness assumption, hence it applies on the 2LSS obtained by applying σ on the 2LSG.

As the behaviour of the resulting 2LSS is clearly the behaviour of σ, we obtain the result. ☐

Our first step is to observe that in the exclusive case $\mathbb{P}^\sigma$ has some special properties.

Definition 4.67

We call a behaviour $\Pi = (\Pi_p)_{p \in \mathsf{Proc}}$ *exclusive* if

- whenever Π_p contains $(*\{\ell_1, \ell_2\}\{\ell_1\}, \{\ell_2\})$ then it contains either $(*\emptyset\{\ell_1\}, \{\ell_2\})$ or $(*\emptyset\{\ell_2\}, \{\ell_1\})$, and
- whenever Π_p contains $(\pi_p, \{\ell_2\})$ with $\textsc{Holds}(\pi_p) = \{\ell_1\}$ then p is $\{\ell_1, \ell_2\}$-lockable in $\mathbb{P}$.

Remark 4.7.2. *Say we have a strong cycle* $\ell_1 \overset{p_1}{\Longrightarrow} \ell_2 \overset{p_2}{\Longrightarrow} \cdots \overset{p_k}{\Longrightarrow} \ell_{k+1} = \ell_1$ *in the lock graph* G_Π *of an exclusive behaviour* Π*, then all* p_i *have a blocking pattern* $(*\{\ell_i, \ell_{i+1}\}\{\ell_i\}, \{\ell_{i+1}\})$ *but not* $(*\emptyset\{\ell_i\}, \{\ell_{i+1}\})$*. Then by definition of exclusive behaviour, they all have a blocking pattern* $(*\emptyset\{\ell_{i+1}\}, \{\ell_i\})$*, hence there is a weak cycle* $\ell_1 = \ell_{k+1} \overset{p_k}{\dashrightarrow} \cdots \overset{p_2}{\dashrightarrow} \ell_2 \overset{p_1}{\dashrightarrow} \ell_1$.

Lemma 4.68

If σ is a locally live strategy in an exclusive 2LSS and $\Pi^\sigma = (\Pi_p)_{p \in \mathsf{Proc}}$ is its behaviour, then Π^σ is exclusive.

Proof. Consider the first condition. Suppose there is a blocking pattern $(*\{\ell_1, \ell_2\}\{\ell_1\}, \{\ell_2\}) \in \mathbb{P}_p$, then there is a process p and a local σ-run of p of the form

$$\varrho = \varrho_1 a_1 \varrho_2 a_2 \varrho_3 (a_3, \mathtt{acq}_{\ell_3}),$$

with $\mathtt{op}(a_1) = \mathtt{acq}_{\ell_1}$, $\mathtt{op}(a_2) = \mathtt{rel}_{\ell_2}$ and $\mathtt{op}(a_3) = \mathtt{acq}_{\ell_3}$ with no $\mathtt{rel}_{\ell_1}$ in ϱ_2 or ϱ_3. Hence, there is a point in the run at which p holds both locks.

If there were always a release between two acquire operations in ϱ_p then p would acquire and then release each lock without ever holding both. In consequence, there must be two acquires in ϱ with no release in-between. As the process is exclusive, the state from which the second lock is taken only has outgoing transitions taking it. Thus there is a weak edge between the first lock taken and the second one.

For the second statement, suppose that $\ell_1 \xrightarrow{p} \ell_2$ is an edge in G_Π. Thus there exists a local σ-run ϱ of p acquiring ℓ_1. The run ϱ is of the form $\varrho_1(a, \mathtt{acq}_{\ell_i})\varrho_2$ for some $i \in \{1, 2\}$ and ϱ_1 containing only local actions. As $\mathcal{S}$ is exclusive, this means that ϱ_1 makes p reach a configuration where all outgoing transitions acquire ℓ_i, and p owns no lock. Since the system is locally live this means that p is $\{\ell_i\}$-lockable, hence also $\{\ell_1, \ell_2\}$-lockable. □

Now consider a decomposition of the lock graph G_Π into strongly connected components (SCC for short).

An SCC of G_Π is a *direct deadlock* if it contains a simple cycle. A *deadlock SCC* is a direct deadlock SCC or an SCC from which a direct deadlock SCC can be reached.

Figure 4.11 illustrates these concepts: the left graph has a direct deadlock SCC formed by the three locks at the top. The two remaining locks form a deadlock SCC, because there is a path towards a direct deadlock SCC. Observe that the two locks at the bottom are not a direct deadlock SCC because there is only one process between the two locks and thus no simple cycle within the SCC.

Let BL_Π be the set of all locks appearing in some deadlock SCC.

Proposition 4.69

Consider an exclusive behaviour Π. There is a sufficient deadlock scheme for Π if and only if all processes are BL_Π-lockable.

The proof follows from the lemmas below.

Lemma 4.70

If all processes are BL_Π-lockable then there is a sufficient deadlock scheme for Π.

Proof. We construct a deadlock scheme for G_Π as follows: For all direct deadlock SCCs we select a simple cycle inside. By Remark 4.7.2 and Lemma 4.68, this cycle is weak or has a reverse weak cycle. We select a direction in which the cycle is weak, and for all ℓ in the cycle we set p_t as the process labeling the edge from ℓ in the cycle.

Then while there is an edge $\ell \xrightarrow{p} \ell$ in G_Π such that p_t is not yet defined but p_ℓ is, we set $p_t = p$. When this ends we have defined p_t for all locks in BL_Π. We define $\mathtt{ds}$ as $\mathtt{ds}(p_t) = \ell \xrightarrow{p_t} \bar{\ell}$ for all $\ell \in BL_\Pi$, and $\mathtt{ds}(p) = \bot$ for all other $p \in \mathsf{Proc}$. We show that $\mathtt{ds}$ is a sufficient deadlock scheme for $\mathbb{P}$.

Clearly, for all $p \in \mathsf{Proc}$, the value $\mathtt{ds}(p)$ is either $\bot$ or a p-labelled edge of G_Π. Furthermore, as all processes are BL_Π-lockable, in particular the ones mapped to $\bot$ by $\mathtt{ds}$ are. It is also clear that all locks of BL_Π have a unique outgoing edge. By construction of $\mathtt{ds}$ we ensured that we had no strong cycle in it. □

Lemma 4.71

If $\mathtt{ds}_Z$ is a sufficient deadlock scheme for Π then $Z \subseteq BL_\Pi$.

Proof. Suppose there is some $\ell \in Z \setminus BL_\Pi$, then there exists p such that $\mathtt{ds}_Z(p) = \ell \xrightarrow{p} \ell$, for some $\ell \in Z$. By definition of BL_Π, there are no edges from $\mathbf{L} \setminus BL_\Pi$ to BL_Π in G_Π, hence $\ell \in Z \setminus BL_\Pi$. By iterating this process we eventually find a simple cycle in G_Π outside of BL_Π, which is impossible, as this cycle should be part of a direct deadlock SCC, and thus included in BL_Π. □

Lemma 4.72

If some process p is not BL_Π-lockable then there is no sufficient deadlock scheme for $\mathbb{P}$.

Proof. Suppose there exists p that is not BL_Π-lockable. Towards a contradiction assume that there is some sufficient deadlock scheme $\mathtt{ds}_Z$ for Π.

As p is not BL_Π-lockable, then by Lemma 4.71 it is not Z-lockable either. Hence, $\mathtt{ds}(p)$ is an edge $\ell_1 \xrightarrow{p} \ell_2$ in G_Π, with $\ell_1, \ell_2 \in Z$, and thus $\ell_1, \ell_2 \in BL_\Pi$.

By Lemma 4.68, p is $\{\ell_1, \ell_2\}$-lockable, and therefore also BL_Π-lockable, yielding a contradiction. □

This concludes the proof of Proposition 4.69. We have simplified the conditions for the existence of a sufficient deadlock scheme.

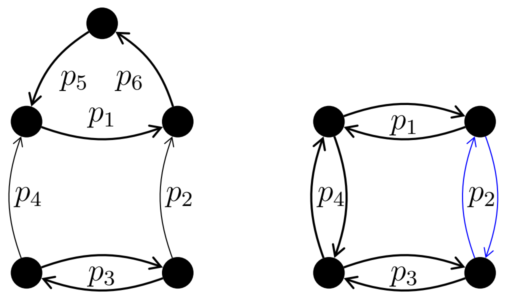


Figure 4.11: An illustration of semi-deadlock SCCs. The blue double edge is not in G_u, but every strategy of the system will induce one of those two edges.

Deciding the existence of a winning strategy for exclusive systems. Now we want to decide if there is a winning strategy. We use the insights above, but we cannot simply enumerate all exclusive behaviours, as there can be exponentially many.

For every process p and every set of edges between two locks of p we can check if there is a local strategy inducing only edges within this set, as a consequence of Lemma 4.61.

We call an edge $\ell_1 \xrightarrow{p} \ell_2$ *unavoidable* if all local strategies of p induce this edge. Let G_u be the graph whose nodes are locks and whose edges are the unavoidable edges.

We will compute a set of locks BL_u in a similar way as BL_Π in the previous section except that we will use slightly more general basic SCCs of G_u.

A *direct semi-deadlock SCC* of G_u is either a direct deadlock SCC or an SCC containing only double edges, and two locks ℓ_1 and ℓ_2 such that for some process p using ℓ_1 and ℓ_2, every strategy for p induces at least one edge between ℓ_1 and ℓ_2. Then a *semi-deadlock SCC* of G_u is either a direct semi-deadlock SCC or an SCC from which a direct semi-deadlock SCC can be reached.

Let BL_u be the set of locks appearing in semi-deadlock SCCs.

In the graph on the right part of Figure 4.11 the black edges are in G_u, the double blue ones are not, but indicate that every local strategy of process p_2 induces one of the two edges in G_Π. The four locks do not form a direct deadlock SCC of G_u as there is no simple cycle. However they do form a direct semi-deadlock one, as p_2 will induce an edge no matter its strategy, forming a simple cycle.

Proposition 4.73

There is a winning strategy if and only if there exists some process p and a local strategy σ_p that prevents p from acquiring a lock from BL_u.

Proof. $\Rightarrow$ We proceed by contraposition. We show that if all control strategies make all processes acquire a lock from BL_u then there is no winning strategy.

Let σ be a control strategy, $\mathbb{P}$ its behaviour and G_Π its lock graph. Note that G_u is a subgraph of G_Π, hence every SCC in G_Π is a superset of an SCC in G_u. Observe that if an SCC in G_Π contains a direct semi-deadlock SCC of G_u then it is a direct deadlock SCC. Indeed, if an SCC in G_u is a direct semi-deadlock but not a direct deadlock one then σ adds an edge $\ell_1 \xrightarrow{p} \ell_2$ to this SCC in G_Π. As ℓ_1, ℓ_2 are in that SCC of G_u, there is a simple path from ℓ_2 to ℓ_1 not involving p. Hence, a direct semi-deadlock SCC becomes a direct deadlock SCC. This implies $BL_u \subseteq BL_\Pi$.

Let $p \in \mathsf{Proc}$, as there is a σ-run of p acquiring a lock of BL_u, either p is BL_u-lockable (and thus BL_Π-lockable) or there is an edge labelled by p towards BL_u, meaning that

both locks of p are in $BL_u \subseteq BL_\Pi$ and thus that p is BL_Π-lockable by Lemma 4.68. As a consequence, all processes are BL_Π-lockable. We conclude by Proposition 4.69.
$\Leftarrow$ We suppose that there exists a process p and a strategy σ_p forbidding p to acquire any lock of BL_u. We construct a strategy σ such that p is not BL_Π-lockable. This will show that σ is winning by Proposition 4.69.

Let $FL_u = \mathbf{L} \setminus BL_u$ be the set of locks not in BL_u. By definition of BL_u, in G_u no node of FL_u can reach a direct semi-deadlock SCC. In particular, there is no direct semi-deadlock SCC in G_u restricted to FL_u. We construct a control strategy σ such that, when restricted to FL_u, the SCCs of G_Π and G_u are the same, where $\Pi = \Pi^\sigma$.

Let us linearly order the SCC of G_u restricted to FL_u in such a way that if a component C_1 can reach a component C_2 then C_1 is before C_2 in the order.

We use strategy σ_p for p. For every process $q \neq p$ we have one of the two cases: (i) either there is a local strategy σ_q inducing only the edges that are already in G_u; or (ii) every local strategy induces some edge that is not in G_u. In the second case there are no q-labelled edges in G_u, and for each of the two possible edges there is a local strategy inducing only this edge.

For a process q from the first case we take a local strategy σ_q that induces only the edges present in G_u.

For a process q from the second case,

- If both locks of q are in BL_u then take any local strategy for q.
- If one of the locks of q is in BL_u and the other in FL_u then choose a control strategy inducing an edge from the BL_u lock to the FL_u lock.
- If both locks of q are in FL_u then choose a control strategy inducing an edge from a smaller to a bigger SCC of G_u.

In the last case, both locks cannot be in the same SCC of G_u: As they are in FL_u, this would have to be an SCC with no simple cycles, i.e., a tree of double edges. But then the existence of q implies that this is a direct semi-deadlock SCC, which contradicts the fact that those locks are in FL_u.

Consider the graph G_Π of the resulting strategy σ. Restricted to FL_u this graph has the same SCCs as G_u. Moreover, there are no extra edges in G_Π added to any SCC included in FL_u, and there are no edges from FL_u to BL_u. As a result, we have $BL_u = BL_\Pi$. As p acquires no lock from BL_u, it is not BL_u-lockable and thus not BL_Π-lockable either. $\square$

Theorem 4.74

The global deadlock avoidance problem is in PTIME for locally live exclusive 2LSS.

Proof. We can compute BL_u in polynomial time and then check the condition from Proposition 4.73. $\square$

Remark 4.7.3. *As a final remark, we can note that many of our results can be straightforwardly adapted to pushdown automata, using the three following facts:*

- *Emptiness of a pushdown automaton is decidable in polynomial time.*
- *Emptiness of a pushdown automaton with an Emerson-Lei acceptance condition is decidable in non-deterministic polynomial time [**EsparzaHRS00**; **LeiSLZ17**].*

- *Reachability games played on the configuration graph of a pushdown automaton are decidable in exponential time, see Theorem 2.10.*

4.8 Conclusion

In this chapter we conducted a detailed study of the verification and controller synthesis problems for LSS against regular objectives. We established PSPACE-completeness for the general problem, and presented several subcases where the verification problem becomes more tractable. We used restrictions on the systems, for instance by considering 2LSS and nested LSS, and on the problem, by considering the particular cases of global deadlocks and process deadlocks. The general takeaway is that the verification of regular objectives is NP-complete on 2LSS and nested LSS. There are two subcases in which we obtain a P complexity: the global deadlock problem for locally live 2LSS and the process deadlock problem for exclusive 2LSS. Apart from those two, we show that the NP lower bound is very robust and thus justify our choice of specifications: the associated complexity is not worse than some of the very restricted cases.

We then tackle the more general problem of controller synthesis with local strategies. Mostly, we show that the problem is undecidable for general LSS, and decidable with a relatively high complexity (NExpTime) in 2LSS and nested LSS. Lastly, we exhibit three cases where the complexity improves: We obtain $\mathcal{S}_2^P$-completeness for the existence of strategies avoiding global deadlocks and process deadlocks, and a polynomial-time algorithm for locally live exclusive 2LSS with respect to global deadlocks. While this last case cumulates many restrictions, it is a non-trivial instance of distributed synthesis that can be solved in polynomial time.

At the time when this document is written, we are working with Romain Delpy towards implementing the algorithms described in this chapter (using a SAT solver for the NP-hard problems).

In terms of techniques, the main ingredient of this chapter is the definition of patterns. Sets of patterns play the same role in this chapter as invariants in the previous one: they are local specifications which characterise the non-existence of bad global runs. The stair patterns used for nested will be generalised in Section 5.6 in the next chapter. Another important aspect is the translation of problems on 2LSS into graph problems. This yields a new view on verification of LSS: we could look for other ways to interpret LSS in graphs or hypergraphs to prove the absence of deadlocks.

Future work Concerning future work, as noted in Remark 4.7.3, most of our results can easily be extended to the case when processes are pushdown systems (except for the general case, which is undecidable instead of PSPACE-complete, see [**KahIvaGup05**]). Another easy extension is to replace nested with bounded lock chains, a weaker condition defined in [**Kahlon09**]. We chose to not include them to avoid unnecessary details and highlight the key ingredients. Of course, one could also complete the detailed complexity table presented in Table 4.1 with synthesis instead of verification.

About open problems, probabilistic algorithms have proven useful in distributed systems (see, for instance, the Lehmann-Rabin algorithm [**LehmannR1981**]), hence one may want to add probabilities to the model, for instance by allowing randomised strategies. This could be especially interesting when dealing with infinite runs.

Finally, versions of the problem with shared variables in addition to the locks could be of interest. This extension of systems with locks with variables will be discussed in a more general setting in Section 5.7. We will give there some ideas on the analysis of systems with locks and variables.

Chapter 5

Dynamic lock-sharing systems

Aux arbres citoyens !

Yannick Noah

Contents

5.1 Introduction

5.1.1 Context and motivation

In this chapter, our system model is based on Dynamic Pushdown Networks (DPNs) as introduced in [**BouajjaniMT05**], where processes are pushdown systems that can spawn new processes. The DPN model was extended in [**LammichMW09**] by adding synchronization through a fixed number of locks. Here we take a step further and allow dynamic lock creation: when spawning a new process, the parent process can pass some of its locks, and new locks can be created for the new thread. This way we model recursive programs with creation of threads and locks. We call such systems *dynamic lock-sharing systems* (DLSS). In Kenter's thesis [**Kenter22**], a similar model is presented, where processes can additionally make recursive calls while passing new locks as arguments. The main result is decidability of reachability of sets of configurations defined through MSO formulas, but no complexity aspect is considered. The model considered in this chapter is weaker: We will see that we can allow processes to use a stack, which models recursive calls, but they cannot create new locks locally. On the other hand, we present a much simpler approach, and prove stronger results: we provide complexity bounds, handle infinite runs, and study synthesis.

The focus in both [**BouajjaniMT05**] and [**LammichMW09**] is computing the Pre^* of a regular set of configurations, and they achieve this by extending suitably the saturation technique from [**BouajjaniEM97**]. Here we consider not only reachability but also infinite behaviours of DLSS under fairness conditions. For this we propose a different approach than saturation from [**BouajjaniMT05**; **LammichMW09**] as saturation is not suited to cope with liveness properties.

We begin by showing that verifying regular properties of DLSS is decidable if every process follows nested lock usage, as defined in the previous chapter. This means that locally every process acquires and releases the locks according to the stack discipline. Without any restriction on lock usage we show that our problem is undecidable, even for finite state processes and reachability properties that refer to a single process. Then we use the tools defined in the verification work to define a new form of patterns to prove that controller synthesis is decidable.

5.1.2 Structure of the chapter.

Our starting point is to use trees to represent configurations of DLSS. This representation was introduced in [**LammichMW09**]. The advantage is that it does not require to talk about interleavings of local runs of processes. Instead it represents every local run as a left branch in a tree and the spawn operations as branching to the right. At each computation step one or two nodes are added below a leaf of the current configuration. Thus, the result of a run of DLSS is an infinite tree that we call a *limit configuration.* Our first observation is that it is easy to read out from a limit configuration of a run if the run is strongly process-fair (Proposition 5.1).

An important step is to characterize those trees that are limit configurations of runs of a given *finite state* DLSS, namely where every process is a finite state system. This is done in Lemma 5.8. To deal with lock creation this lemma refers to the existence of some global acyclic relation on locks. We show that this global relation can be recovered from local orderings in every node of the configuration tree (Lemma 5.9). Finally, we show that there is a nondeterministic Büchi tree automaton verifying all the conditions of Lemmas 5.8 and 5.9. This is the desired tree automaton recognizing limit configurations of process-fair runs. Our verification problem is solved by checking if there is a tree satisfying the specification and accepted by this automaton. This way we obtain the upper bound from Theorem 5.5. Surprisingly the size of the Büchi automaton is linear in the size of DLSS, and exponential only in the *arity* of the DLSS, which is the maximal number of locks a process can access.

The extension of our construction to pushdown processes requires one more idea to get an optimal complexity. In this case, ensuring that the limit tree represents a computation requires using pushdown automata. Hence, the Büchi tree automaton as described in the previous paragraph becomes a pushdown Büchi automaton on trees. The emptiness of pushdown Büchi tree automata is EXPTIME-complete, which is an issue as the automaton constructed is already exponential in the size of the input. However, we observe that the automata we obtain are right-resetting, since new threads are spawned with empty pushdown. Intuitively, the pushdown is needed only on left paths of the configuration tree to check correctness of local runs. A right-resetting automaton resets its stack each time it goes to the right child. We show that the emptiness of right-resetting parity pushdown tree automata can be checked in PTIME if the biggest priority in the parity condition is fixed (if it is not fixed then the problem is at least as complex as solving parity games). This gives the upper bound from Theorem 5.6. We obtain the matching lower bound by proving EXPTIME-hardness of checking if a process of the DLSS has an infinite run (Proposition 5.19). This holds even for finite state processes. We also show that even for finite state processes the DLSS verification problem is undecidable if we allow arbitrary usage of locks (Theorem 5.3).

In Section 5.6, we leverage the characterisation of limit configurations to show decidability of the controller synthesis problem for those systems. More precisely, we construct extended patterns, similarly to Definition 4.13 by combining the characterisation of Lemma 5.8 with the acceptance condition of the objective, given by a parity automaton. We construct patterns for local runs with enough information to characterise the existence of a bad global run (Lemma 5.27). The synthesis problem is then solved by enumerating sets of extended patterns and checking for each one if there is a strategy to ensure that all local runs only have patterns in this set. We show that the controller synthesis problem is 2EXPTIME-complete and NEXPTIME-hard. The gap can be closed by considering objectives given by deterministic tree automata. Interestingly, we observe that by fixing the objective and the arity of processes we obtain an NP upper bound.

Finally, in Section 5.7 we discuss the extension of this model with shared variables. We show that this causes undecidability of state reachability even with nested locks (Theorem 5.33). However, we point towards an exciting research direction, in which the process writing on the variable can only change a bounded number of times.

5.2 Definitions and results

A dynamic lock-sharing system is a set of processes, where each process has access to a set of locks and can spawn other processes. A spawned process can inherit some locks of the spawning process and can also create new locks. All processes run in parallel. A run of the system must be fair, meaning that if a process can move infinitely many times then it eventually does.

More formally, we start with a finite set of *process types* Proc. Each process type $p \in \mathsf{Proc}$ has an *arity* $ar(p) \in \mathbb{N}$ telling how many locks the process uses. The process can refer to these locks through the variables $Var(p) = \{x_1^p, \dots, x_{ar(p)}^p\}$. At each step a process can do one of the following operations:

$$\begin{aligned}\mathsf{Op}(p) =& \{\mathtt{nop}\} \cup \{\mathtt{acq}_x, \mathtt{rel}_x \mid x \in Var(p)\} \\ & \cup \{\mathtt{spawn}(q,\theta) \mid q \in \mathsf{Proc}, \theta : Var(q) \to (Var(p) \cup \{\mathtt{new}\})\}\end{aligned}$$

Operation $\mathtt{nop}$ does nothing. Operation $\mathtt{acq}_x$ acquires the lock designated by x, while $\mathtt{rel}_x$ releases it. Operation $\mathtt{spawn}(q,\theta)$ spawns an instance of process q where every

$$\begin{aligned}
p_{init} :&\quad \mathtt{spawn}(\mathit{first}, \mathtt{new}, \mathtt{new})\\
\mathit{first}(x_l, x_r) :&\quad \mathtt{spawn}(\mathit{phil}, x_l, x_r); \mathtt{spawn}(\mathit{next}, x_r, \mathtt{new}, x_l)\\
\mathit{next}(x_l, x_r, x_{\mathrm{lfirst}}) :&\quad \text{or}\begin{cases}\mathtt{spawn}(\mathit{phil}, x_l, x_{\mathrm{lfirst}})\\ \mathtt{spawn}(\mathit{phil}, x_l, x_r); \mathtt{spawn}(\mathit{next}, x_r, \mathtt{new}, x_{\mathrm{lfirst}})\end{cases}\\
\mathit{phil}(x_l, x_r) :&\quad \text{repeat-forever or}\begin{cases}\mathtt{acq}_{x_l}; \mathtt{acq}_{x_r}; \text{eat}; \mathtt{rel}_{x_r}; \mathtt{rel}_{x_l}\\ \mathtt{acq}_{x_r}; \mathtt{acq}_{x_l}; \text{eat}; \mathtt{rel}_{x_l}; \mathtt{rel}_{x_r}\end{cases}
\end{aligned}$$

Figure 5.1: Dining philosophers: process *first* starts the first philosopher and an iterator process *next* starts successive philosophers. The forks, modelled as locks, are passed via variables x_l and x_r. The third variable x_{lfirst} of *next* is the left fork of the first philosopher used also by the last philosopher. The system is nested as *phil* takes and releases forks in the stack order. The arity of the system is 3.

variable of q designates a lock determined by the substitution θ; this can be a lock of the spawning process or a new lock, if $\theta(x) = \mathtt{new}$.

We require that the mapping θ is *injective* on $\mathit{Var}(p)$. That is, for all $x \neq y$, if $\theta(x) = \theta(y)$ then $\theta(x) = \theta(y) = \mathtt{new}$. This is to ensure that two variables of a process never point to the same lock.

A *dynamic lock-sharing system* (DLSS for short) is a tuple

$$\mathcal{S} = (\mathsf{Proc}, ar, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, p_{init}, \mathcal{Locks})$$

where Proc, and ar are as described above. We write $ar(\mathcal{S})$ for $\max_{p\in\mathsf{Proc}} ar(p)$ and call it the *arity* of $\mathcal{S}$. For every process type p, $\mathcal{A}_p$ is a transition system describing the behaviour of p. Process type $p_{init} \in \mathsf{Proc}$ is the initial process type. Finally, $\mathcal{Locks}$ is an infinite pool of locks.

Each transition system $\mathcal{A}_p$ is a tuple $(S_p, \Sigma_p, \delta_p, \mathtt{op}_p, \mathit{init}_p)$ with S_p a finite set of states, init_p the initial state, Σ_p a finite alphabet, $\delta_p : S_p \times \Sigma_p \to S_p$ a *partial* transition function, and $\mathtt{op}_p : \Sigma_p \to \mathsf{Op}(p)$ an assignment of an operation to each action. We require that the Σ_p are pairwise disjoint, and define $\Sigma = \bigcup_{p\in\mathsf{Proc}} \Sigma_p$. We write $\mathtt{op}(b)$ instead of $\mathtt{op}_p(b)$ for $b \in \Sigma_p$, as b determines the process type p.

For simplicity, we require that p_{init} is of arity 0. In particular, p_{init} has no $\mathtt{acq}$ or $\mathtt{rel}$ operations.

An example in Figure 5.1 presents a DLSS modeling an arbitrary number of dining philosophers. The system can generate a ring of arbitrarily many philosophers, but can also generate infinitely many philosophers without ever closing the ring.

A configuration of $\mathcal{S}$ is a labelled tree representing the runs of all active processes. The leftmost branch represents the run of the initial process p_{init}, the left branches of nodes to the right of the leftmost branch represent runs of processes spawned by p_{init} etc. So a leaf of a configuration represents the current situation of a process that is started at the first ancestor above the leaf that is a right child. A node of a configuration is associated with a process, and tells in what state the process is, which locks are available to it, and which of them it holds.

More formally, a *configuration* is a, possibly infinite, labelled tree (τ, λ), where for each node $\nu \in \tau$ the label $\lambda(\nu)$ is a tuple of the form (p, s, a, L, H) where:

- $p \in \mathsf{Proc}$ is the process type of the process executing in ν,
- $s \in \Sigma_p$ is the current state of that process,
- $a \in \Sigma_p$ is the action executed by the process at ν or $\perp \notin \Sigma$ if ν is a root,
- $L : \mathit{Var}(p) \to \mathcal{L}\mathit{ocks}$ is an assignment of locks to variables of p, and
- $H \subseteq L(\mathit{Var}(p))$ the set of locks p holds before executing a.

We use $p(\nu)$, $s(\nu)$, $a(\nu)$, $L(\nu)$ and $H(\nu)$ to address the components of the label of ν (λ will always be clear from the context). When there is no risk of confusion we will write $\mathit{Var}(\nu)$ instead of $\mathit{Var}(p(\nu))$ to simplify notations.

We write $H(\tau)$ for the set of locks *ultimately held* by some process in τ, that is, $H(\tau) = \{\ell : \exists \nu, \text{for all } \nu' \text{ on the leftmost path from } \nu, \ell \in H(\nu')\}$. If τ is finite this is the same as to say that $H(\tau)$ is the union of $H(\nu)$ over all leaves ν of τ.

The initial configuration is the tree $(\tau_{init}, \lambda_{init})$ consisting only of the root ε labelled by $(p_{init}, init_{p_{init}}, \perp, \emptyset, \emptyset)$. Suppose that ν is a leaf of τ labelled by (p, s, b, L, H), and there is a transition $s \xrightarrow{a} s'$ for some s' in $\mathcal{A}_p$. A transition between two configurations $(\tau, \lambda) \xrightarrow{\nu, a} (\tau', \lambda')$ is defined by the following rules.

- $\tau \subseteq \tau'$ and for all $\nu \in \tau$, $\lambda'(\nu) = \lambda(\nu)$, that is we do not remove or relabel any node.
- If $\mathtt{op}(a) = \mathtt{spawn}(q, \theta)$ then τ' is obtained from τ by adding two children $\nu 0, \nu 1$ of ν. The label of the left child $\nu 0$ is (p, s', a, L, H). The label of the right child $\nu 1$ is $(q, init_q, \perp, L', \emptyset)$ where $L'(x^q) = L(\theta(x^q))$ if $\theta(x^q) \neq \mathtt{new}$ and $L'(x^q) = \ell_{\nu, x^q}$ is a fresh lock, otherwise.
- Otherwise, τ' is obtained from τ by adding a left child $\nu 0$ to ν. The label of $\nu 0$ must be of the form (p, s', a, L, H') subject to the following constraints:
  - If $\mathtt{op}(a) = \mathtt{nop}$ then $H' = H$,
  - If $\mathtt{op}(a) = \mathtt{acq}_x$ and $L(x) \notin H(\tau)$ then $H' = H \cup \{L(x)\}$,
  - If $\mathtt{op}(a) = \mathtt{rel}_x$ and $L(x) \in H$ then $H' = H \setminus \{L(x)\}$.

  Note that we do not allow a process to acquire a lock it already holds, or release a lock it does not have. We assume that the transition system of each process keeps track of the set of locks it holds currently, and that it does not contain any transition contradicting this rule. We call this property *soundness*.

A *run* is a (finite or infinite) sequence of configurations $(\tau_0, \lambda_0) \xrightarrow{\nu_1, a_1} (\tau_1, \lambda_1) \xrightarrow{\nu_2, a_2} \cdots$. As the trees in a run are growing we can define the *limit configuration* of that run as its last configuration if it is finite, and as the limit of its configurations if it is infinite.

Remark 5.2.1. *Note that in a run, at every moment distinct variables of a process are associated with distinct locks: $L(\nu_i)(x) \neq L(\nu_i)(y)$ for all $x, y \in \mathit{Var}(\nu_i)$ with $x \neq y$. This is because we require that the mapping θ is injective in spawn transitions.*

Remark 5.2.2. *The labels L and H can be computed out of the other three labels in the tree just following the transition rules (up to renaming locks). We could have defined configurations as trees with only three labels (p, s, a), but we preferred to include L and H for readability. Yet, later we will work with tree automata recognizing configurations and there it will be important that the labels come from a finite set.*

A configuration (τ, λ) is *fair* if for no leaf ν there is a transition $(\tau, \lambda) \xrightarrow{\nu,a} (\tau', \lambda')$ for some a and (τ', λ). We show that this compact definition of fairness captures strong process fairness of runs. As in Chapter 4, a run is *process-fair* if whenever from some position in the run a process is enabled infinitely often then it moves after this position.

Proposition 5.1

Consider a run $(\tau_0, \lambda_0) \xrightarrow{\nu_1,a_1} (\tau_1, \lambda_1) \xrightarrow{\nu_2,a_2} \cdots$ and its limit configuration (τ, λ). The run is process-fair if and only if (τ, λ) is fair.

Proof. $\Rightarrow$ Suppose towards a contradiction that (τ, λ) is not fair. So there is a transition $(\tau, \lambda) \xrightarrow{\nu,a} (\tau', \lambda')$ for some leaf ν. Let p be the process moving in ν, and let (τ_i, λ_i) be the first configuration where ν appears in τ_i. We show that p moves after this configuration, contradicting the fact that ν is a leaf.

If $\mathtt{op}(a)$ is not an $\mathtt{acq}$ operation then (ν, a) is enabled in every configuration (τ_j, λ_j) for $j > i$. By process-fairness, p must move after i.

A more interesting case is when $\mathtt{op}(a)$ is $\mathtt{acq}_x$ for some x. Let $\ell = L(\nu)(x)$ be the lock taken by the transition. As (ν, a) is enabled in τ, we have that $\ell \notin H(\tau)$. We show that this implies that process p is enabled infinitely often after position i. By soundness, as ν, a is enabled, p cannot hold ℓ: so $\ell \notin H(\nu)$. If $\ell \in H(\tau_i)$ and ℓ is never released afterwards then there is a node ν' in τ_i (and thus in τ) such that for every left descendant ν'' of ν' we have $\ell \in H(\nu'')$. But this is impossible since we have assumed $\ell \notin H(\tau)$. Hence, either ℓ is free in τ_i or it is free in some later configuration $(\tau_{i_1}, \lambda_{i_1})$ such that $(\tau_{i_1}, \lambda_{i_1}) \xrightarrow{\nu',b} (\tau_{i_1+1}, \lambda_{i_1+1})$ and $\mathtt{op}(b) = \mathtt{acq}_y$ with $L(\nu')(y) = \ell$. So, (ν, a) is enabled in $(\tau_{i_1}, \lambda_{i_1})$. If ℓ is never taken after i_1 then (ν, a) is enabled always after i_1, and we get a move of p by strong fairness as before. If ℓ is taken after i_1 then by the same argument as above there must be also a position i_2 when ℓ is released. So, (ν, a) is enabled in $(\tau_{i_2}, \lambda_{i_2})$. This argument shows that (ν, a) must be enabled infinitely often after i, so by process-fairness there must be a move by p after i.

$\Leftarrow$ Suppose that (τ, λ) is process-fair, and the process p is enabled infinitely often after position i. By contradiction, assume that p does not move after position i and let ν be the last node of p's local run. If the action a of p that is enabled infinitely often is not a $\mathtt{acq}$ then a is enabled in every (τ_j, λ_j) with $j \geq i$, and $(\tau, \lambda) \xrightarrow{\nu,a} (\tau', \lambda')$, contradicting process-fairness. Else, $op(a)$ is $\mathtt{acq}_x$ with $L(\nu)(x) = \ell$. Since a is enabled infinitely often, $\ell \notin H(\tau)$. Again we have $\tau \xrightarrow{\nu,a} \tau'$, contradicting process-fairness. □

Objectives. Instead of using some specific temporal logic we stick to a most general specification formalism and use regular tree properties for specifications. A *regular objective* is given by a nondeterministic parity tree automaton $\mathcal{B}$ over $\Sigma \cup \{\perp\}$, which defines a language of accepted limit configurations. Recall that our notion of parity tree automata allows both finite and infinite branches, and has a transition function of the form $\Delta \subseteq Q \times (Q \cup Q \times Q)$. We say that a configuration τ satisfies $\mathcal{B}$ when $\mathcal{B}$ accepts the tree obtained from τ by restricting only to action labels.

Regular objectives can express many interesting properties. For example, "for every instance of process p its run is in a regular language $\mathcal{C}$". Or more complicated "there is an instance of p with a run in a regular language $\mathcal{C}_1$ and all the instances of p have runs in the language $\mathcal{C}_2$". Of course, it is also possible to talk about boolean combinations of such properties for different processes. Observe that the resulting automaton $\mathcal{B}$ for these kinds of properties can be a parity automaton with priorities $1, 2, 3$ (properties of sequences can be expressed by Büchi automata, and priority 3 is used to implement existential quantification on process instances).

Regular objectives can express deadlock properties. Since we only consider process-fair runs, a finite branch in a limit configuration indicates that a process is blocked forever after some point. Hence, we can express properties such as "there is an instance of p that is blocked forever after a finite run in a regular language $\mathcal{C}$". We can also express that all branches are finite, which is equivalent to a global deadlock since we are considering only process-fair runs.

Reachability properties are also expressible with regular objectives. We can check simultaneous reachability of several states in different branches, for instance "there is a reachable configuration in which some process p reaches s while some process p' reaches s'". There are ways to do it directly, but the shortest argument is through a small modification of the DLSS. We can simply add transitions to stop processes non-deterministically in desired states: adding new `nop` transitions from s and s' to new deadlock states. Using ideas from [**LammichMSW13**] we can also check reachability of a regular set of configurations.

Going back to our dining philosophers example from Figure 5.1, we can see also other types of properties we would like to express. For example, we would like to say that there are finitely many philosophers in the system. This can be done simply by saying that there are not infinitely many spawns in the limit configuration. (In this example it is equivalent to saying that there is no branch turning infinitely often to the right.) Then we can verify a property like "if there are finitely many processes in the system and some philosopher eats infinitely often then all philosophers eat infinitely often". This property holds under process-fairness, as philosophers release both their forks after eating.

Definition 5.2 ▶ *DLSS verification problem*

Given a DLSS $\mathcal{S}$ and a non-deterministic parity tree automaton $\mathcal{A}$ decide if there is a process-fair run of $\mathcal{S}$ whose limit configuration (τ, λ) is accepted by $\mathcal{A}$.

Without any further restrictions we show that our problem is undecidable. This result is obtained by creating an unbounded chain of processes simulating a Turing machine. Each process memorizes the content of a position on the tape, and communicates with its neighbours by interleaving lock acquisitions. The trick for processes to exchange information by interleaving lock acquisitions was already used in [**KahIvaGup05**], and requires a non-nested usage of locks.

Theorem 5.3

The DLSS verification problem is undecidable. The result holds even if the DLSS is finite-state and every process uses at most 4 locks.

Proof. The proof idea is to simulate an accepting run of TM M using n cells by spawning a chain of n processes, $P_0, \ldots, P_{n-1}$. We assume that M accepts when the head is leftmost.

The initial process P_0 uses three locks, called a, b, c_1, and acquires a, b before spawning P_1. Process P_1 uses locks a, b, c_1, plus a fresh lock c_2. It acquires c_1 before spawning P_2. More generally, process P_k ($1 \leq k < n-1$) uses locks a, b, c_k, c_{k+1}, and it acquires c_k before spawning the next process P_{k+1}. The last process P_{n-1} uses only three locks, a, b, c_{n-1}.

A configuration $(p, k, A_0 \ldots A_{n-1})$ of the TM corresponds to each P_j storing A_j, with process P_k storing in addition state p. A TM step to the right, from cell k to $k+1$, needs to communicate the next state q.

In the following we denote the process that currently owns locks a and b, as "sender". The notation S^+, S^- used below indicates that the sender tries to send the state to the right or left neighbour, respectively. Similarly, R^+, R^- indicates that a "receiver" is ready to receive from the right or the left neighbour, respectively.

Sending q from P_k to P_{k+1} is implemented by P_k using the following sequences of actions:

$$\begin{aligned} S_a^+ &= \mathtt{rel}_a \mathtt{acq}_{c_{k+1}} \mathtt{rel}_b \ \mathtt{acq}_a \mathtt{rel}_{c_{k+1}} \mathtt{acq}_b \\ S_b^+ &= \mathtt{rel}_b \mathtt{acq}_{c_{k+1}} \mathtt{rel}_a \ \mathtt{acq}_b \mathtt{rel}_{c_{k+1}} \mathtt{acq}_a \end{aligned}$$

Process P_{k+1} ("receiver") uses matching sequences:

$$\begin{aligned} R_a^- &= \mathtt{acq}_a \mathtt{rel}_{c_{k+1}} \mathtt{acq}_b \ \mathtt{rel}_a \mathtt{acq}_{c_{k+1}} \mathtt{rel}_b \\ R_b^- &= \mathtt{acq}_b \mathtt{rel}_{c_{k+1}} \mathtt{acq}_a \ \mathtt{rel}_b \mathtt{acq}_{c_{k+1}} \mathtt{rel}_a \end{aligned}$$

Suppose now that the sender P_k wants to send state q to receiver P_{k+1}. This will be done by P_k by trying to execute the sequence $(S_a^+)^q \, S_b^+$. Every process P_j with $j > k$ is ready to execute either R_a^- or R_b^-. Symmetrically, every process P_j with $j < k$ is ready to execute either R_a^+ or R_b^+.

We show next that the DLSS deadlocks if P_k, P_{k+1} do not execute $(S_a^+)^q \, S_b^+$ and $(R_a^-)^q \, R_b^-$, resp., in lockstep manner:

Claim 5.3.1. *Assume that P_k owns $\{a, b\}$, every P_j, $j < k$, owns c_{j+1}, and every P_j, $j > k$, owns c_j. Moreover, P_k wants to send a to P_{j+1}. Then either P_k, P_{k+1} execute S_a^+ and R_a^-, resp., in lockstep manner, or all processes deadlock.*

Proof of the claim. Process P_k is the only process who can start, since all other processes wait for acquiring either a or b.

After releasing a, process P_k needs c_{k+1}. It can only proceed and take c_{k+1} if P_{k+1} starts executing R_a^-, taking a and releasing c_{k+1}. Then P_k releases b, and waits to get back a. If b is taken by another process than the receiver, say P_j, $j \neq k+1$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock a will never become available because P_{j+1} will not release it, so all processes deadlock.

Assume that P_{j+1} takes b, and releases a. If a is taken by another process than the sender, say P_j, $j \neq k$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock a will never become available because P_{j+1} does not release b, so all processes deadlock.

Assume that P_j takes a back. Then it releases c_{j+1}, which can be taken only by P_{j+1}, who releases also b. If b is taken by another process than the sender, say P_j, $j \neq k$, then P_j will release its lock $c \neq c_{j+1}$, and c is now the only available lock. Lock b

will never become available because P_j does not release a anymore. Once again, all processes deadlock. ■

We conclude the proof by noting that P_0 reaches a final state of M if and only if M accepts. □

The situation improves significantly if we assume nested usage of locks.

Definition 5.4

A process $\mathcal{A}_p$ is *nested* if it takes and releases locks according to a stack discipline, i.e., for all $x, y \in \mathit{Var}(p)$, for all paths $s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n$ in $\mathcal{A}_p$, with $op(a_1) = \mathtt{acq}_x$, $op(a_n) = \mathtt{rel}_x$, $op(a_m) \neq \mathtt{rel}_x$ for all $m < n$: if $op(a_i) = \mathtt{acq}_y$ for some $i < n$ then there exists $i < k < n$ such that $op(a_k) = \mathtt{rel}_y$. A DLSS is nested if all its processes are nested.

We can state the first main result of this chapter. Its proof is outlined in the next two sections.

Theorem 5.5

The DLSS verification problem for nested DLSS is EXPTIME-complete. It is in PTIME when the number of priorities in the specification automaton, and the maximal arity of processes is fixed.

We can extend this result to DLSS where transition systems of each process are given by a pushdown automaton (see definitions in Section 5.5). The complexity remains the same as for finite state processes.

Theorem 5.6

The DLSS verification problem for nested pushdown DLSS is EXPTIME-complete. It is in PTIME when the number of priorities in the specification automaton, and the maximal arity of processes is fixed.

5.3 Characterizing limit configurations

A configuration is a labelled tree. We give a characterization of such trees that are limit configurations of a process-fair run of a given DLSS. In the following section we will show that the set of limit configurations of a given DLSS is a regular tree language, which will imply the decidability of our verification problem.

Definition 5.7

Given a configuration (τ, λ) with nodes ν, ν' and variables $x \in \mathit{Var}(\nu)$, $x' \in \mathit{Var}(\nu')$, we write $x \sim_{\tau,\lambda} x'$ if $L(\nu)(x) = L(\nu')(x')$, so if x and x' are mapped to the same lock. We will often omit the subscript and simply write $x \sim x'$ when (τ, λ) is given by the context.
The *scope* of a lock ℓ is the set $\{\nu : \ell \in L(\nu)(\mathit{Var}(\nu))\}$.

Remark 5.3.1. *It is easy to see that in any configuration, the scope of a lock is a subtree.*

We say that a node ν is labelled by an *unmatched* $\mathtt{acq}$ if it is labelled by some $\mathtt{acq}_x$ and there is no $\mathtt{rel}_x$ operation in the leftmost path starting from ν. Recall that $H(\tau)$ is the set of locks ℓ for which there is some node ν with an unmatched $\mathtt{acq}_x$ and $L(\nu)(x) = \ell$.

We define a relation $\prec_H$ on $H(\tau)$ by letting $\ell \prec_H \ell'$ if there exist two nodes ν, ν' such that ν is an ancestor of ν', ν is labelled with an unmatched $\mathtt{acq}$ of ℓ, and ν' is labelled with a $\mathtt{acq}$ of ℓ'.

After these preparations we can state a central lemma giving a structural characterization of limit configurations of process-fair runs. The intuitions behind some of the conditions are illustrated in Figure 5.2.

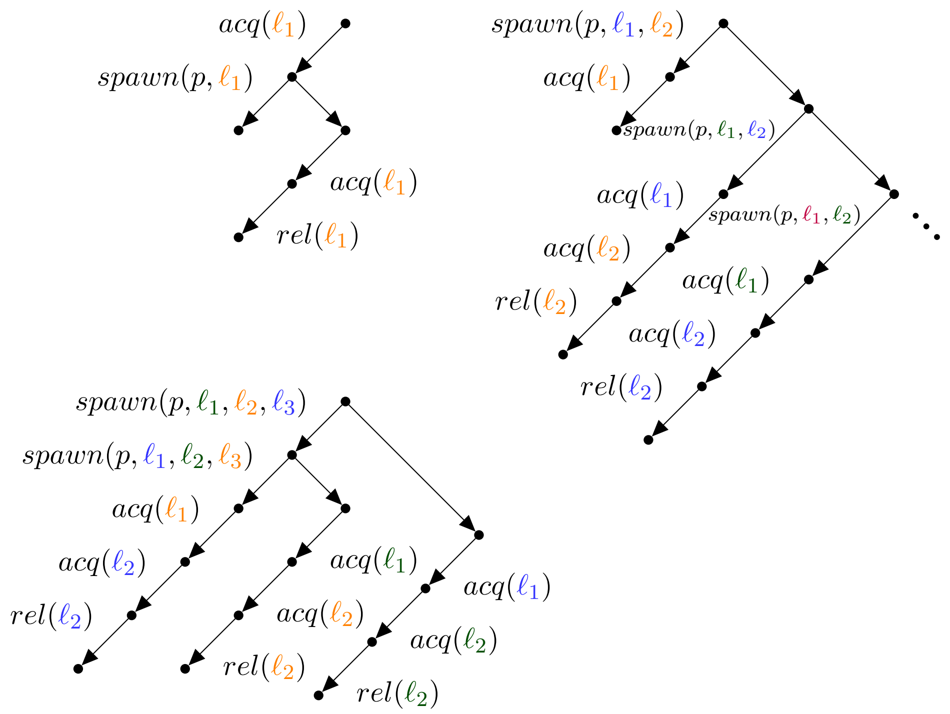


Figure 5.2: Some trees that cannot be obtained as the limit of a run.

Example 5.3.1. *In Figure 5.2 we draw examples of trees that are* ***not*** *limit configurations of runs of a DLSS. Locks are represented by colours: lock variables are coloured with the lock assigned to them.*

In the top left one, the lock 🟧 is taken before the second process is created, and never released, making it impossible for the second process to acquire it. In the top right one, we see an infinite sequence of locks and processes. Observe that lock 🟦 is acquired and never released by a process that later acquires 🟧. Thus in a run yielding that tree the last operation on 🟦 would have to be before the last operation on 🟧. Similarly, the last operation on 🟩 would have to be before the last operation on 🟦, and so on.

In the bottom tree, we see a cycle of three locks that have to be taken (and never released) before each other.

Lemma 5.8

A labelled tree (τ, λ) is the limit configuration of a process-fair run of a nested DLSS $\mathcal{S}$ if and only if all the following conditions hold

F1 The node labels in (τ, λ) match the local transitions of $\mathcal{S}$.

F2 For every leaf ν every possible transition from $s(\nu)$ has operation $\mathtt{acq}_x$ for some x with $L(\nu)(x) \in H(\tau)$.

F3 For every lock $\ell \in H(\tau)$ there are finitely many nodes with operations on ℓ, and there is a unique node labelled with an unmatched $\mathtt{acq}$ of ℓ.

F4 The relation $\prec_H$ is acyclic.

F5 The relation $\prec_H$ has no infinite descending chain.

Before presenting the proof of the previous lemma note that the main difficulty is the fact that some locks can be taken and never released. If $H(\tau) = \emptyset$ then from τ we can easily construct a run with limit configuration τ by exploiting the nested lock usage. This is because any local run can be executed from a configuration where all locks are available.

Proof. $\Longrightarrow$ Suppose that we have a process-fair run $(\tau_0, \lambda_0) \xrightarrow{\nu_1, a_1} (\tau_1, \lambda_1) \xrightarrow{\nu_2, a_2} \cdots$ with limit configuration (τ, λ).

With every lock $\ell \in H(\tau)$ we associate the maximal position $m = m_\ell$ such that $op(a_m) = \mathtt{acq}_x$ and $L(\nu_m)(x) = \ell$, so the position m_ℓ where ℓ is acquired for the last time (and never released after).

It remains to check the conditions of the lemma. The first one holds by definition of a run. The second condition is due to process-fairness and soundness, since a process can always execute transitions other than acquiring a lock, and locks not in $H(\tau)$ are free infinitely often. All actions involving $\ell \in H(\tau)$ must happen before position m_ℓ, hence there are finitely many of them. Moreover, a lock cannot be acquired and never released more than once. This shows condition F3. Conditions F4 and F5 are both satisfied because if $\ell \prec_H \ell'$ then $m_\ell < m_{\ell'}$. Thus $\prec_H$ is acyclic and it cannot have infinite descending chains.

$\Longleftarrow$ Let (τ, λ) satisfy all conditions of the lemma. In order to construct a run from (τ, λ) we first build a total order $<$ on $H(\tau)$ that extends $\prec_H$ and has no infinite descending chain. Let $\ell'_0, \ell'_1, \ldots$ be some arbitrary enumeration of $H(\tau)$ (which exists as τ is countable, thus so is $H(\tau)$). For all i let $\downarrow \ell'_i = \{\ell' \in H(\tau) \mid \ell' \prec_H^+ \ell'_i\}$. As (τ, λ) satisfies condition F3, the set of nodes that are ancestors of a node with an operation on ℓ'_i is finite. Since additionally by condition F5 there are no infinite descending chains for $\prec_H$, the set $\downarrow \ell'_i$ is finite as well (by König's lemma). As $\prec_H$ is acyclic by condition F4, we can chose some strict total order $<_i$ on $\downarrow \ell'_i$ that extends $\prec_H$. We define for all $\ell \in H(\tau)$ the index $m_\ell = \min\{i \in \mathbb{N} \mid \ell \in \downarrow \ell'_i\}$. Finally, we set $\ell < \ell'$ if either $m_\ell < m_{\ell'}$ or if $m_\ell = m_{\ell'}$ and $\ell <_{m_\ell} \ell'$. By definition $<$ is a strict total order on $H(\tau)$ with no infinite descending chains. Moreover it is easy to see that if $\ell \prec_H \ell'$ then $\ell < \ell'$. This is the case because $\ell \prec_H \ell'$ and $\ell' \prec_H^+ \ell_i$ implies $\ell \prec_H^+ \ell_i$, so $m_\ell \leq m_{\ell'}$.

Using the order $<$ on $H(\tau)$ we construct a process-fair run $(\tau_0, \lambda_0) \xrightarrow{+} (\tau_1, \lambda_1) \xrightarrow{+} \cdots$ with τ as limit configuration. During the construction we maintain the following invariant for every i:

> There exists $k_i \in \mathbb{N}$ such that all operations on locks ℓ_j with $j < k_i$ are already executed in τ_i (there is no operation on these locks in $\tau \setminus \tau_i$). Moreover, all other locks are free after executing τ_i: $H_i := H(\tau_i) = \{\ell_0, \dots, \ell_{k_i-1}\}$.

For $i = 0$ the invariant is clearly satisfied as all locks are free ($k_0 = 0$).

For $i > 0$ we assume that there is a run $\tau_0 \xrightarrow{+} \tau_i$ and τ_i satisfies the invariant. Thus, all locks ℓ_j with $j < k_i$ are ultimately held and all other locks are free in τ_i.

We say that a leaf ν of τ_i is *available* if one of the following holds:

1. either there is a descendant $\nu' \neq \nu$ on the leftmost path from ν in τ with $H(\nu') = H(\nu)$ in τ,

2. or the left child ν' of ν in τ is labelled with an unmatched `acq` of ℓ_{k_i}, and there is no further operation on ℓ_{k_i} in $\tau \setminus \tau_i$.

In particular, leaves of τ cannot be available. The strategy is to find the smallest available node ν in BFS order, and execute the actions on the left path from ν to ν'. The execution is possible as on this path there are no actions using locks from H_i and all other locks are free. Let τ_{i+1} denote the configuration thus obtained from τ_i. The invariant is satisfied after this execution, with $H_{i+1} = H_i$ in the first case above, resp. $H_{i+1} = H_i \cup \{\ell_{k_i}\}$ in the second case.

It remains to show that if a node is a leaf in τ_i for all i after some point, then it is a leaf in τ. This shows, in particular, that there always exists some available node.

Suppose that ν and i_0 are such that ν is a leaf of τ_i for all $i \geq i_0$. If ν becomes available at some point then it stays available in all future configurations, and there are finitely many nodes before ν in the BFS order. Thus ν cannot be available in some τ_i, as otherwise it would eventually be taken. Note that by the invariant (and soundness), no leaf of τ_i has the left child labelled by some `rel` operation in τ. Moreover, every leaf ν of τ_i with left child ν' in τ labelled by `nop`, `spawn`, or by some matched `acq`, is available (the latter because we consider nested DLSS). Hence, the left child of ν must be labelled with an unmatched `acq` of some $\ell \in H(\tau)$. Thus there is some unmatched `acq` on a lock of $H(\tau)$ that is never executed.

Let m be the minimal index in the enumeration of $H(\tau)$ such that an unmatched `acq` of ℓ_m in τ is never executed. By minimality of m, there exists i_1 such that $m = k_i$ for all $i \geq i_1$. After i_1, all operations on locks $\ell < \ell_m$ have been executed. Thus, as $<$ extends $\prec_H$, all unmatched `acq` operations that have some descendant in τ with operation on ℓ_m, have been executed. By the previous argument, the nodes with left child not labelled with an unmatched `acq` cannot stay leaves forever. Hence, all nodes whose left child has some operation on ℓ_m eventually become leaves. The ones with matched `acq` or other operations are then available and eventually executed.

Hence, after some point the only remaining operations on ℓ_m are unmatched `acq`. Furthermore by the condition F3 of the lemma there is exactly one. As a result, when it is reached and all other operations on ℓ_m have been executed, it becomes available, and is thus eventually executed, contradicting the definition of m.

This proves that the limit of the run we have constructed is (τ, λ). Observe finally that the run is process-fair because of condition F2 of the lemma. □

The next lemma is an important step in the proof as it simplifies condition F4 of Lemma 5.8. This condition talks about the existence of a global order on some locks.

The next lemma replaces this order with local orders in each of the nodes. These orders can be guessed by a finite automaton.

Lemma 5.9

Suppose that (τ, λ) satisfies the first three conditions of Lemma 5.8. The relation $\prec_H$ is acyclic if and only if there is a family of strict total orders $<_\nu$ over a subset of variables from $\mathit{Var}(\nu)$ such that:

F4.1 x is ordered by $<_\nu$ if and only if $L(\nu)(x) \in H(\tau)$.

F4.2 if $x <_\nu x'$, ν' is a child of ν, and $y, y' \in \mathit{Var}(\nu')$ are such that $x \sim y$ and $x' \sim y'$ then $y <_{\nu'} y'$.

F4.3 if $x, x' \in \mathit{Var}(\nu)$ and $L(\nu)(x) \prec_H L(\nu)(x')$ then $x <_\nu x'$.

Proof. $\Rightarrow$ We fix a strict total order $<$ on $H(\tau)$ that is compatible with $\prec_H$ (for instance the strict order $<$ defined in the proof of Lemma 5.8). Then we order the variables $x \in \mathit{Var}(\nu)$ with $L(\nu)(x) \in H(\tau)$ according to $<$. The three conditions of the lemma then follow directly.
$\Leftarrow$ We define $\prec$ on $H(\tau)$ by $\ell \prec \ell'$ if for some node ν with variables $x \neq x'$ such that $L(\nu)(x) = \ell$ and $L(\nu)(x') = \ell'$ we have $x <_\nu x'$.

We start by showing that $\prec$ is acyclic. Assume by contradiction that $\ell_0 \prec \ell_1 \cdots \prec \ell_k \prec \ell_0$ is a cycle of minimal length, so the locks $\ell_i \in H(\tau)$ are all distinct. We use indices modulo $k+1$, so $k+1 \equiv 0$. Note that $k > 1$ because of condition F4.2.

By assumption, the scopes of ℓ_i and ℓ_{i+1} intersect, for every i. Since scopes are subtrees of τ (Remark 5.3.1) this means that two scopes that intersect have roots that are ordered by the ancestor relation in τ.

Assume first that $k > 2$. Let i be such that the depth of the root of the scope of ℓ_i is maximal. So the roots of the scopes of ℓ_{i-1} and ℓ_{i+1} are ancestors of the root ν of the scope of ℓ_i. In the scope of ℓ_i there exist nodes that belong to the scope of ℓ_{i-1} and of ℓ_{i+1}, respectively. This means that ν is in the scope of ℓ_{i-1}, ℓ_i and ℓ_{i+1}. So the scopes of ℓ_{i-1} and ℓ_{i+1} intersect, and we have either $\ell_{i-1} <_\nu \ell_{i+1}$ or $\ell_{i+1} <_\nu \ell_{i-1}$. Thus we get from the definition of $\prec$ either $\ell_{i-1} \prec \ell_{i+1}$ or $\ell_{i+1} \prec \ell_{i-1}$. In both cases the cycle $\ell_0 \prec \ell_1 \cdots \prec \ell_k \prec \ell_0$ is not minimal, a contradiction.

It remains to consider the case $\ell_0 \prec \ell_1 \prec \ell_2 \prec \ell_0$. With a similar argument as before there exists a node ν which is in the scope of all of ℓ_0, ℓ_1, ℓ_2, so this node gives a total order on these locks and there cannot exist a cycle.

We now show that $\prec_H$ is acyclic as well.

Like before, suppose there exists a cycle of distinct nodes $\ell_0 \prec_H \ell_1 \prec_H \cdots \prec_H \ell_k \prec_H \ell_{k+1} = \ell_0$ with $k > 0$. We consider such a cycle of minimal size. Hence every ℓ_i is comparable with ℓ_{i-1}, ℓ_{i+1} and incomparable with all the other ℓ_j (as otherwise we would obtain a shorter cycle).

Given $\ell, \ell' \in H(\tau)$ such that $\ell \prec_H \ell'$, let ν be the node with an unmatched `acq` of ℓ. By condition $F3$, this node is unique. By the definition $\prec_H$ this node has some descendant ν' with an operation on ℓ'. There are two possibilities, one is that the scopes of ℓ, ℓ' intersect, in which case by condition F4.3 we have $\ell \prec \ell'$. The other possibility is that the two subtrees do not intersect, in which case the root of $\theta(\ell')$ is strictly below the unmatched get of ℓ.

As $\prec$ is acyclic, there exists some i such that the scopes of ℓ_{i-1} and ℓ_i are disjoint, hence all nodes of the scope of ℓ_i are below the unmatched `acq` of ℓ_{i-1}. In particular the unmatched `acq` of ℓ_{i-1} is an ancestor of the unmatched `acq` of ℓ_i. As a result, by the definition of $\prec_H$, $\ell_{i-1} \prec_H \ell_{i+1}$.

If $k \geq 2$ then the above argument shows that the cycle was not minimal, yielding a contradiction.

If $k = 1$ then we have a contradiction as well, as either the scopes intersect, so we cannot have both $\ell_0 \prec \ell_1$ and $\ell_1 \prec \ell_0$. Or they do not intersect, but then there is a node ν in the intersection with either $\ell_0 <_\nu \ell_1$ or $\ell_1 <_\nu \ell_0$, but not both.

As a result, the relation $\prec_H$ is acyclic. □

5.4 Recognizing limit configurations

Recall that a configuration is a possibly infinite labelled tree with a labelling that consists of five labels p, s, a, L, H. As we have mentioned in Remark 5.2.2, configurations need actually only three labels p, s, a. The other two can be calculated from the tree. Hence, in this section we consider *configurations* are labelled trees with node labels coming from a finite alphabet. Our goal in this section is to define a tree automaton recognizing limit configurations of process-fair runs of a given DLSS.

Our plan is to check the conditions (F1-5) of Lemma 5.8. Actually we will check (F1-3,5) and the conditions of Lemma 5.9 that are equivalent to F4 of Lemma 5.8.

We first observe that since our processes are finite state it is immediate to construct a nondeterministic tree automaton $\mathcal{B}_1$ verifying condition F1. This automaton just verifies local constraints between the labelling of a node and the labellings of its children. The constraints talk only about the labels p, s, a. The automaton does not need any acceptance condition, every run is accepting. We will say τ is *process-consistent* if it is accepted by $\mathcal{B}_1$.

Checking condition F2 is more complicated because it refers to a set $H(\tau)$ of locks that are ultimately held by some process. Our approach will be to define four types of predicates and colour the nodes of τ with these predicates. From a correct colouring of τ it will be easy to read out $H(\tau)$. Then we will show that the correct colouring can be characterized by conditions verifiable by finite tree automata. The colouring will be also instrumental in checking the remaining conditions F3, F4, F5.

For a configuration (τ, λ), a node ν and a variable $x \in \mathit{Var}(\nu)$ we define four predicates.

- $\nu \models \mathit{keeps}(x)$ if at ν process $p(\nu)$ holds the lock $\ell = L(\nu)(x)$ and never releases it: $\ell \in H(\nu')$ for every left descendant ν' of ν.
- $\nu \models \mathit{ev\text{-}keeps}(x)$ if $\nu \not\models \mathit{keeps}(x)$ and there is a descendant ν' of ν and a variable $x' \in \mathit{Var}(\nu')$ with $x \sim x'$ and $\nu' \models \mathit{keeps}(x')$.
- $\nu \models \mathit{avoids}(x)$ if neither $p(\nu)$ nor any descendant takes $\ell = L(\nu)(x)$, namely $\ell \notin H(\nu')$ for every descendant ν' of ν (including ν).
- $\nu \models \mathit{ev\text{-}avoids}(x)$ if $\nu \not\models \mathit{avoids}(x)$ and on every path from ν there is ν' such that $\nu' \models \mathit{avoids}(x)$.

Observe a different quantification used in *ev-keeps* and *ev-avoids*. In the first case we require one ν' to exist, in the second we want that such a ν' exists on every path.

The next lemma shows how we can use the colouring to determine $H(\tau)$.

Lemma 5.10

Let τ be a process-consistent configuration. A lock $\ell \in H(\tau)$ if and only if there is a node ν of τ and a variable $x \in \mathit{Var}(\nu)$ such that $\nu \models \mathit{keeps}(x)$ and $L(\nu)(x) = \ell$.

Proof. Follows from the definitions, since $\nu \models \mathit{keeps}(x)$ if and only if $\ell \in H(\nu')$ for every left descendant ν' of ν. □

The above conditions define a *semantically correct* colouring of nodes of a configuration τ by sets of predicates

$$\mathcal{C}(\nu) = \{P(x) : x \in \mathit{Var}(\nu), \nu \models P(x)\}$$

where $P(x)$ is one of $\mathit{keeps}(x)$, $\mathit{ev\text{-}keeps}(x)$, $\mathit{avoids}(x)$, $\mathit{ev\text{-}avoids}(x)$. Observe that the four predicates are mutually exclusive, but it may be also the case that none of them holds. We say that a variable $x \in \mathit{Var}(\nu)$ is *uncoloured* in ν if $\mathcal{C}(\nu)$ contains no predicate on x.

We now describe consistency conditions on a colouring of configurations guaranteeing that a colouring is semantically correct.

Before moving forward we introduce one piece of notation. A node that is a right child, namely a node of a form $\nu 1$ is due to $\mathtt{spawn}(q, \theta)$ operation. More precisely $\mathtt{op}(\nu 0) = \mathtt{spawn}(q, \theta)$. We refer to this θ as $\theta(\nu 1)$.

A colouring of τ is *branch-consistent* with respect to a configuration (τ, λ) if for every node ν of τ and every variable $x \in \mathit{Var}(\nu)$ the following conditions are satisfied.

- If ν has one successor $\nu 0$ then $\nu 0$ inherits the colours from ν except for two cases depending on $\mathtt{op}(\nu 0)$, i.e, the operation used to obtain $\nu 0$:
  - If $\mathit{ev\text{-}keeps}(x)$ is in $\mathcal{C}(\nu)$ and the operation is $\mathtt{acq}_x$ then $\mathcal{C}(\nu 0)$ must have either $\mathit{ev\text{-}keeps}(x)$ or $\mathit{keeps}(x)$.
  - If $\mathit{ev\text{-}avoids}(x)$ is in $\mathcal{C}(\nu)$ and the operation is $\mathtt{rel}_x$ then $\mathcal{C}(\nu 0)$ must have either $\mathit{ev\text{-}avoids}(x)$ or $\mathit{avoids}(x)$.
- If ν has two successors $\nu 0$, $\nu 1$, and there is no y with $\theta(\nu 1)(y) = x$ then $\nu 0$ inherits x colour from ν and there is no constraint due to x on colours in $\nu 1$.
- If ν has two successors and $x = \theta(\nu_1)(y)$ for some $y \in \mathit{Var}(\nu 1)$ then
  - If $\mathit{keeps}(x)$ in $\mathcal{C}(\nu)$ then $\mathit{keeps}(x)$ in $\mathcal{C}(\nu 0)$ and $\mathit{avoids}(y)$ in $\mathcal{C}(\nu 1)$.
  - If $\mathit{avoids}(x)$ in $\mathcal{C}(\nu)$ then $\mathit{avoids}(x)$ in $\mathcal{C}(\nu 0)$ and $\mathit{avoids}(y)$ in $\mathcal{C}(\nu 1)$.
  - If $\mathit{ev\text{-}keeps}(x)$ in $\mathcal{C}(\nu)$ then either
    - $\mathit{ev\text{-}keeps}(x)$ in $\mathcal{C}(\nu 0)$ and either $\mathit{avoids}(y)$ or $\mathit{ev\text{-}avoids}(y)$ in $\nu 1$, or
    - $\mathit{ev\text{-}keeps}(y)$ in $\mathcal{C}(\nu 1)$ and either $\mathit{avoids}(x)$ or $\mathit{ev\text{-}avoids}(x)$ in $\nu 0$.
  - If $\mathit{ev\text{-}avoids}(x)$ is ν then $\mathit{ev\text{-}avoids}(x)$ in $\mathcal{C}(\nu 0)$ and $\mathit{ev\text{-}avoids}(y)$ in $\mathcal{C}(\nu 1)$.

Next we describe when a colouring is *eventuality-consistent*. An *ev-trace* is a sequence of pairs $(\nu_1, x_1), (\nu_2, x_2), \ldots$ where :

- $\nu_1, \nu_2, \ldots$ is a path in τ,

- $x_i \in Var(\nu_i)$; moreover $x_{i+1} = x_i$ if ν_{i+1} is the left successor of ν_i, and $\theta(\nu_{i+1})(x_{i+1}) = x_i$ if ν_{i+1} is the right successor of ν_i.
- $\textit{ev-keeps}(x_i)$ or $\textit{ev-avoids}(x_i)$ is in $\mathcal{C}(\nu_i)$.

Observe that it follows that it cannot be the case that we have $\textit{ev-keeps}(x_i)$ and $\textit{ev-avoids}(x_{i+1})$ or vice versa. A colouring is eventuality-consistent if every ev-trace in the colouring of a configuration is finite.

Finally, a colouring is *recurrence-consistent* if for every ν and uncoloured $x \in Var(\nu)$ the lock $\ell = L(\nu)(x)$ is taken and released infinitely often below ν.

A colouring is *syntactically correct* if it is branch-consistent, eventuality-consistent, and recurrence-consistent. We show that syntactically correct colourings characterize semantically correct colourings. The two implications are stated separately as the statements are slightly different.

Lemma 5.11

If (τ, λ) is a limit configuration and $\mathcal{C}$ is a semantically correct colouring of τ then $\mathcal{C}$ is syntactically correct.

Proof. Suppose $\mathcal{C}$ is a semantically correct colouring of τ. Clearly τ is process-consistent. Branch consistency follows from Lemma 5.8 (condition F3). Indeed, all the clauses for $\textit{keeps}(x)$ and $\textit{ev-keeps}(x)$ hold because of the third condition of this lemma. The clauses for $\textit{avoids}(x)$ and $\textit{ev-avoids}(x)$ follow directly from the semantics. Directly from definition of the correct colouring it follows that it is also eventuality-consistent. It is slightly more difficult to verify that it is recurrence-consistent.

To verify recurrence consistency of $\mathcal{C}$ consider an arbitrary node ν of τ and an uncoloured variable $x \in Var(\nu)$. We find an infinite sequence:

$$(\nu, x) = (\nu_0, x_0), (\nu_1, x_1), \dots$$

such that

- $x_i \in Var(\nu_i)$ and x_i is uncoloured in ν_i,
- $\nu_0, \nu_1, \dots$ is a path,
- $x_i \sim x_{i+1}$.

Let us see why it is possible. Since every leaf satisfies either $\nu \models \textit{keeps}(x)$ or $\nu \models \textit{avoids}(x)$, node ν is not a leaf. If $\nu 0$ satisfies $\textit{keeps}(x)$ or $\textit{ev-keeps}(x)$ then so does ν. If $\nu 0$ satisfies $\textit{avoids}(x)$ or $\textit{ev-avoids}(x)$ and is the unique successor then so does ν. Hence, if $\nu 0$ is the unique successor of ν then x cannot be coloured in $\nu 0$. If $\nu 1$ exists, but there is no y with $x = \theta(\nu 1)(y)$ then the same verification shows that x cannot be coloured in $\nu 0$. Finally, if $x = \theta(\nu 1)(y)$ and y is coloured in $\nu 1$ then x must be coloured too. Hence, y is not coloured in $\nu 1$ in this last case. This shows how to find (ν_1, x_1). Repeating this argument we obtain the desired sequence.

To terminate we show why the existence of the above sequence implies the recurrence condition. First note that $x_i \sim x_j$ for all $i, j \geq 0$. Let $\ell = L(\nu)(x)$. We observe that since ν_i does not satisfy $\textit{avoids}(x_i)$ then there must be an operation on ℓ below ν_i, and since it does not satisfy $\textit{ev-keeps}(x)$ it must be a release. So we have found an infinite path such that in the subtree of every node of this path there is a release operation. This means that there are infinitely many get and release operations on ℓ in the tree below ν $\square$

For the other direction, we prove a more general statement without assuming that τ is a limit configuration. This is important as ultimately we will use the consistency properties to test if τ is a limit configuration.

Lemma 5.12

If τ is a configuration and $\mathcal{C}$ a syntactically correct colouring of τ, then $\mathcal{C}$ is semantically correct.

Proof. Process-consistency guarantees that labels follow the transition relations locally. Branch-consistency on $keeps(x)$ and $avoids(x)$ labels guarantees that if ν is labelled by one of these predicates then the predicate holds in ν. To get the same property for $ev\text{-}keeps(x)$ and $ev\text{-}avoids(x)$ we need the eventuality-consistent condition. Finally, if x is uncoloured at ν then the recurrence-consistency condition implies that x satisfies none of the four predicates. $\square$

Having a correct colouring will help us to verify all conditions of Lemma 5.8. Condition F2 refers to $L(\nu)(x) \in H(\tau)$. We need another labelling to be able to express this.

A *syntactic H-labelling* of τ assigns to every node ν a subset $H^s(\nu) \subseteq Var(\nu)$. We require the following properties:

- For the root ε we have $H^s(\varepsilon) = \emptyset$.
- If $\nu 0$ exists: $x \in H^s(\nu 0)$ if and only if $x \in H^s(\nu)$.
- If $\nu 1$ exists: $y \in H^s(\nu 1)$ if and only if either $\theta(\nu 1)(y) = \texttt{new}$ and $\nu 1 \models ev\text{-}keeps(y)$, or $\theta(\nu 1)(y) = x$ and $\nu \models ev\text{-}keeps(x)$.

It is clear that every configuration tree has a unique H^s labelling.

Lemma 5.13

Let (τ, λ) be a process-consistent configuration with a syntactically correct colouring. For every node ν and variable $x \in Var(\nu)$ we have: $L(\nu)(x) \in H(\tau)$ if and only if $x \in H^s(\nu)$.

Proof. Suppose $\ell = L(\nu)(x)$ and $\ell \in H(\tau)$. Take the node ν' that is closest to the root and has $\ell = L(\nu')(x')$ for some x'. We have $\nu' \models ev\text{-}keeps(x')$ and ν' is a right child (it cannot be the root as $Var(\varepsilon) = \emptyset$). Hence, $x' \in H^s(\nu')$. By induction on the length of the path from ν' to ν we show that $x \in H^s(\nu)$.

For the other direction, if $\nu \models ev\text{-}keeps(x)$ then $L(\nu)(x) \in H(\tau)$. It is also easy to see that membership in $H(\tau)$ is preserved by all the rules. $\square$

Thanks to Lemma 5.13 we obtain

Corollary 5.14

Let τ be a process-consistent configuration with a syntactically correct colouring. Condition F2 of Lemma 5.8 holds for τ if and only if for every leaf ν of τ, every possible transition from $s(\nu)$ has some $\texttt{acq}_x$ operation with $x \in H^s(\nu)$.

Proof. By Lemma 5.13. $\square$

Lemma 5.15

Let τ be a process-consistent configuration with a syntactically correct colouring. Then condition F3 of Lemma 5.8 holds for τ.

Proof. Consider $\ell \in H(\tau)$. By Lemma 5.10 there is a node ν and $x \in \mathit{Var}(\nu)$ with $\nu \models \mathit{keeps}(x)$, $\ell = L(\nu)(x)$. Let ν' be the root of the scope of ℓ. We have $\nu' \models \mathit{ev\text{-}keeps}(x')$ for $x' \in \mathit{Var}(\nu')$ with $x' \sim x$. By consistency conditions on the colouring:

- for every node ν'' on the path from ν' to ν we have $\nu'' \models \mathit{ev\text{-}keeps}(x'')$ for $x'' \sim x$, and for every right child ν''' of ν'' we have $\nu''' \models \mathit{ev\text{-}avoids}(x''')$ for $x''' \sim x$.

Observe that $\nu''' \models \mathit{ev\text{-}avoids}(x''')$ guarantees that there are only finitely many operations on ℓ below ν''', and that there is no unmatched get of ℓ below ν'''. Since there are no operations on ℓ below ν, we are done. ☐

It remains to deal with conditions F4 and F5 of Lemma 5.8. Condition F4 is more difficult to check as it requires to find an acyclic relation with some properties. Fortunately Lemma 5.9 gives an equivalent condition talking about a family of local orders $<_\nu$ for every node ν of a configuration. An automaton can easily guess such a family of orders. We show that it can also check the required properties.

A *consistent order labelling* assigns to every node ν of τ a total order $<_\nu$ on some subset of $\mathit{Var}(\nu)$. The assignment must satisfy the following conditions for every node ν:

1. x is ordered by $<_\nu$ if and only if $x \in H^s(\nu)$,
2. if $x <_\nu x'$ then $x <_{\nu 0} x'$,
3. if $x <_\nu x'$, $\nu 1$ exists, and $\theta(\nu 1)(y) = x$, $\theta(\nu 1)(y') = x'$ then $y <_{\nu 1} y'$,
4. if $\nu \models \mathit{keeps}(x)$ and $y <_\nu x$ then $\nu \models \mathit{keeps}(y)$ or $\nu \models \mathit{avoids}(y)$.

Lemma 5.16

Let (τ, λ) be a process-consistent configuration with a syntactically correct colouring. A family of local orders $<_\nu$ is a consistent order labelling of τ if and only if it satisfies the conditions of Lemma 5.9.

Proof. Let us take a family of orders $<_\nu$ satisfying conditions F4.1, F4.2, F4.3 of Lemma 5.9. We show that it is a consistent order labelling of τ. By Lemma 5.13 the first condition is satisfied. The next two conditions follow from condition F4.2. The fourth condition requires some verification. Consider ν as in that condition, so with $y <_\nu x$ and $\nu \models \mathit{keeps}(x)$. It follows that there is some ancestor ν' of ν, together with some $x' \sim x$, $x' \in \mathit{Var}(\nu')$, such that the action at ν' is an unmatched $\mathtt{acq}_{x'}$ of the lock $\ell = L(\nu')(x') = L(\nu)(x)$. If there were some operation on $\ell' = L(\nu)(y)$ below or at ν then $\ell \prec_H \ell'$, implying $x <_\nu y$ by F4.3. Thus there is no operation on ℓ' below or at ν, meaning that $\nu \models \mathit{keeps}(y)$ or $\nu \models \mathit{avoids}(y)$.

For the other direction, take a consistent order labelling $<_\nu$. We show that it satisfies the conditions F4.1, F4.2, F4.3 of Lemma 5.9. From the first condition on $<_\nu$ and Lemma 5.13 we see that $<_\nu$ orders only variables associated with locks from $H(\tau)$; this gives us F4.1. Condition F4.2 follows directly from the second and third property of consistent order labelling.

It remains to show F4.3. For this take a node ν and two locks $\ell = L(\nu)(x)$ and $\ell' = L(\nu)(y)$ for some $x, y \in H^s(\nu)$. Suppose $\ell \prec_H \ell'$. This means that there is an unmatched get of ℓ, say in a node ν', and an operation on ℓ' at some node ν'' below ν'.

We show below that we can find some node ν_1 in the scope of both ℓ and ℓ', and such that $\nu_1 \models \mathit{keeps}(x_1)$ and $\nu_1 \not\models \mathit{keeps}(y_1)$ and $\nu_1 \not\models \mathit{avoids}(y_1)$, with $x \sim x_1$ and $y \sim y_1$. This will show that we cannot have $y_1 <_{\nu_1} x_1$, so it must hold that $x_1 <_{\nu_1} y_1$, thus also $x <_\nu y$ by local consistency.

- If either ν, ν' are incomparable, or ν is an ancestor of ν', or $\nu = \nu'$, then ν' and $\nu'0$ are in the scope of both ℓ and ℓ' (note that $\nu'0$ is an ancestor of ν'', or they can be equal). We choose $\nu_1 = \nu'0$.
- If $\nu' \neq \nu$ is an ancestor of ν, but ν and ν'' are either incomparable, or ν is an ancestor of ν'', then we chose ν_1 as the least common ancestor of ν'' and ν. Note that ν_1 is below or equal to $\nu0$, and belongs to the scope of both ℓ and ℓ'.
- If ν'' is an ancestor of ν then ν'' is in the scope of both ℓ and ℓ', so we chose ν_1 to be ν''.

□

We consider now condition F5. We say that a consistent order labelling of τ admits an *infinite descending chain* if there exist a sequence of nodes $\nu_1, \nu_2, \ldots$ and variables $(x_i)_i, (y_i)_i$ such that for every $i > 0$: (i) ν_i is an ancestor of ν_{i+1}, (ii) $y_i \sim x_{i+1}$, and (iii) $y_i <_{\nu_i} x_i$.

Lemma 5.17

Let τ be a process-consistent configuration with a syntactically correct colouring. If $\prec_H$ has no infinite descending chain then there is a consistent order labelling of τ with no infinite descending chain. If $\prec_H$ has an infinite descending chain then every consistent order labelling of τ admits an infinite descending chain.

Proof. The first statement is easy: take the well-founded strict order on locks $<$ defined in the proof of Lemma 5.8, and for each node ν take as $<_\nu$ the order given by $<$ on $L(\nu)(\mathit{Var}(\nu))$. The well-foundedness of $<$ implies that there is no infinite descending chain in the order labelling.

For the second part, assume $\prec_H$ has an infinite descending chain and let $(<_\nu)_{\nu\in\tau}$ be a consistent order labelling of τ. Let $\ell_0 \succ_H \ell_1 \succ_H \cdots$ be an infinite descending chain for $\prec_H$.

For all $i \geq 1$ let μ_i be a node with an unmatched `acq` of ℓ_i and with a descendant with a `acq` of ℓ_{i-1}. Let c_i be the root of the scope of ℓ_i in τ. As μ_{i+1} is an ancestor of a node where ℓ_i appears, it is comparable with c_i (two nodes are *comparable* if one is an ancestor of the other). As c_{i+1} is an ancestor of μ_{i+1}, it is comparable with c_i.

Claim 5.17.1. *For all $a \leq b$, there exists $i \in \{a, \ldots, b\}$ such that c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$.*

Proof of the claim. We proceed by induction on $b - a$. If $b - a = 0$ this is clear. If $b - a > 0$, by induction hypothesis there exists $i \in \{a, \ldots, b-1\}$ such that c_i is an ancestor of all $(c_j)_{a \leq k \leq b-1}$. As c_b is comparable with c_{b-1}, which is a descendant of

c_i, c_b is comparable with c_i. If c_b is a descendant of c_i, then c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$. If c_b is an ancestor of c_i, then c_b is an ancestor of all $(c_k)_{a \leq k \leq b}$. ■

Consider the subtree of τ formed by all c_i and their ancestors. It is an infinite, but finitely-branching tree, thus it has an infinite branch by König's lemma. We first argue that there must be infinitely many c_i on that branch. Let $a \in \mathbb{N}$, let ν_a be the lowest ancestor of c_a on the branch. Let ν' be lower on the branch than ν_a, then ν' has some descendant c_b. Note that ν_a is the lowest common ancestor of c_a and c_b. We can assume $a < b$ (the case $b < a$ is symmetric), then by Claim 1 there is some $i \in \{a, \dots, b\}$ such that c_i is an ancestor of all $(c_k)_{a \leq k \leq b}$, and in particular of c_a and c_b. Further, as ν_a is the lowest common ancestor of c_a and c_b, c_i is an ancestor of ν_a and is thus on the branch. As a result, for all $a \in \mathbb{N}$ we can find $i \geq a$ such that c_i is on the branch.

We pick a sequence of c_i as follows: we start with the highest c_{i_0} on the branch, and then define $c_{i_{j+1}}$ as the highest c_i on the branch with $i > i_j$, for all j. By definition for all j we have that no c_i with $i > i_j$ is a strict ancestor of $c_{i_{j+1}}$.

As a consequence of Claim 1, there exists $i \in \{i_j + 1, \dots, i_{j+1}\}$ such that c_i is an ancestor of all $(c_k)_{i_j+1 \leq k \leq i_{j+1}}$. As noted above, as $i > i_j$ we cannot have c_i as a strict ancestor of $c_{i_{j+1}}$, hence $i = i_{j+1}$. As a result, $c_{i_{j+1}}$ is an ancestor of all $(c_k)_{i_j < k < i_{j+1}}$.

For the remaining of the proof we fix a consistent order labelling $(<_\nu)_\nu$ for τ.

Claim 5.17.2. *For all $a < b$, if node ν is in the scope of both ℓ_a and ℓ_b, and if ν is ancestor of all $(c_k)_{a<k<b}$, then $x >_\nu y$, with x, y such that $L(\nu)(x) = \ell_a$ and $L(\nu)(y) = \ell_b$.*

Proof of the claim. Let x, y be such that $L(\nu)(x) = \ell_a$ and $L(\nu)(y) = \ell_b$. We proceed by induction on $b - a$.
If $b = a + 1$ then $\ell_a \succ_H \ell_b$. Since we assume that $(<_\nu)_n$ is a consistent order labelling, by Lemma 5.16 and F4.3 of Lemma 5.9 we have $x >_\nu y$ as claimed.
If $b - a \geq 2$, by Claim 1, there exists $i \in \{a+1, \dots, b-1\}$ such that c_i is an ancestor of all $(c_k)_{a<k<b}$. In particular, c_i is an ancestor of c_{a+1}, itself an ancestor of μ_{a+1}, itself an ancestor of a node ν' with a `acq` of ℓ_a. Recall that ν itself is an ancestor of c_i, by assumption. As the scope of a lock is a subtree, ℓ_a appears in all nodes between ν and ν', thus in particular in c_i.
Moreover, μ_b is an ancestor of some node with a `acq` of ℓ_{b-1}, which is a descendant of c_{b-1}, thus of c_i, hence μ_b and c_i are comparable. If μ_b is an ancestor of c_i, then as ν' is a descendant of c_i, ν' is also a descendant of μ_b, hence $\ell_a \succ_H \ell_b$. As a result, $x >_\nu y$ as the consistent order labelling satisfies F4.3 (Lemma 5.16). If μ_b is a descendant of c_i then as the scope of a lock is a subtree, ℓ_b appears in all nodes between ν and μ_b, thus in particular in c_i. We set $x', y', z' \in \mathit{Var}(c_i)$ such that $L(c_i)(x') = \ell_a$, $L(c_i)(y') = \ell_b$ and $L(c_i)(z') = \ell_i$ and by induction hypothesis we have $x >_{c_i} z$ and $z >_{c_i} y$ thus $x >_{c_i} y$ as $>_{c_i}$ is total. Finally, as we have a consistent order labelling, $x >_\nu y$ holds as well. ■

Recall that $c_{i_{j+1}}$ is an ancestor of all $(c_k)_{i_j<k<i_{j+1}}$. In particular, $c_{i_{j+1}}$ is an ancestor of c_{i_j+1}, thus of μ_{i_j+1}, itself an ancestor of some node ν' with a `acq` of ℓ_{i_j}. Thus ℓ_{i_j} appears in $c_{i_{j+1}}$, because $c_{i_{j+1}}$ is between c_{i_j} and ν'.

Let x_j, y_j be such that $L(c_{i_{j+1}})(x_j) = \ell_{i_j}$ and $L(c_{i_{j+1}})(y_j) = \ell_{i_{j+1}}$. By Claim 2, we have $x_j >_\nu y_j$. The sequences $(c_{i_{j+1}})_{j>0}$, $(x_j)_{j>0}$ and $(y_j)_{j>0}$ thus form an infinite descending chain, proving the lemma. □

The next proposition summarizes the development of this section stating that all the relevant properties can be checked by a Büchi tree automaton.

Proposition 5.18

For a given DLSS, there is a non-deterministic Büchi tree automaton $\widehat{\mathcal{B}}$ accepting exactly the limit configurations of process-fair runs of DLSS. The size of $\widehat{\mathcal{B}}$ is linear in the size of the DLSS and exponential in the maximal arity of the DLSS.

Proof. Given a tree τ labelled with p, a, s the automaton $\widehat{\mathcal{B}}$ guesses a colouring $\mathcal{C}$, labelling H^s and an ordering labelling $\mathcal{O}$. It then checks if $\mathcal{C}$, H^s and $\mathcal{O}$ satisfy all the consistency conditions. This automaton is a product of the following automata:

- $\mathcal{B}_1$ recognizing process-consistent trees,
- $\mathcal{B}_{\mathcal{C}}$ checking if the colouring is syntactically correct,
- $\mathcal{B}_H$ checking if H^s is a syntactic H-labelling,
- $\mathcal{B}_2$ checking the conditions of Corollary 5.14,
- $\mathcal{B}_{\mathcal{O}}$ checking if $\mathcal{O}$ is a consistent order labelling.
- $\mathcal{B}_5$ checking the absence of infinite descending chains (Lemma 5.17).

Apart from $\mathcal{B}_{\mathcal{C}}$ and $\mathcal{B}_5$ the other automata only check relations between a node and its children and some additional conditions local to a node. So they are automata with trivial acceptance conditions. Automaton $\mathcal{B}_{\mathcal{C}}$ needs a Büchi condition to check that $\mathcal{C}$ is eventuality-consistent and recurrence-consistent. The number of labels is polynomial in the size of DLSS and exponential in the maximal arity as we have sets of predicates and orderings on variables as labels. Automaton $\mathcal{B}_5$ can be obtained by first constructing an automaton for its complement: one can easily define a non-deterministic Büchi automaton guessing a branch and following a sequence of variables along that branch witnessing an infinite decreasing sequence of locks. As it only needs to remember a pointer to one of the variables of a node, its number of states is the maximal arity of the DLSS. Thus we can complement it to get a non-deterministic Büchi automaton checking the absence of such sequence, of size exponential in the maximal arity of the DLSS, and polynomial in the alphabet (itself exponential in the arity and polynomial in the DLSS).

We need to check that τ is a fair limit configuration if and only if it is accepted by $\widehat{\mathcal{B}}$.

If τ is a limit configuration then it is process consistent, so it is accepted by $\mathcal{B}_1$. Guessing $\mathcal{C}$ to be semantically correct colouring ensures that $\mathcal{B}_{\mathcal{C}}$ accepts τ with this colouring (Lemma 5.11). As we have observed, given the colouring there is unique syntactic H-labelling, so $\mathcal{B}_H$ can accept it. By Lemma 5.8, configuration τ satisfies properties F1-5. So τ is accepted by $\mathcal{B}_2$. Finally, by Lemma 5.9, τ satisfies properties F4.1, F4.2, F4.3, so τ is accepted by $\mathcal{B}_{\mathcal{O}}$ thanks to Lemma 5.16. By Lemma 5.17, the automaton $\mathcal{B}_5$ accepts τ as well.

For the other direction suppose τ is accepted by $\widehat{\mathcal{B}}$. Thanks to Lemma 5.8 it is sufficient to check properties F1-5. Property F1 is verified by automaton $\mathcal{B}_1$. Thanks to $\mathcal{B}_{\mathcal{C}}$ we know that the guessed colouring is syntactically correct. Then $\mathcal{B}_2$ ensures that τ satisfies F2 thanks to Corollary 5.14. Lemma 5.15 ensures that τ satisfies F3. Finally, automaton $\mathcal{B}_{\mathcal{O}}$ checks that the guessed orderings are a consistent order labelling. Hence, Lemma 5.16 guarantees that τ satisfies the conditions of Lemma 5.9 giving us F4. Finally, by Lemma 5.17 automaton $\mathcal{B}_5$ verifies condition F5. □

We will show that the previous proposition yields an EXPTIME algorithm. We match it with an EXPTIME lower bound to obtain completeness. The hardness proof involves a reduction from the problem of determining whether the intersection of the languages of k deterministic tree automata over binary trees is empty. To achieve this, we create a DLSS that simulates all the tree automata concurrently. Each node of the tree in the intersection is simulated by a process, which encodes a state for each automaton through the locks it holds. So each process creates two children with whom it shares locks. The children are able to access the states of the parent by the following technique: Suppose processes p and q share locks 0 and 1, and p acquires one lock and retains it indefinitely. In this scenario, q can guess the lock chosen by p and try to acquire the other lock. If q guesses incorrectly, the system deadlocks. However, if the guess is correct, the execution continues, and q knows about the lock held by p.

Proposition 5.19

The DLSS verification problem for nested DLSS and Büchi objective is EXPTIME-hard. The result holds even if the Büchi objective refers to a single process.

Proof sketch. A detailed proof can be found in [**MascleMW23**]. We provide a proof sketch here.

We show that the difficulty of the problem stems from the systems and not the specification, by proving that checking if some process has an infinite run is already EXPTIME-hard.

We give a reduction from the emptiness problem for the intersection of top-down deterministic tree automata (over finite trees). This problem is EXPTIME-complete [**Goubault94**]. Let $\mathcal{A}_1, \dots, \mathcal{A}_k$ be finite deterministic tree automata over an alphabet A, with $\mathcal{A}_i = (Q_i, \delta_i, s_{0,i}, F_i)$. We are going to construct a DLSS that simulates their computations simultaneously on the same tree T, by using locks to memorize their states. We use three processes, p_{root}, p_{left} and p_{right}.

The idea is to have a new process copy for each node of T. Each such copy uses the variables x_i^q, y_i^q and z_i^q for all $1 \leq i \leq k$ and $q \in Q_i$. Variable x_i^q is supposed to encode the information about the state of the parent node in the run of $\mathcal{A}_i$, while y_i^q and z_i^q will encode the state of $\mathcal{A}_i$ at the current node. The y_i^q are meant to give that information to the left child and z_i^q to the right child.

Each process does the following steps. First for each i it chooses some s and takes all $x_i^{q'}$ with $q' \neq q$. Then it chooses a letter $a \in A$. Let $\delta(q, a) = (q', q'')$, the process takes $y_i^{q'}$ and $z_i^{q'}$ if it is p_{left} and $y_i^{q''}$ and $z_i^{q''}$ if it is p_{right}.

After doing this for each i, it can either just stop or spawn two processes p_{left} and p_{right}. It passes to them the locks corresponding to respectively y_i^q and z_i^q as their x_i^q and fresh locks for their y_i^q and z_i^q.

The specification is that there should be finitely many processes, and all processes that do not spawn other processes should be in a final state of all $\mathcal{A}_i$. This is only possible if the state q chosen by each process matches the state q' chosen by their parent process. If $q \neq q'$, the child process will be blocked while trying to get $x_i^{q'}$. From the states and letters chosen by the processes we can reconstruct a full labelled binary tree accepted by all $\mathcal{A}_i$.

If we want a sound system, we have to modify it: as it is, processes would have to remember a state s for each i, and thus have exponentially many states. This can be fixed: instead of taking a lock, processes spawn an auxiliary process which takes that lock and stops. We add in the specification that those auxiliary processes should all take their lock.

The DLSS we constructed is clearly nested, which shows the claim. □

Now we have all ingredients for the proof of Theorem 5.5:

Proof of Theorem 5.5. The lower bound follows from Proposition 5.19.

For the upper bound we use the Büchi tree automaton $\hat{\mathcal{B}}$ recognizing limit configurations of the DLSS (Proposition 5.18).

We build the product of $\hat{\mathcal{B}}$ with the regular objective automaton $\mathcal{A}$, which is a parity tree automaton. From $\hat{\mathcal{B}} \times \mathcal{A}$ we can obtain with a bit more work an equivalent parity tree automaton $\mathcal{C}$ with the same number of priorities, plus one. For this we modify the rank function in order to only store in the state the maximal priority seen between two consecutive occurrences of Büchi accepting states, and make the maximal priority visible at the next Büchi state. When the state of the $\hat{\mathcal{B}}$ component is not a Büchi state, the priority is odd and lower than all the ones of $\mathcal{A}$.

By Proposition 5.18, $\mathcal{C}$ is non-empty if and only if there exists a limit configuration of the system that satisfies the regular objective $\mathcal{A}$. Moreover, we know that $\hat{\mathcal{B}}$ has size linear in the size of the DLSS and exponential only in arity of the DLSS. So $\mathcal{C}$ has size that is exponential w.r.t. the DLSS and the objective, and polynomial size if the maximal arity is fixed.

Finally, non-emptiness of $\mathcal{C}$ amounts to solve a parity game of the same size as $\mathcal{C}$: player Automaton chooses transitions of $\mathcal{C}$, and player Pathfinder chooses the direction (left/right child). To sum up, we obtain a parity game of exponential size, so solving the game takes exponential time since the number of priorities is polynomial. If both the number of priorities and the arity of the DLSS is fixed, the game can be solved in polynomial time. □

5.5 Pushdown systems with locks

Till now in this chapter every process has been a finite state system. Here we consider the case when processes can be pushdown automata. The definition of a *pushdown DLSS* is the same as before but now each automaton $\mathcal{A}_p$ is a deterministic pushdown automaton.

We will reduce our verification problem to the emptiness test of a nondeterministic pushdown automata on infinite trees. These automata will have parity acceptance conditions. While in general testing emptiness of such automata is EXPTIME-complete, we will notice that the automata we construct have a special form allowing to test emptiness in PTIME for a fixed number of ranks in the parity condition.

We start by defining *pushdown tree automata*. Our automaton will be quite standard but for an additional stack instruction. Apart standard $\texttt{pop}(a)$ and $\texttt{push}(a)$, we have a $\texttt{reset}$ instruction that empties the stack. A pushdown tree automaton is a tuple $(Q, \Sigma, \Gamma, \Delta, init, F, \texttt{pr})$, where Q is a finite set of states, Σ an input alphabet, Γ a stack alphabet, $q_{init} \in Q$ an initial state, F a set of final states and $\texttt{pr} : Q \to [0, d]$ a parity condition. Finally, $\mathcal{D}$ is a partial transition function taking as the arguments the current state q, the current input letter a, and the current stack symbol γ. The transitions in $\mathcal{D}$ are of the form either $\delta(q, a, \gamma) = (q', \texttt{instr})$ or $\delta(q, a, \gamma) = ((q_l, \texttt{instr}_l), (q_r, \texttt{instr}_r))$, where $\texttt{instr}, \texttt{instr}_l, \texttt{instr}_r$ are stack instructions.

A run of such an automaton on a Σ-labelled tree is an assignment of configurations to nodes of the tree; each configuration has the form (q, z) where $q \in Q$ is a state and $z \in \Gamma^+$ is a sequence of stack symbols representing the stack (top symbol being the leftmost). The root is labelled with (q_{init}, ε). The labelling of children must depend on the labelling of the parent according to the transition function δ. A run is accepting if every leaf is labelled with a state of F and for every infinite branch the sequence of assigned states satisfies the max parity condition given by $\mathtt{pr}$: the maximum of ranks of states seen on the path must be even.

We say that a pushdown tree automaton is *right-resetting* if for every transition $\delta(q, a, \gamma) = ((q_l, \mathtt{instr}_l), (q_r, \mathtt{instr}_r))$ we have that $\mathtt{instr}_r$ is $\mathtt{reset}$.

Proposition 5.20

For a fixed d, the emptiness problem for right-resetting pushdown tree automata $\mathcal{A}$ with a parity condition over ranks $\{1, \ldots, d\}$ can be solved in PTIME.

Proof. We consider the representative case of $d = 3$. Suppose we are given a right-resetting pushdown tree automaton $\mathcal{A} = (Q, \Sigma, \Gamma, q^0, \bot, \delta, \mathcal{W})$.

The first step is to construct a pushdown word automaton $\mathcal{A}^l(G_1, G_2, G_3)$ depending on three sets of states $G_1, G_2, G_3 \subseteq Q$. The idea is that $\mathcal{A}^l$ simulates the run of $\mathcal{A}$ on the leftmost branch of a tree. When $\mathcal{A}$ has a transition going both to the left and to the right then $\mathcal{A}^l$ goes to the left and checks if the state going to the right is in an appropriate G_i. The states of $\mathcal{A}^l(G_1, G_2, G_3)$ are $Q \times \{1, 2, 3\}$ with the second component storing the maximal rank of a state seen so far on the run. The transitions of $\mathcal{A}^l(G_1, G_2, G_3)$ are defined according to the above description. We make precise only the case for a transition of $\mathcal{A}$ of the form $\delta(q, a, \gamma) = ((q_l, \mathtt{instr}_l), (q_r, \mathtt{instr}_r))$. In this case, $\mathcal{A}^l$ has a transition $\delta^l((q, i), a, \gamma) = ((q_l, \max(i, \mathcal{W}(q_l))), \mathtt{instr}_l)$ if $q_r \in G_{\max(i, \mathcal{W}(q_r))}$. Observe that $\mathtt{instr}_r$ is necessarily $\mathtt{reset}$ as $\mathcal{A}$ is right-resetting.

What the fixpoint computation does can be described at high-level as follows. While the word pushdown automaton $\mathcal{A}^l$ takes care of the parity condition on tree paths that are ultimately left paths, the sets G_i do this for paths that branch to the right infinitely often. For such paths we need for example to guarantee through set G_3 that priority 3 is seen finitely often. We do this through a least fixpoint computation for G_3. For G_2 we compute a greatest fixpoint since we want priority 2 to be seen infinitely often. Finally, for G_1 we compute a least fixpoint since priority 1 should be seen finitely often before seeing priority 2.

The next step is to observe that for given sets G_1, G_2, G_3 we can calculate in PTIME the set of states from which $\mathcal{A}^l(G_1, G_2, G_3)$ has an accepting run.

The last step is to compute the following fixpoint expression in the lattice of subsets of Q:

$$\begin{aligned} W =& \,\mathsf{LFP} X_3.\ \mathsf{GFP} X_2.\ \mathsf{LFP} X_1.\ P(X_1, X_2, X_3) \qquad \text{where} \\ P(X_1, X_2, X_3) =& \{q : \mathcal{A}^l(X_1, X_2, X_3) \text{ has an accepting run from } q\}\ . \end{aligned}$$

Observe that $P : \mathcal{P}(Q)^3 \to \mathcal{P}(Q)$ is a monotone function over the lattice of subsets of Q. Computing W requires at most $|Q|^3$ computations of P for different triples of sets of states.

We claim that $\mathcal{A}$ has an accepting run from a state q, if and only if, $q \in W$.

Let us look at the right-to-left direction of the claim. For this we recall how the least fixpoint is calculated. Consider any monotone function $R(X)$ over $\mathcal{P}(Q)$, and its least fixpoint $R^\omega = \mathsf{LFP}X.\ R(X)$. This fixpoint can be computed by a sequence of approximations:

$$R^0 = \emptyset \qquad R^{i+1} = R(R^i)$$

The sequence of R^i is increasing and $R^\omega = R^i$ for some $i \leq |Q|$.

Now we come back to our set W. Observe that $W = \mathsf{LFP}X_3.\ R$ where $R(X) = \mathsf{GFP}X_2.\mathsf{LFP}X_1.\ P(X_1, X_2, X)$. As in the previous paragraph we can define

$$W^0 = \emptyset \qquad \text{and} \qquad W^{i+1} = \mathsf{GFP}X_2.\mathsf{LFP}X_1.\ P(X_1, X_2, W^i)\ .$$

So, if $q \in W$ then $q \in W^i$ for some i. Now observe that $W^i = \mathsf{LFP}X_1.P(X_1, W^i, W^{i-1})$, since W^i is a fixpoint of $\mathsf{GFP}X_2$. By similar reasoning we define

$$W^{i,0} = \emptyset \qquad \text{and} \qquad W^{i,j+1} = P(W^{i,j}, W^i, W^{i-1})\ .$$

Now, $q \in W^i$ implies $q \in W^{i,j}$ for some j. We write $sig(q)$ for the lexicographically smallest (i,j) such that $q \in W^{i,j}$.

We examine what $sig(q) = (i,j)$ means. By definition $q \in P(W^{i,j-1}, W^i, W^{i-1})$, so there is an accepting run of $\mathcal{A}^l(W^{i,j-1}, W^i, W^{i,j-1})$ from q. Looking at the run of $\mathcal{A}$ that $\mathcal{A}^l$ simulates we can see that whenever this run branches to the right with some q' and e is the maximal rank on the run till this branching then

- if $e = 1$ then $q' \in W^{i,j-1}$,
- if $e = 2$ then $q' \in W^{i,k}$ for some k,
- if $e = 3$ then $q' \in W^{i-1,k}$ for some k.

With this observation we can construct an accepting run of $\mathcal{A}$ from every state in W. If $sig(q) = (i,j)$ then consider an accepting run of $\mathcal{A}^l$ on the left path given by $P(W^{i,j-1}, W^i, W^{i-1})$. For every state branching to the right from this left path we recursively apply the same procedure. By construction, every path that is eventually a left path is accepting. A path branching right infinitely often is also accepting by the previous paragraph since signatures cannot go below 0. More precisely, the path cannot see 3 infinitely often because the first component of the signature decreases. If it sees 1 infinitely often then it needs to see also 2 infinitely often, because of the second component that decreases.

Let us now look at the left-to-right direction. Take an accepting run of $\mathcal{A}$ from q^0. We construct something that we call a skeleton tree of this run. As the nodes of the skeleton tree we take the root and all the nodes that are a right child; so these are the nodes of the tree of the form $(0^*1)^*$. The skeleton has an edge $\nu \xrightarrow{e} \nu 0^k 1$ if e is the maximal rank of a state of $\mathcal{A}$ on the path from ν to $\nu 0^k 1$. Observe that a node can have infinitely many children. As we have started with an accepting run, every path in this skeleton tree satisfies the parity condition. In particular, for every node, on every path from this node there is a finite number of 3 edges. Thus, to every node ν we can assign an ordinal $\theta^3(\nu)$ such that if $\nu \xrightarrow{e} \nu'$ then $\theta^3(\nu') \leq \theta^3(\nu)$ and the inequality is strict if $e = 3$. It is also the case that on every path from ν there is a finite number of 1 edges before some 2 or 3 edge. This allows to define $\theta^1(\nu)$ with the property that if $\nu \xrightarrow{1} \nu'$ then $\theta^1(\nu') < \theta^1(\nu)$. Now we can show that for every node ν of the skeleton tree, if q is the state assigned to ν then $q \in W^{\theta^3(\nu),\theta^1(\nu)}$ for $W^{i,j}$ as defined in the computation of W (putting $W^{\theta^3} = W$ for every $\theta^3 > |Q|$, and $W^{\theta^3,\theta^1} = W^{\theta^3}$ for every $\theta^1 > |Q|$). The proof is by induction on the lexicographic order on $(\theta^3(\nu), \theta^1(\nu))$. □

Proof of Theorem 5.6. The lower bound follows already from Theorem 5.5.

For the upper bound we reuse the Büchi tree automaton $\hat{\mathcal{B}}$ from Proposition 5.18. This time $\hat{\mathcal{B}}$ is a pushdown tree automaton, however it is right-resetting because processes are spawned with empty stack. We follow the lines of the proof of Theorem 5.5, building the product of $\hat{\mathcal{B}}$ with the regular objective automaton $\mathcal{A}$, and constructing an equivalent parity, right-resetting pushdown tree automaton $\mathcal{C}$. Proposition 5.20 concludes the proof. □

5.6 Synthesis of DLSS

In this section we leverage our understanding of the verification of DLSS to study the synthesis problem on them. We introduce *dynamic lock-sharing games*, which are DLSS in which states are split between controllable and uncontrollable ones.

Definition 5.21

A *dynamic lock-sharing game* (DLSG) is described by a tuple $(\mathcal{S}, (S_p^C, S_p^E)_{p\in\mathsf{Proc}}, \mathcal{B})$, where $\mathcal{S} = (\mathsf{Proc}, ar, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, p_{init}, \mathcal{L}ocks)$ is a DLSS, $\mathcal{B}$ is a parity tree automaton and for all $p \in \mathsf{Proc}$, $S_p^C \sqcup S_p^E$ is a partition of the states of $\mathcal{A}_p$.
Just like in the previous chapter, we assume that all actions from Controller states have operation $\mathtt{nop}$.

See Remark 4.7.1 for an explanation for the last assumption. We call a DLSG nested when the underlying DLSS is.

A *control strategy* is a family of functions $\sigma = (\sigma_p)_{p\in\mathsf{Proc}}$ with $\sigma_p : \Sigma_p^* \to \Sigma_p$ for all p. A *σ-run* is a run $\tau_0 \xrightarrow{\nu_1,a_1} \tau_1 \xrightarrow{\nu_2,a_2} \cdots$ where for all $i \geq 1$, if $st(\nu_i) \in \mathcal{S}^C_{pr(\nu_i)}$ then $a_i = \sigma_{pr(\nu_i)}(a_1 \cdots a_{i-1})$. A control strategy is *winning* if no process-fair σ-run is in $\mathcal{L}(\mathcal{B})$. Note the negation here: we assume that $\mathcal{L}(\mathcal{B})$ describes a set of runs that we want to *avoid*.

We can now define the problem, which is to decide if there exists a winning control strategy for a given DLSG.

Definition 5.22 ▶ *Regular control problem for DLSG*

Given a DLSG $(\mathcal{S}, (S_p^C, S_p^E)_{p\in\mathsf{Proc}}, \mathcal{B})$, decide if there is a winning control strategy.

The undecidability of that problem immediately follows from Theorem 5.3. Our main results of this section are the following complexity bounds on that problem in the case of nested DLSG.

Theorem 5.23

The regular control problem for DLSG is in 2EXPTIME and NEXPTIME-hard over nested DLSG.

We will comment on the gap between the bounds at the end of the section. For now let us simply remark that a deterministic procedure for a NExpTime-hard problem may also take double-exponential time. We start by showing the lower bound.

Proposition 5.24

The regular control problem for DLSG is NExpTime-hard over nested DLSG.

Proof sketch. We reduce from the exponential grid tiling problem. The idea is as follows:

The initial process p_{init} has $8n$ locks $\ell_1^\alpha, \dots, \ell_n^\alpha$ for each $\alpha \in \{x, y, x', y'\}$, plus $2|T|$ locks ℓ_t, ℓ'_t for each tile.

Environment encodes coordinates (x, y) and (x', y') in the locks: he takes ℓ_i^x if he wants the ith bit of x to be one, and $\bar{\ell}_i^x$ otherwise. Doing so, he can choose those coordinates one of three ways:

- either $x' = x$ and $y' = y$
- or $x' = x + 1$ and $y' = y$
- or $y' = y + 1$ and $x' = x$

He can make sure of that using $O(n)$ states by choosing x, x' and y, y' simultaneously.

Note that this system is not sound: there are exponentially many possible sets of locks taken in n, much more than the number of states. However, we can avoid this by replacing the single process p_{init} with a chain of processes, each choosing one bit and spawning the next (and carrying the necessary information by choosing which next process to spawn).

After the coordinates have been chosen, two processes p, p' are spawned. In p, Environment starts by taking and releasing one of $\ell_i^x, \overline{\ell^x}_i$ and one of $\ell_i^y, \overline{\ell^y}_i$, before going to a state controlled by Controller. Controller must then choose a tile t, and take the associated lock ℓ_t. The actions taken by Environment just before allow her to know x and y. The specification forbids that Controller stays stuck there: she must take the lock she picked.

Similarly, Controller picks a lock $\ell'_{t'}$ in p'.

The specification is that Environment wins if either:

- $x = x'$, $y = y'$ and $t \neq t'$.
- $x' = x + 1$, $y = y'$ and $t.right \neq t'.left$
- $x' = x + 1$, $y = y'$ and $t.up \neq t'.down$
- $x = 0$ (resp. $x = 2^n - 1$, $y = 0$, $y = 2^n - 1$) and $t.left \neq B$ (resp. right, down, up).

This condition can be expressed as a parity tree automaton of polynomial size: The automaton guesses one of the above conditions and checks them on the branches corresponding to p_{init}, p and p'. For instance, the automaton may guess that we are in the second case, and guess different values for $t.right$ and $t'.left$. It then checks that those values are indeed the tiles selected in p and p', and that the choice made in p_{init} matches the second case.

The strategy of Controller comes down to two tilings of the exponential grid. The specification forces both tilings to be the same, and to be locally consistent.

Hence a winning strategy for Controller induces a valid tiling. Conversely a valid tiling gives Controller a winning strategy. The problem is therefore NEXPTIME-hard. □

For the upper bound, we rely on our characterisation of runs in a DLSS in terms of limit configurations. In all that follows we fix a nested DLSG $(\mathcal{S}, (S_p^C, S_p^E)_{p \in \mathsf{Proc}}, \mathcal{B})$, where we have

$$\mathcal{S} = (\mathsf{Proc}, ar, (\mathcal{A}_p)_{p \in \mathsf{Proc}}, p_{init}, \mathcal{L}ocks),$$

$$\mathcal{A}_p = (S_p, \Sigma_p, \delta_p, init_p) \text{ for all } p, \text{ and}$$

$$\mathcal{B} = (Q, \Sigma, \Delta, q_{init}, F, \mathtt{pr})$$

.

In order to characterise winning strategies we will need the notions of local runs, which are left branches of configurations. We will also define labelled local runs, which are local runs annotated with states of $\mathcal{B}$. They will allow us to reduce the regular control problem for DLSG to a usual two-player game.

An *agent* in a tree (τ, λ) is a node that is either the root or the right child of another node. We call them agents as they are the starts of left branches in the tree, so we identify them with agents executing local runs.

A *local run* on process p is a sequence of the form

$$u_p = (s_0, B_0, a_0) \longrightarrow_p (s_1, B_1, a_1) \longrightarrow_p \cdots$$

so that $s_0, s_1, \ldots \in S_p$, $a_0, a_1, a_2, \ldots \in \Sigma_p$, $\delta_p(s_i, a_{i+1}) = s_{i+1}$, for all i and $(s_0, B_0) = (init_p, \emptyset)$. Furthermore it should be consistent with lock acquisitions, i.e., for all i:

- If $op(a_{i+1}) = \mathtt{spawn}(q, \theta)$ then $B_{i+1} = B_i$.
- If $op(a_{i+1}) = \mathtt{acq}_x$ then $x \notin B_i$ and $B_{i+1} = B_i \cup \{x\}$.
- If $op(a_{i+1}) = \mathtt{rel}_x$ then $x \in B_i$ and $B_{i+1} = B_i \setminus \{x\}$.

A *local run* of a configuration (τ, λ) is the local run defined by the left branch from an agent in τ. Formally, it is a local run of the form $u_p = (s(\nu), H(\nu), a(\nu)) \longrightarrow_{p(\nu)} (s(\nu 0), H(\nu 0), a(\nu 0)) \longrightarrow_{p(\nu)} \cdots$ with ν an agent.

It is a *σ-local run* if for all i such that $s_i \in S_p^C$, $a_{i+1} = \sigma(a_1 \cdots a_i)$.

We also define a *labelled local run* on process p as a local run annotated with states of the objective automaton $\mathcal{B}$: it is a sequence of the form $\bar{u}_p = (s_0, B_0, q_0, a_0) \xrightarrow{d_1}_p (s_1, B_1, q_1, a_1) \xrightarrow{d_2}_p \cdots$ so that $(s_0, B_0, a_0) \longrightarrow_p (s_1, B_1, a_1) \longrightarrow_p \cdots$ is a local run on p, $q_0, q_1, \ldots \in Q$, and for all i, $d_i \in \Delta$ and either $\mathtt{op}(a_i)$ is not a spawn $d_i = (q_{i-1}, a_{i-1}, q_i)$ or it is a spawn and there exists q such that $d_i = (q_{i-1}, a_{i-1}, q_i, q)$.

We say that $\bar{u}_p$ is *accepting* if either it is finite and the last state q_i is in F, or it is infinite and $\min(\{pr(q) \mid \forall i, \exists j > i, q = q_j\})$ is even.

Now that we have a notion of local run, the next step is to characterise winning control strategies. To begin with, we need to adapt Lemma 5.8 a little to characterise trees that are limit configurations of σ-runs.

Corollary 5.25

Let σ be a control strategy. A tree τ is the limit configuration of a process-fair σ-run if and only if all the following conditions hold

F1 The node labels in τ match the local transitions of the system, and for all ν that is a right child or the root, the left branch starting from ν is a σ-local run.

F2 For every leaf ν every possible transition from $s(\nu)$ has operation $\mathtt{acq}_x$ for some x with $L(\nu)(x) \in H(\tau)$.

F3 For every lock $\ell \in H(\tau)$ there are finitely many nodes with operations on ℓ, and there is a unique node labelled with an unmatched $\mathtt{acq}$ of ℓ.

F4 The relation $\prec_H$ is acyclic.

F5 The relation $\prec_H$ has no infinite descending chain.

Proof. For each process p, we can obtain an infinite transition system by applying the strategy σ in p to choose actions from states of S_p^C. We can then apply Lemma 5.8 to the resulting DLSS and we get this corollary.

Item F2 simply translates item F2 using the assumption that all actions from Controller states have operation $\mathtt{nop}$ (by definition of DLSG). □

We once again go through a definition of patterns, which are summaries of local runs of bounded length. We will show that we can determine whether a set of local runs can be combined to form a limit configuration in $\mathcal{L}(\mathcal{B})$ from the set of their patterns.

Definition 5.26

We define $\Sigma_{\mathtt{spawn}} = \{a \in \Sigma \mid \exists p', \theta, \mathtt{op}(a) = \mathtt{spawn}(p', \theta)\}$. We call an *annotated spawn* of p an element of $Q \times \Sigma_{\mathtt{spawn}} \times \mathtt{pr}(Q)$. Let AS_p be the set of annotated spawns.

An *extended pattern* is a tuple $(p, q, SP_0T_0x_1SP_1T_1 \cdots x_kSP_kT_k, IT)$ where

- $p \in \mathsf{Proc}$
- $q \in Q$
- $x_1 \cdots x_k$ is a sequence of distinct lock variables of $\mathit{Var}(p)$
- For each i, $SP_i : AS_p \to [0, ar(p) + 1] \cup \{+\infty\}$
- $IT \subseteq \mathit{Var}(p)$

Those patterns describe the sequence of unmatched $\mathtt{acq}$ happening in a labelled local run, and the numbers of spawn operations made between each pair of consecutive events. Spawn operations are annotated with the maximal priority seen so far and the current state of $\mathcal{B}$. We count the number of occurrences of each spawn operation between two events up to $ar + 1$. We also have a set of lock variables IT, representing the set of locks taken infinitely many times.

We now formally define the extended pattern associated with a labelled local run. Let $u_p = (s_0, B_0, a_0, q_0) \xrightarrow{d_1}_p (s_1, B_1, a_1, q_1) \xrightarrow{d_2}_p \cdots$ be a finite or infinite labelled local run.

- Let $\{i_1, \dots, i_k\}$ be the set of indices i such that $\mathtt{op}(a_i) = \mathtt{acq}_x$ and $\mathtt{op}(a_{i'}) \neq \mathtt{rel}_x$ for all $i' > i$. For each $j \in [1, k]$, we define x_j so that $\mathtt{op}(a_{i_j}) = \mathtt{acq}_{x_j}$. Those are the indices corresponding to the unmatched $\mathtt{acq}$.
- Then for each $j \in [0, k-1]$, we define $T_j = \{x \in \mathit{Var}(p) \mid \exists i \in [i_j + 1, i_{j+1} - 1], \mathtt{op}(a_i) = \mathtt{acq}_x\}$. Those are the sets of locks acquired temporarily between two unmatched $\mathtt{acq}$.
- We also set, for each $j \in [0, k-1]$, $q \in Q$, $a \in \Sigma_{\mathtt{spawn}}$ and $pr \in \mathtt{pr}(Q)$,

  $I_j(q, a, pr) = \{i \in [i_j, i_{j+1}-1] \mid d_i = (q_{i-1}, a_{i-1}, q_i, q), a_i = a, pr = \max(\{\mathtt{pr}(q_{i'}) \mid i' < i\})\}$.

  We then set $SP_j(q, a, pr) = \min(|I_j|, ar(p) + 1)$.
- For $j = k$ we do the same but take into account that there may be infinitely many spawns. Let

  $$I_k(q, a, pr) = \{i > i_k \mid q_{i-1} = q, a_i = a, pr = \max(\{\mathtt{pr}(q_{i'}) \mid i' < i\})\}.$$

  We then set $SP_k(q, a, pr)$ as $\min(|I_k|, ar(p) + 1)$ if $I_k(s, q, a, pr)$ is finite and $+\infty$ otherwise.
- Let $IT = \{x \in \mathit{Var}(p) \mid \forall i, \exists i' > i, \mathtt{op}(a_{i'}) = \mathtt{acq}_x\}$.

The extended pattern of u_p is $\pi(u_p) = (p, q_0, SP_0 x_1 SP_1 \cdots x_k SP_k, IT)$.

Given a (non-labelled) local run u_p, a state $q \in Q$ and an extended pattern π, we say that π is an extended pattern of u_p if there is an accepting labelled local run whose sequence of transitions is u_p and whose extended pattern is π.

We now go through a sequence of definitions that resemble the one from Section 4.3.3 in the previous chapter. A finite local run u is *risky* if it ends in a state where all outgoing transitions acquire a lock. If u is risky then we define $\mathrm{B}(u)$ as the set of locks that can be acquired from its last state.

An *extended blocking pattern* is a pair (π, B) where π is an extended pattern and $B \in \mathit{Var}(p)$ for some $p \in \mathsf{Proc}$. Given a risky local run u, we say that (π, B) is a extended blocking pattern if π is a extended pattern of u, u is finite and risky and $\mathrm{B}(u) = B$.

We now show that given a strategy σ, the extended patterns of σ-local runs by a control strategy determine whether it is winning or not.

Given a control strategy σ, we write $\Pi_b(\sigma)$ for the set of extended blocking patterns of finite risky accepting labelled σ-local runs $\bar{u}$. We write $\Pi_\infty(\sigma)$ for the set of extended patterns of accepting infinite labelled σ-local runs $\bar{u}$. We call the pair $(\Pi_b(\sigma), \Pi_\infty(\sigma))$ the *behaviour* of σ.

We now prove a lemma justifying these definitions. It states that whether a strategy is winning depends only on its behaviour.

Lemma 5.27

Let $(\tau, \lambda) \in \mathcal{L}(\mathcal{B})$ be a limit configuration accepted by $\mathcal{B}$. Let Π_b be the set of extended blocking patterns of risky local runs of (τ, λ), and Π_∞ the set of extended patterns of its infinite local runs.
Then for all control strategy σ' such that $\Pi(\sigma') \subseteq \Pi$ and $\Pi_\infty(\sigma') \subseteq \Pi_\infty$, σ' is losing.

Proof. As τ is in $\mathcal{L}(\mathcal{B})$, we can annotate it τ with the states of an accepting run of $\mathcal{B}$. In what follows we will consider that every leftmost branch from an agent in τ is a labelled local run.

We construct a tree (τ', λ') from (τ, λ) using the σ'-local runs and show that is in $\mathcal{L}(\mathcal{B})$ and that it is the limit of a σ'-run. We will write pr', st', act', L', H' for the labelling functions of (τ', λ').

A node of a labelled tree is called *important* if it is either

- a right child of its parent
- or the root
- or labelled with an unmatched `acq`.

We write $\mathbf{IN}(\tau)$ for the set of important nodes of τ.

We present this construction in the form of an algorithm, that constructs this tree "layer by layer". While we construct (τ', λ'), we also define a mapping $\beta : \mathbf{IN}(\tau') \to \mathbf{IN}(\tau)$, which maps each important node in τ' to a node in τ that it mimics.

We start with a tree τ' that is just the root, which we map to the root of τ.

We apply the following step indefinitely:

- First we pick an agent ν' in τ' that is of minimal depth among the ones that have not been treated yet. The minimal depth condition is there to make sure that every agent is eventually treated.
- Then we consider the labelled local run $\bar{u}$ that is the left branch rooted in $\beta(\nu')$. Let p be the corresponding process. There exists an accepting labelled σ-local run $\bar{u}'$ whose extended pattern is $\pi(u)$ and such that if u is finite then so is u' and $\mathrm{B}(u) = \mathrm{B}(u')$. We add u' to τ' as a left branch rooted in ν'.
- As $\pi(u) = \pi(u')$, they have unmatched `acq` on the same lock variables in the same order. For each node ν with an unmatched `acq` in this branch of τ' we define $\beta(\nu)$ as the corresponding unmatched `acq` in τ.
- Then, for each interval between two unmatched `acq`, we give a right sibling to each node with a spawn operation. We label them as follows. Let $q \in Q$, $a \in \Sigma_{\texttt{spawn}}$ and $pr \in \texttt{pr}(Q)$. Between each pair of unmatched `acq`, u and u' have the same number of spawns corresponding to those parameters, or both have more than ar. In the first case, we define β on those children in τ' as a one-to-one matching with the corresponding right children in τ.

  In the second case, let L be the set of locks that are used in u and passed to the children. Those are the same locks for all those children, as they are spawned from the same process using the same action a. Let C be the set of those children in τ which have an unmatched `acq` of a lock of L in their descendants. As $|L| \leq ar(p)$, we can pick one of the children in τ' for each element of C and make β map it to that element. Then we pick a child in τ that is not in C and map the rest of the children in τ' to it.

  After the last unmatched `acq`, we have three cases. Either u and u' have the same number of spawns corresponding to those parameters, or both have more than ar but finitely many, or both have infinitely many. We handle the first two cases as before. For the third case, we pick a one-to-one mapping from the right children in τ' to those in τ.

We start by showing that every branch of τ' either ends with an accepting state of $\mathcal{B}$ or is such that the minimal priority seen infinitely often is even.

First of all, every left branch is accepting, by construction. As a consequence, every finite branch ends with an accepting state of $\mathcal{B}$. It remains to consider branches that go to the right infinitely many times. Consider such a branch, let $\nu_0, \nu_1, \nu_2 \dots$ be the sequence of right children along that branch. By construction, in τ we have an infinite branch that goes infinitely many times to the right and whose sequence of right children is $\beta(\nu_0), \beta(\nu_1), \dots$. By definition of β, the maximal priority seen between $\beta(\nu_i)$ and $\beta(\nu_{i+1})$ is the same as the maximal priority between ν_i and ν_{i+1}. As a consequence, the maximal priority seen infinitely often is the same in the two branches. Since we labelled τ with an accepting run of $\mathcal{B}$, we conclude that this priority is even.

As we build τ' using only σ'-local runs, it suffices to show that τ is a limit configuration to show that it is the limit configuration of a σ'-run and thus prove the lemma.

Claim 5.27.1. *The construction satisfies the following properties:*

(1) For all $\nu' \in \mathbf{IN}(\tau')$, $pr'(\nu') = pr(\beta(\nu'))$

(2) For all $\nu' \in \mathbf{IN}(\tau')$, ν is labelled with an unmatched `acq` *of x if and only if $\beta(\nu')$ is.*

(3) For all $\nu_1, \nu_2 \in \mathbf{IN}(\tau')$, if $\beta(\nu_1)$ is a strict ancestor of $\beta(\nu_2)$ then ν_1 is a strict ancestor of ν_2.

(4) For all $\nu_1, \nu_2 \in \mathbf{IN}(\tau')$, and $x_1 \in Var(\nu_1)$ and $x_2 \in Var(\nu_2)$, if $x_1, \nu_1 \sim x_2, \nu_2$ then $x_1, \beta(\nu_1) \sim x_2, \beta(\nu_2)$.

(5) For all $\nu' \in \mathbf{IN}(\tau')$ and $x \in Var(\nu')$, $L(\nu')(x) \in H'(\tau')$ if and only if $L(\beta(\nu'))(x) \in H(\tau)$.

(6) For all $\nu_1, \nu_2 \in \mathbf{IN}(\tau')$, and $x_1 \in Var(\nu_1)$ and $x_2 \in Var(\nu_2)$, if $L(\nu_1)(x_1) \prec_H L(\nu_2)(x_2)$ then $L(\beta(\nu_1))(x_1) \prec_H L(\beta(\nu_2))(x_2)$.

(7) For all $\nu' \in \tau'$, ν' and $\beta(\nu')$ have the same number of ancestors that are right children.

(8) For all $\nu'_1 \neq \nu'_2 \in \tau'$, if $\beta(\nu'_1) = \beta(\nu'_2)$ then let ν' be their closest common ancestor in τ', for all $x \in Var(\nu')$, there is no unmatched `acq` *of any y in a descendant ν'' of ν' such that $\nu', x \sim \nu'', y$.*

(9) For all $\nu' \in \mathbf{IN}(\tau)$, $\beta^{-1}(\nu')$ is finite

Proof of the claim.

- Items (1) to (3) are immediate from the construction.
- Item (4) results from a straightforward induction on the number of steps of the algorithm.
- For item (5), we prove the two directions:
  - If $L(\nu')(x) \in H'(\tau')$ then there is a node $\mu' \in \tau'$ and $y \in Var(\nu')$ with $\mu', y \sim \nu', x$ and μ' is labelled with an unmatched $\texttt{acq}_y$. By item (4) we have

$\beta(\mu'), y \sim \beta(\nu'), x$ and by item (2) $\beta(\mu')$ is labelled with an unmatched $\mathtt{acq}_y$, hence $L(\beta(\nu'))(x) \in H(\tau)$.

– Suppose $L(\beta(\nu'))(x) \in H(\tau)$. Let $\nu'_1, \dots, \nu'_k$ be the right children on the branch from the root to ν'. By construction, $\beta(\nu'_1), \dots, \beta(\nu'_k)$ are the right children on the branch from the root of τ to $\beta(\nu')$. Let i be the smallest index such that $\beta(\nu'_i), y \sim \beta(\nu'), x$. Let $\mu_1, \dots, \mu_r$ be the sequence of right children between $\beta(\nu'_i)$ and the node μ with the unmatched acq of $L(\beta(\nu'))(x)$. Note that they all use lock $L(\beta(\nu'))(x)$, with a descendant that has an unmatched acq of it. In the algorithm defined above, we make sure that all important nodes in τ with a descendant who has an unmatched acq have a predecessor by β. A simple induction shows that there is a sequence of nodes $\mu'_1, \dots, \mu'_r, \mu'$ such that $\beta(\mu'_j) = \mu_j$ for all j and $\beta(\mu') = \mu$. Let z be the variable such that μ is labelled with $\mathtt{acq}_z$. As $\beta(\nu'), x \sim \mu, z$ and $\mu = \beta(\mu')$, by items (2) and (4) we have $L(\nu'), x \in H'(\tau')$.

- For (6), let $\ell'_1 = L'(\nu_1)(x_1)$ and $\ell'_2 = L'(\nu_2)(x_2)$, suppose $\ell'_1 \prec_H \ell'_2$. There exist $\mu'_1, \mu'_2 \in \tau'$ such that μ'_1 has an unmatched acq of ℓ'_1 and μ'_2 has an acq of ℓ'_2 and μ'_1 is an ancestor of μ'_2. Let μ'' be the closest ancestor of μ'_2 that is a right child.

  – If μ'' is an ancestor of μ'_1 then the left branch from μ'' contains an unmatched acq of x and later an acq of y, with $L'(\mu'')(x) = \ell'_1$ and $L'(\mu'')(y) = \ell'_2$. Hence the left branch from $\beta(\mu'')$ also contains an unmatched acq of x and later an acq of y, as the two have the same extended pattern. Therefore $L(\beta(\mu''))(x) \prec_H L(\beta(\mu''))(y)$. Using item (4) we obtain $L(\nu_1)(x_1) \prec_H L(\nu_2)(x_2)$.

  – If μ'' is a descendant of μ'_1 then $\beta(\mu'')$ is a descendant of $\beta(\mu'_1)$. Furthermore as the left branches from μ'' and $\beta(\mu'')$ have the same extended pattern, the one from $\beta(\mu'')$ contains an operation $\mathtt{acq}_y$ with $L'(\mu'')(y) = \ell'_2$.

    Let x be the variable such that $\beta(\mu'_1)$ is labelled $\mathtt{acq}_x$, then we have $L(\beta(\mu'_1))(x) \prec_H L(\beta(\mu'')(y))$. Using item (4) we again obtain $L(\nu_1)(x_1) \prec_H L(\nu_2)(x_2)$.

- Item (7) is immediate from the construction.

- For (8), let μ'_1, μ'_2 be the first right children on the branches from ν' to ν'_1 and ν'_2 respectively. They must exist as two nodes on the same left branch cannot have the same image and (7) implies that there are as many right nodes from ν' to ν'_1 and ν'_2.

  Then $\beta(\mu'_1)$ and $\beta(\mu'_2)$ must both be ancestors of $\beta(\nu'_1) = \beta(\nu'_2)$, thus they are comparable for the ancestor relation. As ν' is the closest ancestor of ν'_1 and ν'_2, μ'_1 and μ'_2 must be incomparable, thus by (3) we have $\beta(\mu'_1) = \beta(\mu'_2)$. As μ'_1 and μ'_2 must be added to τ' on the same step of the algorithm, the construction guarantees that $\beta(\mu'_1)$ and $\beta(\mu'_2)$ have no descendant with unmatched acq of a lock used by $\beta(\nu')$. By (2) and (4), this means that μ'_1 and μ'_2 have no descendant with unmatched acq of a lock used by ν', which implies the same property for ν'_1 and ν'_2.

- Item (9) follows by induction on the depth of ν'. It clearly holds for the root. Let ν' be a node, let μ' be its closest strict ancestor that is a right node or the root. By induction hypothesis $\beta^{-1}(\mu')$ is finite. We only add preimages to ν' when we treat leaves that are in $\beta^{-1}(\mu')$, and we can only add finitely many ones each time. Hence $\beta^{-1}(\nu')$ is finite.

■

In the rest of the proof, we show that (τ', λ') satisfies all conditions in Lemma 5.25. As (τ, λ) is the limit configuration of a σ-run, it satisfies those conditions.

- F1 is straightforwardly satisfied: we built τ' while respecting the transitions of the DLSG.
- For F2, consider a leaf ν of τ'. Let ν' be its closest ancestor that is an agent: ν must be a left descendant of ν'. By construction, $\beta(\nu')$ has a left descendant in τ that is a leaf labelled with the same state s as ν. Hence every operation from s does an $\mathtt{acq}_x$ with $L(\nu)(x) \in H(\tau)$, and thus by (5) $L'(\nu')(x) \in H'(\tau')$.
- For F3, let $\ell' \in H'(\tau')$. There is a node ν' with an unmatched $\mathtt{acq}$ of x, with $L'(\nu')(x) = \ell'$. Then by (2), the node $\nu = \beta(\nu')$ also has an unmatched $\mathtt{acq}$ of x. Therefore we have a lock $\ell \in H(\tau)$ with $L(\nu)(x) = \ell$. Let N' be the set of agents μ' of τ' which have a $\mathtt{acq}_y$ operation in a left descendant and with $x, \nu' {\sim} y, \mu'$. For all $\mu' \in N'$, the left branch from $\beta(\mu')$ and μ' have the same extended patterns. Hence all $\mu' \in N'$ have finitely many left descendants with operations on ℓ'. Thus all $\mu \in \beta(N')$ have a left descendant with a $\mathtt{acq}_y$ operation and are such that $x, \nu {\sim} y, \mu$. Hence $\beta(N')$ must be finite, and as $\beta^{-1}(\nu''')$ is finite for all ν''' (by (9)), N' must be finite. As a result, there are finitely many operations on ℓ' in τ'.

  We now show that there cannot be ν'_1, ν'_2 with both unmatched $\mathtt{acq}$ of ℓ. Suppose it is the case, let x_1, x_2 be the variables acquired in ν'_1 and ν'_2. Then we have $\nu'_1, x_1 {\sim} \nu'_2, x_2$, and $\beta(\nu'_1), x_1 {\sim} \beta(\nu'_2), x_2$ by (4). Assume that $\beta(\nu'_1) \neq \beta(\nu'_2)$, then we obtain a contradiction as τ would not satisfy F3. On the other hand, if $\beta(\nu'_1) = \beta(\nu'_2)$, then by (8) we get a contradiction.
- By (6), if $\prec_H$ has a cycle or an infinite descending chain then so does $\prec_{H'}$. Thus F4 and F5 hold.

We have verified all the conditions. □

As a consequence of the previous lemma, we can use behaviours as invariants to solve distributed synthesis, as in the previous chapter. Furthermore, the size of a behaviour is only exponential in $ar(\mathcal{S})$ and $|Q|$ (and linear in Proc). They can thus be enumerated in double-exponential time.

Remark 5.6.1. *The number of extended patterns is at most*

$$\varphi(\mathcal{S}, \mathcal{B}) = |\mathsf{Proc}||Q|(ar(\mathcal{S})(|Q| \cdot |\Sigma_{\mathtt{spawn}}| \cdot |\mathtt{pr}(Q)|)^{ar(\mathcal{S})+2} 2^{ar(\mathcal{S})})^{ar(\mathcal{S})+1} 2^{ar(\mathcal{S})}.$$

This formula is obtained directly from the definition. The number of extended blocking patterns can be assumed to be at most that bound times $2^{ar(\mathcal{S})}$.

It remains to show that we can decide, given a behaviour, if there is a strategy

To do so, we show that we can construct a non-deterministic Muller automaton recognising the set of local runs whose extended patterns are in a given set. The size of the automaton is exponential in the size of the DLSG but only polynomial in the size of Π. We then determinise this automaton and use it to check both conditions of the previous lemma.

Lemma 5.28

Let Π_b be a set of extended blocking patterns. For all $p \in \mathsf{Proc}$ there is a DFA with $\sum_{p\in\mathsf{Proc}} |S_p| 2^{|Q|\varphi(\mathcal{S},\mathcal{B})}$ states recognising finite local runs of p with an extended blocking pattern in Π_b.

Proof. We start by building a DFA which reads labelled local runs, checks consistency and keeps track of their extended patterns and the last state of Q they visit. This information is easy to update. Its number of states is exponential in $|Q|$ and $ar(\mathcal{S})$.

By removing the states of Q from the input, we obtain an NFA reading local runs, guessing an execution of $\mathcal{B}$ on it and keeping track of the extended pattern and the last state of Q reached by the resulting labelled local run. We can then determinise it to get a DFA reading a local run u and keeping track of the set of extended patterns of labelled local runs whose underlying local run is u, and the subset of accepting ones.

Finally, it suffices to additionally keep track of the state reached in S_p and restrict the final states to the ones that contain an extended pattern π of an accepting labelled local run such that $(\pi, B) \in \Pi_b$ with B the set of locks that can be taken from the last state of S_p visited, assuming all outgoing transitions take a lock. $\square$

Remark 5.6.2. *Importantly, in the following construction, we consider local runs simply as sequences of actions without restricting them to the ones that can be produced by $\mathcal{S}$. This ensures that the size of the resulting automaton does not depend on the number of states of the processes but only on their arity and the number of states of $\mathcal{B}$.*

Lemma 5.29

Let Π_∞ be a set of extended patterns, let $p \in \mathsf{Proc}$ and $q \in Q$.
There is a deterministic Muller automaton with $\varphi(\mathcal{S},\mathcal{B})$ states, and with $\sum_{p\in\mathsf{Proc}} 2ar(\mathcal{S})^2 + |AS|(ar(p)+2)ar(p)$ colours recognising infinite accepting labelled local runs which have an extended pattern that is in Π_∞.

Proof. We consider labelled local runs as finite or infinite words over the finite alphabet $\bigcup_{p\in\mathsf{Proc}}(Q\Sigma)$.

It is easy to check that the sequence of states of Q is a run of $\mathcal{B}$ by simply keeping track of the current state reached. We will ignore this part in the rest of the proof. The fact that the labelled local run is accepting is a parity condition, easily encoded as a Muller condition.

We now build an automaton checking whether a labelled local run has an extended pattern that is not in Π.

States are of the form $SP_0T_0x_1SP_1T_1 \cdots x_kSP_kT_k$. The x_i are lock variables that were acquired and not released so far, in the order in which they were last acquired. The T_i are the sets of locks acquired and released between the last acquisitions of the x_i. The SP_i count the number of each annotated spawn between the last acquisitions of the x_i, up to $ar(p)+1$. This information is easy to keep track of, and the number of states is less than the number of extended patterns.

We have colours of the form $(x, i) \in \mathit{Var}(p) \times [1, |\mathit{Var}(p)|]$, $(as, j, i) \in AS \times [0, ar(p)+1] \times [1, \mathit{Var}(p)]$ and $\overline{(y, i)} \in \mathit{Var}(p) \times [1, |\mathit{Var}(p)|]$. We colour the states with the corresponding colours as follows:

- Colour (x, i) indicates that $x = x_i$
- Colour (as, j, i) indicates that $SP_i(as) = j$
- Colour $\overline{(y, i)}$ indicates that $y \in T_i$

It is then easy to check that each set of colours seen infinitely many times determines the extended pattern of the labelled local run. We can thus express the fact that this extended pattern is in Π as a Muller condition. It suffices to take the conjunction of that Muller condition with the parity condition expressing that the labelled local run is accepting to get the result. ☐

Lemma 5.30

Let Π_∞ be a set of extended patterns. There is a deterministic parity automaton with $2^{2^{(|\mathsf{Proc}|+|Q|+ar(\mathcal{S})+|\Sigma_{\mathsf{spawn}}|)^{O(1)}}}$ states and with $2^{(|\mathsf{Proc}|+|Q|+ar(\mathcal{S})+|\Sigma_{\mathsf{spawn}}|)^{O(1)}}$ many priorities recognising infinite local runs which have an extended pattern $\pi \in \Pi_\infty$.

Proof. Consider the Muller automaton obtained in Lemma 5.29. Let N be its number of states and K its number of colours.

We can obtain a non-deterministic Muller automaton recognising infinite local runs which have an extended pattern $\pi \notin \Pi$ by additionally guessing a sequence of transitions of $\mathcal{B}$ to turn the input local run into a labelled local run. Doing so, we only multiply the number of states by $|Q|$.

We can then apply Proposition 2.3 to turn the resulting automaton into a non-deterministic parity automaton, with $M = N|Q| \cdot K \cdot K!$ states and K priorities.

It then suffices to apply Proposition 2.4 to turn the resulting automaton into a deterministic parity automaton, with $2M^M(K+1)^{M(K+1)}(M(K+1))!$ states and using $2M(K+1)$ priorities. A little computation shows the lemma. ☐

Now that we have a deterministic parity automaton, we can use it to decide a two player game characterising the existence of a strategy satisfying a behaviour.

Lemma 5.31

Let Π_b be a set of extended blocking patterns and Π_∞ a set of extended patterns. We can check double-exponential time whether there is a control strategy σ such that $\Pi_b(\sigma) \subseteq \Pi_b$ and $\Pi_\infty(\sigma) \subseteq \Pi_\infty$.

Proof. Suppose there is a winning control strategy σ. Let $p \in \mathsf{Proc}$, consider the following two-player game: First Environment picks a process $p \in \mathsf{Proc}$. Then we play on the transition system of p, each player picks the next transition from their state. The objective for Environment is that either:

- at some point the local run u constructed so far has an extended blocking pattern that is not in Π_b.

- the play is infinite and the local run u constructed has an extended pattern that is not in Π_∞.

By Lemma 5.30, this objective is recognised by a deterministic parity automaton with double-exponentially many states and exponentially many priorities in the size of the system.

We can solve this game in double-exponential time by Proposition 2.9. □

We can then infer the following result.

Proposition 5.32

The regular control problem for DLSG is in 2EXPTIME over nested DLSG.

Proof. It suffices to enumerate behaviours, and for each one (Π_b, Π_∞), check that there is a strategy σ such that $\Pi_b(\sigma) \subseteq \Pi_b$ and $\Pi_\infty(\sigma) \subseteq \Pi_\infty$, and that there is a limit configuration (τ, λ) accepted by $\mathcal{B}$ and such that $\Pi_b(\tau, \lambda) \subseteq \Pi_b$ and $\Pi_\infty(\tau, \lambda) \subseteq \Pi_\infty$.

If the first answer is yes and the other one is no, then the strategy computed is winning. If this is never the case, then every strategy is losing by Lemma 5.27.

The existence of the limit configuration (τ, λ) can be checked using a product of $\mathcal{B}$ with the automata constructed in Proposition 5.18, Lemma 5.28 and Lemma 5.29. We obtain a Muller tree automaton with exponentially many colours and double-exponentially many states, whose emptiness can be checked in double-exponential time.

By Lemma 5.31, the existence of the strategy can also be checked in double-exponential time. As a consequence, we have a 2EXPTIME algorithm to check if Controller wins a DLSG. □

Finally, let us comment on the results obtained in this section. The complexity of our algorithm arises mostly from the determinisation step in 5.30. Everything else can be done in non-deterministic exponential time: in particular we can guess sets of extended patterns $(\Pi_a)_{a\in\Sigma}$ and Π_∞, since they have at most exponential size.

If the regular objective is given by a deterministic automaton, then we do not need the determinisation step and we end up with an automaton of exponential size only. As a consequence, when $\mathcal{B}$ is deterministic the problem is in NEXPTIME.

Furthermore, the size of $\Sigma_{\texttt{spawn}}$ can be bounded by $|\mathsf{Proc}|^2 ar(\mathcal{S})!$. Indeed, we can replace each spawn transition by two consecutive transitions, one doing `nop` and the other the spawn. This ensures that every spawn transition is the only transition from its source state. We can then have a unique letter for each $\texttt{spawn}(p', \Sigma)$ while keeping the system deterministic. Thus, if we fix both $ar(\mathcal{S})$ and $\mathcal{B}$, as well as the number of process types, we obtain an NP algorithm.

In particular, deadlock objectives like “A process is blocked forever”, or “Infinitely many processes get blocked” or “Every process ends up in a deadlock” can be expressed with tree automata of constant size. Thus checking if there is a control strategy against one of those objectives can be done in non-deterministic polynomial time when the arity and the number of process types are fixed.

5.7 Towards integrating variables

In this section we study the addition of shared variables in the model at hand. We show several undecidability results, but also highlight a research direction that may lead to positive results.

A *dynamic lock-sharing system with variables* (DLSSV for short) is defined similarly to a DLSS:

$$\mathcal{S} = (\mathsf{Proc}, ar, (\mathcal{A}_p)_{p\in\mathsf{Proc}}, p_{init}, \mathcal{L}ocks, \Gamma, \alpha_{init})$$

Each transition system $\mathcal{A}_p$ is again a tuple $(S_p, \Sigma_p, \delta_p, \mathsf{op}_p, init_p)$.

The only two differences are: a finite alphabet Γ, which is the set of values taken by the shared variable, and the set of available operations $\mathsf{Op}(p)$ which is extended with $\{\mathbf{rd}(\alpha), \mathbf{wr}(\alpha) \mid \alpha \in \Gamma\}$.

Configurations are configurations of DLSS, extended with a letter of Γ, indicating the current value of the shared variable: $(\tau, \lambda), \alpha$ with (τ, λ) a configuration of $\mathcal{S}$ and $\alpha \in \Gamma$. There is a *step* $(\tau_1, \lambda_1), \alpha_1 \xrightarrow{\nu,a} (\tau_2, \lambda_2), \alpha_2$ if either

- $\mathsf{op}(a)$ is $\mathtt{nop}$, a spawn or a lock operation and $(\tau_1, \lambda_1) \xrightarrow{\nu,a} (\tau_2, \lambda_2)$
- or $(\tau_1, \lambda_1) = (\tau_2, \lambda_2)$ and $\mathsf{op}(a) = \mathbf{rd}(\alpha_1)$ and $\alpha_2 = \alpha_1$,
- or $(\tau_1, \lambda_1) = (\tau_2, \lambda_2)$ and $\mathbf{act}(a) = \mathbf{wr}(\alpha_2)$

A *run* is, as usual, a sequence of steps $(\tau_0, \lambda_0), \alpha_0 \xrightarrow{\nu_1,a_1} (\tau_1, \lambda_1), \alpha_1 \xrightarrow{\nu_2,a_2} \cdots$. It is *initial* if $(\tau_0$ is just the root ε with $\lambda_0(\varepsilon) = (p_{init}, init_{p_{init}}, \bot, \emptyset, \emptyset)$ and $\alpha_0 = \alpha_{init}$.

This extension of the system with a shared variable turns out to yield an undecidable state reachability problem (given a DLSSV and a state, is there a reachable configuration in which a node is labelled with that state?).

Theorem 5.33

The state reachability problem for DLSSV is undecidable.

Proof. We reduce from the emptiness of intersection of two context-free grammars. Let $\mathcal{G}_1, \mathcal{G}_2$ be two context-free grammars over a common alphabet A. We can assume that in both grammars every rule is of the form $X \to Yb$, $X \to aY$ or $X \to x$.

We start by defining a system simulating a single grammar, and then extend it to a system in which two grammars are simulated in parallel, while interleaving their runs to make sure that they produce the same word.

Let $\mathcal{G}$ be a context-free grammar of that form. We construct a DLSSV such that the set of sequences of letters of A written by runs reaching a designated state q_f is exactly $L(\mathcal{G})$.

For each rule $X \to \alpha$ of $\mathcal{G}$ we have a process $p_{X\to\alpha} \in \mathsf{Proc}$, of arity 2. Process $p_{X\to\alpha}(\ell_1, \ell_2)$ starts by taking locks ℓ_1 and ℓ_2. It then goes through α from left to right as follows. When it encounters a terminal a, it writes a on the shared variable. When it encounters a non-terminal Y, it picks a rule $Y \to \beta$, writes *start*, releases ℓ_2, spawns $p_{Y\to\beta}(\ell_2, new)$, reads *done* and takes back ℓ_2.

After going through the whole rule, it writes *done*, releases ℓ_2, then ℓ_1, and stops.

Figure 5.3 displays an example.

We add an initial process $p_{init}(\ell_1)$ that picks a rule $I \to \alpha$, writes *start*, spawns $p_{I\to\alpha}(\ell, new)$, reads *done*, takes ℓ and releases ℓ, then goes to q_f.

We call a leaf *active* if it holds all its locks.

We show both directions in two separate claims, that are formulated in a way that makes the inductive proof more direct. The first one states that we only produce words

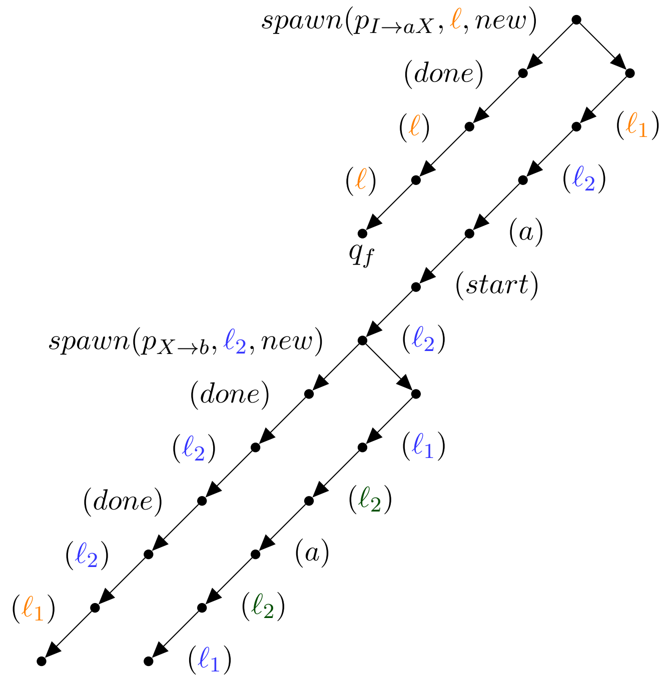


Figure 5.3: A run of a DLSSV $\mathcal{S}$ as in the proof of Theorem 5.33, simulating the rules $I \to aX$, $X \to b$.

that are in the language of the grammar, and that there is at most one active leaf at all times.

Claim 5.33.1. *Let τ be a configuration with a leaf ν such that $st(\nu) = q_{init}^{X\to\alpha}$, with X a non-terminal of $\mathcal{G}$, and locks of ν are both free. Consider a non-empty run $\varrho : (\tau, \lambda), start \xrightarrow{*} (\tau', \lambda'), \alpha$ with τ' a tree obtained from τ by replacing ν with a subtree, and so that the first lock of ν is free. Then $\alpha = done$, all descendant processes of ν in τ' have reached the end of their local run, and the sequence of letters of A written during the run is a word $w \in L(X)$.*

Further, at all times of ϱ there is at most one active node in the descendants of ν.

Proof of the claim. Suppose α is of the form aY, the cases Yb and x are similar. The first steps of ϱ must consist in a left branch from ν that takes ℓ_1 and ℓ_2, writes a, then writes *start*, releases ℓ_2 and spawns a node $\bar{\nu}$ labelled $p_{Y\to\beta}(\ell_2, new)$ for some rule $Y \to \beta$.
We are then in a configuration $\bar{\tau}, start$. Consider the run section from $\bar{\tau}, start$ to the configuration just before the process takes back ℓ_2. During that part of the run, it read *done* and did nothing else. Furthermore it is not active, as it is not holding ℓ_2. This means that, by removing this read, we obtain a run $\bar{\tau}, start$ to $\bar{\tau}', \bar{\alpha}$ where the only actions taken are by descendants of $\bar{\nu}$.
We can apply the lemma inductively on this part of the run. We obtain that $\bar{\alpha} = done$ and the sequence of writes of letters of A made during that part of the run is a word $\bar{w} \in L(Y)$. Also, all processes in the tree rooted in $\bar{\nu}$ have finished their local run, and there was at most one active leaf below $\bar{\nu}$ during that part of the run.
Then the first process takes ℓ_2, then writes *done*, releases ℓ_2, and then ℓ_1 and stops.
We must therefore have $\alpha = done$, and the sequence of letters of A written during the run is $w = a\bar{w}$, which is in $L(X)$. All processes below ν have stopped, and at all times

there were at most one active process. ■

The second claim shows that every word of the language of the grammar can be produced by a run of our DLSSV.

Claim 5.33.2. *Let τ be a limit configuration with a leaf ν such that $st(\nu) = q_{init}^{X}$, with X a non-terminal of $\mathcal{G}$ and locks of ν are both free.*

Let $w \in L(X)$, there is a run $(\tau, \lambda) start \xrightarrow{*} (\tau', \lambda') done$ such that locks of ν are both free at the end, τ' is obtained from τ by replacing ν with a subtree whose leaves are all labelled by final states, and the sequence of letters of A written during the run is w.

Proof of the claim. Let $X \to aY$ be a rule such that $w = av$ with $v \in L(Y)$. We build a run as follows.
From ν we make the process take ℓ_1 and ℓ_2, write a, then write *start*, release ℓ_2 and spawn a node $\bar{\nu}$ labelled $p_Y(\ell_2, new)$.
We are then in a configuration $\bar{\tau}, start$. We apply the induction to get a run to $\bar{\tau}', done$ during which v has been written, and only descendants of $\bar{\nu}$ have been created. Further, ℓ_2 is free.
Then the first process takes ℓ_2, then writes *done*, releases ℓ_2, and then ℓ_1 and stops.
We have a run $(\tau, \lambda) start \xrightarrow{*} (\tau', \lambda') done$, and the sequence of letters of A written during the run is indeed $w = av$. Moreover, ℓ_1 and ℓ_2 are free. ■

As a consequence of those claims, we obtain that:

- The sequences of letters of A that can be written by runs of this DLSSV are exactly the words of $L(\mathcal{G})$.
- At all times of the run there is at most one active leaf. As processes only write while holding all of their locks. this implies that at most one process is able to write on the shared variable at all times.

We construct the final system by applying this construction for both grammars $\mathcal{G}_1$ and $\mathcal{G}_2$. We obtain two DLSSV $\mathcal{S}_1$ and $\mathcal{S}_2$.

We then modify both systems as follows: Let A_1 and A_2 be two disjoint copies of A. We write a_1 and a_2 for the copies of a in A_1 and A_2.

We replace every $\mathbf{wr}(a)$ in $\mathcal{S}_1$ by two actions $\mathbf{rd}(a_2)\mathbf{wr}(a_1)$, while in $\mathcal{S}_2$ we replace them with $\mathbf{wr}(a_2)\mathbf{rd}(a_1)$. We also rename the letters *start* and *done* in $\mathcal{S}_1$ (resp. $\mathcal{S}_2$) by $start_1$ and $done_1$ (resp. $start_2$ and $done_2$).

The final system is obtained by making an initial process spawn the initial processes of those two systems with distinct locks.

The specification is simply that both initial processes end in q_f.

Suppose we have a run satisfying it. We saw previously that this the sequence of letters of A_1 written by $\mathcal{S}_1$ must lie in $L(\mathcal{G}_1)$. We showed that there is at most one active leaf in the run of $\mathcal{S}_1$ at all times. Furthermore, the $\mathbf{rd}(a_2)\mathbf{wr}(a_1)$ actions are made consecutively by a process holding both of its locks. Therefore the sequence of writes and reads of $\mathcal{S}_1$ in $A_1 \cup A_2$ must be of the form $\mathbf{rd}(a_2^1)\mathbf{wr}(a_1^1) \cdots \mathbf{rd}(a_2^k)\mathbf{wr}(a_1^k)$ with $a^1 \cdots a^k \in L(\mathcal{G}_1)$.

As a result, the sequence of values taken by the shared variable in A_1 must be a subword of the one in A_2. A symmetric argument shows that the sequence in A_2 is also a subword of the one in A_1, thus the two are equal. As a result, we obtain a word generated by both grammars.

Now suppose we have a word w generated by both grammars. Then we can simply take the two runs of $\mathcal{S}_1$ and $\mathcal{S}_2$ corresponding to w and interleave them so that for each letter a first $\mathcal{S}_2$ writes a_2 then $\mathcal{S}_1$ reads a_2 and writes a_1 and then $\mathcal{S}_2$ reads a_1.

There are no other interactions between the two parts of the run as they use disjoint locks and letters apart from $A_1 \cup A_2$.

As a result there is a run satisfying the specification if and only if the two grammars produce a common word. □

We can observe that the construction above requires processes to take turns writing on the shared variable an unbounded amount of times. This invites the following restriction, which requires that the process writing on the shared variable only changes a bounded number of times.

Definition 5.34

Let $K \in \mathbb{N}$, let ϱ be a run. We say that ϱ has K *writer reversals* if we can cut ϱ into K runs $\varrho_0, \ldots, \varrho_K$ such that in each ϱ_i every **wr** operation happens on the same left branch.

We can therefore redefine the problems seen so far with *bounded writer reversals*, that is, we are given a bound K in the input and only consider runs with at most K writer reversals.

It is already known that this restriction does not let us recover decidability when processes have stacks. Here we call pushdown DLSSV the systems obtained by extending pushdown DLSS with a shared variable.

Theorem 5.35

The state reachability problem for pushdown DLSSV is undecidable even with bounded writer reversals.

Proof. It was shown in [**AtigBKS14**] that the state reachability problem is undecidable for three pushdown processes communicating via a shared variable, even with only three phases (which they call stages).

It suffices to build a system in which three processes are spawned and apply their construction to obtain the undecidability. □

However, we conjecture that the problems at hand in this chapter are decidable on DLSSV with bounded writer reversals and finite-state processes.

Conjecture 5.36

The verification of DLSSV against regular objectives is decidable with bounded writer reversals.

Conjecture 5.37

The controller synthesis problem for DLSSV against regular objectives is decidable with bounded writer reversals.

The problem above asks for a strategy σ such that limit configurations of runs respecting σ are all accepted by a given parity tree automaton. There is a closely-related question, stated below, which has the advantage of providing a winning control strategy when the answer is yes. The previous problem only asks for a strategy that is winning on runs with bounded writer reversals.

Conjecture 5.38

Given a DLSSV, a parity tree automaton $\mathcal{B}$ and a bound K, is there a control strategy σ such that every σ-run has at most K writer reversals and has a limit configuration $(\tau, \lambda) \notin \mathcal{L}(\mathcal{B})$?

5.8 Conclusions

We have considered verification of parametric lock sharing systems where processes can spawn other processes and create new locks. Representing configurations as trees, and the notion of the limit configuration, are instrumental in our approach. We believe that we have made stimulating observations about this representation. It is very easy to express fairness as a property of a limit configuration. Many interesting properties, including liveness, can be formulated very naturally as properties of limit trees (cf. page 167). Moreover, there are structural conditions characterizing when a tree is a limit configuration of a run of a given system (Lemma 5.9).

As the dining philosophers example suggests, for many systems the maximal arity should be quite small (cf. Figure 5.1). Indeed, the maximal arity of the system corresponds to the tree width of the graph where process instances are nodes and edges represent sharing a lock. The maximal priority will be often 3. In our opinion, most interesting properties would have the form “there is a left path such that” or “all left paths are such that”, and these properties need only automata with three priorities. So in this case our verification algorithm is in PTIME.

Our handling of pushdown processes is different from the literature. Most of our development is done for finite state processes, while the transition to pushdown process is handled through right-resetting concept. Proposition 5.20 implies that in our context pushdown processes are essentially as easy to handle as finite processes.

We leveraged our results on verification to reach our final goal of this chapter, i.e., handle synthesis of local controllers. There is a complexity gap there. It would be interesting to find a way to avoid the determinisation step in Lemma 5.30.

We observed that the extension of the model with a shared variable (with only write and read operations) is undecidable. We suggest an approach, which requires that the writer changes only a bounded number of times. As further work it would be interesting to see if it is possible to extend our approach to treat join operation [**GawlitzaLMSW11**]. An important question is how to extend the model with some shared state in addition to the locks and still retain decidability for the pushdown case.

Chapter 6

Conclusion and Perspectives

I was working on the proof of one of my poems all the morning and took out a comma. In the afternoon—well, I put it back again.

OSCAR WILDE

Overview of the thesis

This work aims at presenting a fresh approach to the distributed controller synthesis problem. We study three a priori different models. Surprisingly, a common proof structure emerges in the three cases for the decidability of controller synthesis.

Let us reformulate it once again:

1. We define "good" sets of local runs: invariants in Chapter 3, behaviours in Chapters 4 and 5. They come with a finite description: behaviours are finite sets of patterns, while invariants can be finitely described using bases.

2. Then we show that if a strategy is winning then the set of local runs allowed by this strategy is included in a good set. Additionally, for invariants we have to show a bound on the size of the description. For behaviours, this is clear.

3. We show that we can decide, given such a family, whether there is a strategy that only allows runs in it. We do this by encoding this problem as a two-player game with a regular winning condition.

4. The resulting algorithms enumerate or guess good sets of runs up to the computed bound, and for each one of them, look for local strategies which only allow local runs within this set.

This is a step in the right direction, which gives us a path towards synthesizing controllers for increasingly powerful models. A simplified version of this approach can be used for verification.

Let us now go over what we learned in each chapter.

Broadcast networks of register automata This chapter introduces a very powerful model of networks of processes communicating by unreliable broadcasts of signed messages. The main theorem is the decidability of controller synthesis for this model. However, we think that the simpler case of signature BGR already improves considerably our understanding of the problems at hand. It also illustrates very well the idea of invariants, which is central in this work. We now understand quite well the general model of BGR. This is illustrated in the section discussing extensions: the fact that we can easily handle extensions of the model shows that our understanding is robust. The study of 1BGR also brings some interesting points: first of all in the literature many models give each process a single datum, such as data VASS or population protocols with data. We may thus build connections between 1BGR and these other works. In addition, the complexity of the problems we study in this chapter drop dramatically between BGR and 1BGR.

Lock-sharing systems The point of this chapter is to explore a framework where distributed verification and synthesis are not only non-trivial and decidable but even tractable. The central results are the NP-completeness of verification for 2LSS and nested LSS, along with the Σ_2^P-completeness of synthesis for both of those cases. This strongly contrasts with the undecidability of synthesis in general LSS. The decidability results stem from the use of patterns, with which we describe local runs and strategies. The other important tool in our approach is the graph representation of deadlocks in 2LSS. This lets us reduce the verification and synthesis with respect to deadlock avoidance to the exploration of a graph. This yields surprisingly good results when we add some extra requirements: Many of the results assume that the systems are locally live or exclusive (or both). Those are not strong requirements: an abstraction from a program to an LSS can easily be made to satisfy both those conditions.

Dynamic lock-sharing systems As LSS with a fixed number of processes are quite well understood, we extend them to be able to model less restricted programs. We make the nested assumption once again, and obtain mainly two decidability results: verification of nested against regular tree languages (even with pushdown processes) and synthesis of that same model. This is promising for several reasons. First, we obtain low complexities for the problems studied when the arity of the processes are fixed. Second, this provides solid ground on which to build increasingly powerful models, as suggested in the section about addition of variables. Third, we brought down the verification and synthesis of those systems to questions on tree automata. The remaining problems (for instance about the exact complexity of synthesis) can be formulated as motivating questions for the tree automata community.

Future directions

The most exciting open problems from this thesis are Conjectures 5.36 and 5.37. The verification and synthesis of dynamic systems with nested locks and variable with bounded writer reversal would be a big step in the study of those systems. As a side note, it would let us solve the Prisoners and lightbulb problem presented in the introduction automatically, since this problem admits a solution with bounded writer reversal. Of course, the addition of variables (with bounded writer reversals) is also possible in LSS. We are however quite optimistic and hope that we can make this work directly for DLSS. The ultimate goal would be to obtain an NP algorithm for the controller synthesis of DLSSV and a P algorithm for the verification, when the adequate parameters are bounded. It would also be interesting to analyse the parameterised complexity of all problems considered in Chapter 5 with respect to the arity of processes.

For LSS, the most tractable cases call for an implementation. We are currently working on a project with Romain Delpy to implement some of the algorithms presented in this thesis in C. We use SAT solvers for the NP-complete problems. Experimental results will let us validate or not the theoretical gains in efficiency. From the theoretical standpoint, the interplay of randomisation with this model would be very interesting: for instance we could strengthen the fairness assumption on the scheduler to only consider randomised schedulers.

Concerning broadcast networks, we now have a reasonably good understanding of potential extensions of the model. What could be interesting to investigate is the connection to immediate-observation population protocols with data, as presented in Section 3.8.3. In particular we think that proving Open problem 3.64 requires some new abstraction techniques. Indeed, common tools like the so-called *copycat property* (which allows us to duplicate agents as many times as needed and still obtain a valid run) fail here. Distributed systems with data are promising and we hope to be able to build connections with other models.

Overall, we expect that we will be able to apply the methods developed here in many other frameworks, in particular distributed systems with other means of communication such as rendezvous or shared variables. We may then obtain an even stronger breach in the challenging problem of distributed synthesis.

Appendix A

Run reduction of register automata

This appendix follows Chapter 3. Our aim is to prove Theorem A.1, which states that given a local run u, we can always find a local run v with the same start and end local configurations, of length bounded by a function of $|\mathcal{R}|$, and "cheaper" in the sense that the d-input required by v is a subword of the one required by u, for all data d.

In addition to this new run reduction technique, we obtain the decidability of Cover for signature BNRA as a corollary, yielding a simpler proof than the one presented in Section 3.4. In fact, the decidability of Cover for general BNRA can also be inferred in a similar way.

Theorem A.1 ▶ Register transducer run reduction

Let $\mathcal{R}$ be a register transducer with r registers. Let $u : (q_{start}, c_{start}) \xrightarrow{*} (q_{end}, c_{end})$ be a local run of $\mathcal{R}$. Then there exists $v : (q_{start}, c_{start}) \xrightarrow{*} (q_{end}, c_{end})$ of length at most $\varphi(\mathcal{R}, r, r)$ and such that for all $d \in \mathbb{D}$, $\mathbf{In}_d(v) \sqsubseteq \mathbf{In}_d(u)$.

The function φ will be defined later. The main difficulty is that the set of configurations is infinite, since the set of data is. Usually, in register automata we allow ourselves to switch or rename data at will throughout the local run, which allows us to shorten runs more easily. However, here, as we want the resulting d-input to be a subword of the previous one, we cannot replace data with other data arbitrarily. Finally, it may be the case that some data appear in many or even all local configurations throughout the local run, while others appear sporadically.

First let us describe informally how we prove the theorem. We consider a very long local run u and attempt to reduce it. We call a register active if its content is modified at some point during the local run.

We proceed by induction over the number of active registers. If there are no active registers, then the number of different local configurations in u is bounded by $|Q|$. If u is of length more than Q, we can reduce it.

Now assume that we can reduce all runs with at most p active registers to runs of length at most M. We consider a run u with $p + 1$ active registers. If there is a section of u of length more than M with $\leq p$ active registers, then we can reduce that section to reduce u. Otherwise, every section of at least M steps has $p+1$ active registers. In other words, in every section of M steps, all $p+1$ active registers of u are updated at some point. The trick is then to look for two such sections with the same sequence of transitions, in which the persistent data appear at the same places and such that the sections "look the same up to renaming data". We are then able to merge those two pieces of run by replacing data one by one when registers are updated (see Figure A.2).

To prove this theorem, an important step is to be able to "merge" two pieces of local runs that have the same transitions and other similarities, but may contain different data. To do so, we use, amongst other things, a combinatorial lemma of Erdös and Rado, often called the sunflower lemma.

Definition A.2 ▶ Sunflower

A *sunflower* is a family of sets $\mathcal{S}$ such that there exists a set Y (the *kernel*) such that $A \cap B = Y$ for all $A \neq B \in \mathcal{S}$. Its *size* is the number of sets in $\mathcal{S}$

Lemma A.3 ▶ Sunflower lemma [ErdosR1960intersection]

Let $r, s \in \mathbb{N}$, and let $\mathcal{F}$ be a family of subsets of a set U, such that each set $F \in \mathcal{F}$ is of cardinality s. If $|\mathcal{F}| > s!(r-1)^s$ then $\mathcal{F}$ contains a sunflower of size r.

We start by slightly modifying the statement to make it more convenient for us. We want to allow sets of cardinality *at most* k and to allow a set to appear several times.

Corollary A.4

Let $r, s \in \mathbb{N}$, and let $F_1, \ldots, F_m$ be a family of subsets of a set U, such that each set F_i is of cardinality at most s. If $m > (s+1)!(r-1)^{s+1}$ then there exists $I \subseteq [1, m]$ and $Y \subseteq U$ such that $|I| \geq r$ and for all $i \neq j \in I$, $F_i \cap F_j = Y$.

Proof. We simply add fresh distinct elements to each F_i to obtain a family of distinct sets, all of cardinality $s+1$, and then apply the sunflower lemma.

Formally, let $U' = U \sqcup \{x_{p,q} \mid i \in [1, m], j \in [1, s+1]\}$ be a set containing U and $(s+1)m$ fresh extra elements.

For all $i \in [1, m]$, we define $F'_i = F_i \cup \{x_{i,j} \mid j \in [|F_i|+1, s+1]\}$. Note that for all $i \neq k \in [1, m]$, $F'_i \cap F'_k = F_i \cap F_k$.

As the F'_i are all distinct and of cardinality $s+1$, we can apply the sunflower lemma. We obtain a set $I \subseteq [1, m]$ such that $|I| \geq r$ and there exists $Y \subseteq U'$ such that for all $i, k \in I$, $F_i \cap F_k = F'_i \cap F'_k = Y$ (thus $Y \subseteq U$). This concludes our proof. ☐

We use the sunflower lemma to demonstrate a key lemma for our main result on register transducers. The lemma states that in a large family of functions from a finite set we can find two that are akin, as defined below and illustrated in Figure A.1.

Definition A.5 ▶ Akin functions

Let S be a finite set. Two functions $f, g \in \mathbb{D}^S$ are *akin* if:

1. for all $s \in S$, either $f(s) = g(s)$ or $f(s) \notin g(S)$ and $g(s) \notin f(S)$.
2. for all $s, s' \in S$, $f(s) = f(s')$ if and only if $g(s) = g(s')$.

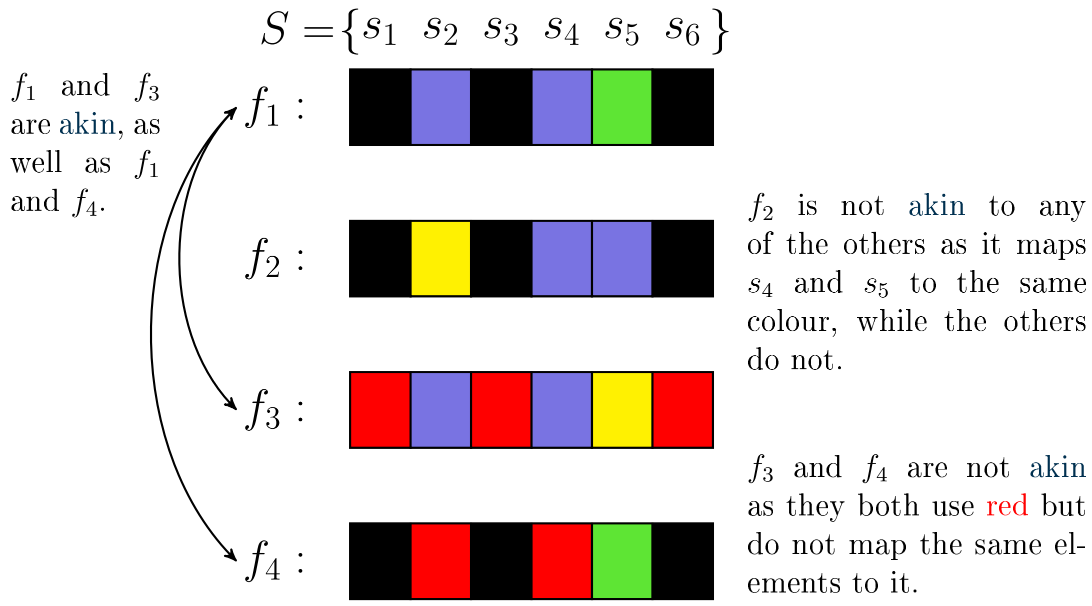

$S = \{s_1 \ s_2 \ s_3 \ s_4 \ s_5 \ s_6\}$
f_1 and f_3 are akin, as well as f_1 and f_4.
$f_1 :$
$f_2 :$
$f_3 :$
$f_4 :$
f_2 is not akin to any of the others as it maps s_4 and s_5 to the same colour, while the others do not.
f_3 and f_4 are not akin as they both use red but do not map the same elements to it.

Figure A.1: Four functions from a set S with 6 elements to $\mathbb{D}$ (data are represented by colours), illustrating the definition of akin.

The intuition of the next corollary is as follows: We have a family of functions $\mathcal{T}$ from a finite set S to $\mathbb{D}$. We show that if that family is large enough then two of those functions are akin. To do so, we start by considering the family $(f(S))_{f\in\mathcal{T}}$. If $\mathcal{T}$ is large enough, then that family contains a large sunflower with a kernel Y. We then apply pigeonhole arguments to find two functions which agree on elements mapped to Y (condition 1) and whose set of preimages is the same (condition 2).

Define $\psi(n) = (n+1)!2^{(n+1)^3}(n+1)^{(n+1)^2}$

Lemma A.6 ▶ Akin functions lemma

Let S be a finite set and $\mathcal{T} \subseteq \mathbb{D}^S$. If $|\mathcal{T}| \geq \psi(|S|)$ then there exist $f, g \in \mathcal{T}$ such that f and g are akin.

Proof. We apply Lemma A.3 on $(f(S))_{f\in\mathcal{T}}$, with $s = |S|$ and $r = 2^{|S|^2}(|S|+1)^{|S|}$.

As $|\mathcal{T}| \geq (|S|+1)!2^{(|S|+1)^3}(|S|+1)^{(|S|+1)^2} > (s+1)!(r-1)^{|S|+1}$, we obtain a subfamily $\mathcal{U} \subseteq \mathcal{T}$ and a set $Y \subseteq S$ such that $|\mathcal{U}| \geq r$ and for all $f \neq g \in \mathcal{U}$, $f(S) \cap g(S) = Y$. Note that $|Y| \leq |S|$.

We then construct a large subset of functions of $\mathcal{U}$ that are equal on elements mapped to Y: for all $f \in \mathcal{U}$ let $\tilde{f} : S \to Y \cup \{\bot\}$ be the function such that $\tilde{f}(s) = f(s)$ if $f(s) \in Y$ and $\tilde{f}(s) = \bot$ otherwise.

As there are $(|Y|+1)^{|S|} \leq (|S|+1)^{|S|}$ possibilities for $\tilde{f}$, and as $|\mathcal{U}| \geq r = 2^{|S|^2}(|S|+1)^{|S|}$, there exists $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| \geq 2^{|S|^2}$ and $\tilde{f} = \tilde{g}$ for all $f, g \in \mathcal{V}$.

As a last step, for all $f \in \mathcal{V}$ we define $\equiv_f \subseteq S \times S$ as the relation such that $s \equiv_f s'$ if and only if $f(s) = f(s')$. There are less than $2^{|S|^2}$ possibilities for $\equiv_f$, hence $\mathcal{V}$ contains two functions f and g such that $\equiv_f$ and $\equiv_g$ are equal.

To sum up, we have found two functions such that:

- $f(S) \cap g(S) = Y$ and $\tilde{f} = \tilde{g}$: as a consequence, for all $s \in S$, either $f(s) = g(s)$ or $f(s) \notin g(S)$ and $g(s) \notin f(S)$.
- $\equiv_f$ and $\equiv_g$ are equal: in other words, for all $s, s' \in S$, $f(s) = f(s')$ if and only if $g(s) = g(s')$.

This concludes our proof. □

We can now see how to "merge" very similar local runs.

We say that a register i is *active* in local run u if its content changes at some point in u. A register that is not active is called *passive*. We also say that a datum d is *persistent* in u if it appears in every local configuration in u. Otherwise we say that d *vanishes* in u.

The next lemma is quite technical, but can be visualised relatively easily, see Figure A.2.

Lemma A.7 ▶ Merging akin runs

Let u, u' be two local runs of length n with the same sequence of transitions.
$u = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$
$u' = (q_0, c'_0) \xrightarrow{\mathbf{op}_1(m_1,d'_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_n(m_n,d'_n)}_{\delta_n} (q_n, c'_n)$
Let $f, f' : [0, r] \times [1, M] \to \mathbb{D}$ be such that $f(i, p) = c_p(i)$ and $f'(i, p) = c'_p(i)$ for all $(i, p) \in [1, r] \times [1, M]$ and $f(0, p) = d_p$ and $f'(0, p) = d'_p$.
If f and f' are akin, and every datum that does not appear in u' vanishes in u then there is a local run w of length M from (q_0, c_0) to (q'_M, c'_M) with the same sequence of transitions and such that for all $d \in \mathbb{D}$, either $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u)$ or $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u')$.

Proof. For each d appearing in u but not u', since d vanishes in u we can define $k(d)$ as an index such that d does not appear in $c_{k(d)}$.

For all $\ell \in [0, n]$ we define d^w_ℓ as d_ℓ if d_ℓ appears in u' or $\ell \leq k(d_\ell)$ and d'_ℓ otherwise.

We also define, for all $\ell \in [0, n]$, c^w_ℓ as follows: for all $i \in [1, r]$, $c^w_\ell(i) = c_\ell(i)$ if $c_\ell(i)$ appears in u' or $\ell \leq k(c_\ell(i))$ and $c^w_\ell(i) = c'_\ell(i)$ if $\ell \geq k(c_\ell(i))$. Note that this definition is consistent as we cannot actually have that $c_\ell(i)$ does not appear in u' and $\ell = k(c_\ell(i))$, by definition of $k(c_\ell(i))$.

We now prove that $(q_{\ell-1}, c^w_{\ell-1}) \xrightarrow{\mathbf{op}_\ell(m_\ell,d^w_\ell)}_{\delta_\ell} (q_\ell, c^w_\ell)$ is a valid local step for all $\ell \in [1, M]$.

We do so using the following claims. The first one ensures that a the datum d^w_ℓ involved in a transition appears at the same places before and after the transition in w as d_ℓ in u.

Claim A.7.1. *For all* $\ell \in [1, M]$, $c^w_{\ell-1}(i) = d^w_\ell \Leftrightarrow c_{\ell-1}(i) = d_\ell$ *and* $c^w_\ell(i) = d^w_\ell \Leftrightarrow c_\ell(i) = d_\ell$

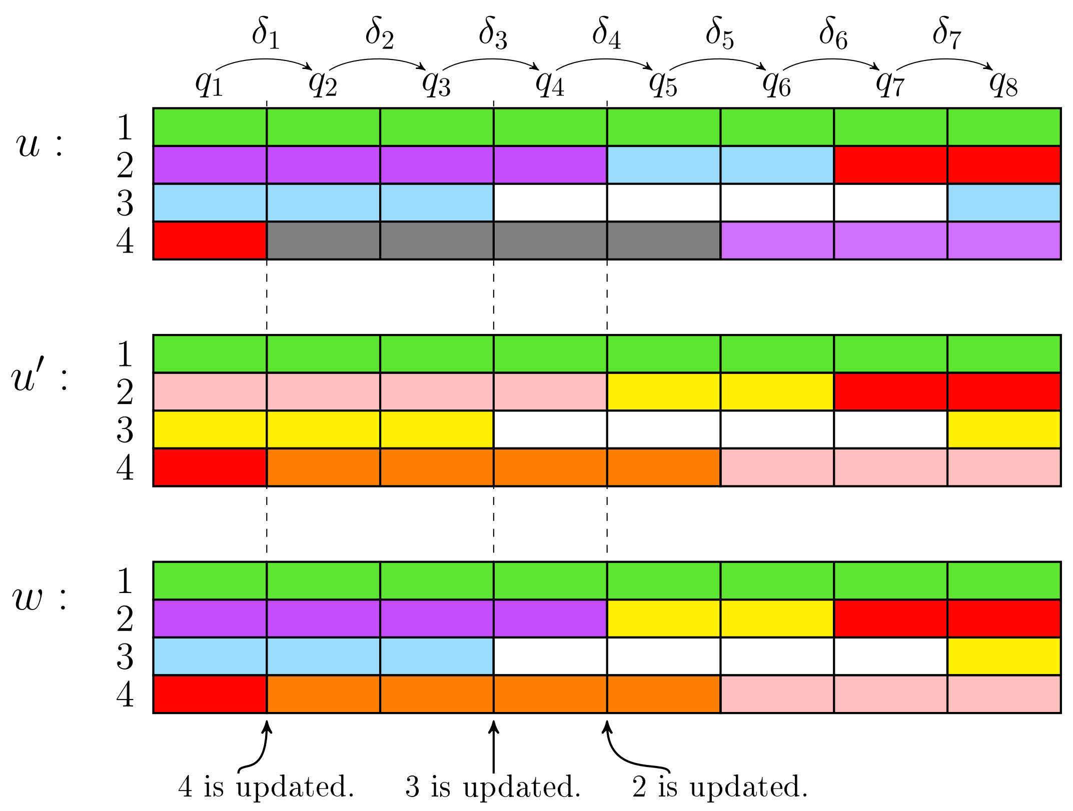

δ_1 δ_2 δ_3 δ_4 δ_5 δ_6 δ_7
q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8
u :
u' :
w :
1
2
3
4
4 is updated.
3 is updated.
2 is updated.

Figure A.2: Illustration of Lemma A.7. The functions mapping positions and registers to the data in u and u' are akin: white, green and red appear at the same places in both, and the purple, blue and gray positions in u match respectively the pink, yellow and orange positions in u'.
We construct w by picking a position where purple (resp. blue, gray) does not appear, and use it to switch from purple to pink (resp. blue to yellow, gray to orange) without causing conflicts in the equality tests.

Proof of the claim. First, note that for all p, by definition if $c_p^w(i) = d$ then either $c_p(i) = d$ or $c'_p(i) = d$.

- If d_ℓ^w appears in both u and u', then as f and f' are akin $f(i,p) = d_\ell^w \Leftrightarrow f'(i,p) = d_\ell^w$ for all i, p, by 1 of the definition of akin. Hence $c_p(i) = d_\ell^w \Leftrightarrow c'_p(i) = d_\ell^w$ for all i, p.

  As a consequence, if $c_p(i) = d$ then $c_p^w(i) = d$ (by definition) and if $c_p^w(i) = d$ then $c_p(i) = d$ or $c'_p(i) = d$ and as $c_p(i) = d \Leftrightarrow c'_p(i) = d$, $c_p(i) = d$. Thus, for all i, p, $c_p^w(i) = d \Leftrightarrow c_p(i) = d$.

- If d_ℓ^w appears in u but not u' then $d_\ell^w = d_\ell$ and $\ell \leq k(d_\ell)$. As a consequence, for all $p \leq \ell$, if $c_p(i) = d_\ell^w$ then $c_p^w(i) = d_\ell^w$ (by definition) and if $c_p^w(i) = d_\ell^w$ then $c_p(i) = d_\ell^w$ or $c'_p(i) = d_\ell^w$ and as d_ℓ^w does not appear in u', $c_p(i) = d_\ell^w$. Thus, for all i and $p \leq \ell$, $c_p^w(i) = d_\ell^w \Leftrightarrow c_p(i) = d_\ell$.

- Similarly, if d_ℓ^w appears in u' but not u then $d_\ell^w = d'_\ell$ and $\ell > k(d_\ell)$. As a consequence, for all $p \geq \ell - 1$, if $c_p(i) = d_\ell$ then $c_p^w(i) = d'_\ell = d_\ell^w$ (by definition) and if $c_p^w(i) = d_\ell^w$ then $c_p(i) = d_\ell^w$ or $c'_p(i) = d_\ell^w$ and as d_ℓ^w does not appear in u', $c'_p(i) = d_\ell^w$. Thus, for all i and $p \geq \ell - 1$, $c_p^w(i) = d_\ell^w \Leftrightarrow c_p(i) = d_\ell$.

In particular, in all three cases we have that $c^w_{\ell-1}(i) = d^w_\ell \Leftrightarrow c_{\ell-1}(i) = d_\ell$ and that $c^w_\ell(i) = d^w_\ell \Leftrightarrow c_\ell(i) = d_\ell$. ■

The second claim states that if the content of a register stays the same before and after a step in u then it does so also in w.

Claim A.7.2. *For all $\ell \in [1, M]$, if $c_{\ell-1}(i) = c_\ell(i)$ then $c^w_{\ell-1}(i) = c^w_\ell(i)$.*

Proof of the claim. By definition we cannot have $k(c_{\ell-1}(i)) = \ell - 1$. Thus either $k(c_{\ell-1}(i)) \geq \ell$ or $k(c_{\ell-1}(i)) < \ell - 1$.
In the first case we have $c^w_{\ell-1}(i) = c_{\ell-1}(i) = c_\ell(i) = c^w_\ell(i)$.
In the second case, since f and f' are akin and $f(i, \ell - 1) = f(i, \ell)$, we obtain that $f'(i, \ell-1) = f'(i, \ell)$, hence $c'_{\ell-1}(i) = c'_\ell(i)$. As a result, $c^w_{\ell-1}(i) = c'_{\ell-1}(i) = c'_\ell(i) = c^w_\ell(i)$.
This concludes our proof ■

Now let $\ell \in [1, M]$, we must show that $(q_{\ell-1}, c_{\ell-1}) \xrightarrow{\mathbf{op}_\ell(m_\ell, d_\ell)}_{\delta_\ell} (q_\ell, c_\ell)$.

If δ_ℓ is a record transition $\delta_\ell = q_{\ell-1} \xrightarrow{\mathbf{rec}(m_\ell, \downarrow i)} q_\ell$ then, by the previous claim, for all $j \in [1, r]$,

- $c_{\ell-1}(j) \neq d_\ell$ and by Claim A.7.1, $c^w_{\ell-1}(j) \neq d^w_\ell$
- $c_\ell(i) = d_\ell$ and by Claim A.7.1, $c^w_\ell(i) = d_\ell$
- if $j \neq i$ then $c_\ell(j) = c_{\ell-1}(j)$ and by Claim A.7.2, $c^w_\ell(j) = c_\ell j$

If δ_ℓ is an equality transition $\delta_\ell = q_{\ell-1} \xrightarrow{\mathbf{rec}(m_\ell, = i)} q_\ell$ then, by the previous claim, for all $j \in [1, r]$,

- $c_{\ell-1}(i) = d_\ell$ and by Claim A.7.1, $c^w_{\ell-1}(i) = d^w_\ell$
- $c_\ell(j) = c_{\ell-1}(j)$ and by Claim A.7.2, $c^w_\ell(j) = c_\ell j$

If δ_ℓ is a disequality transition $\delta_\ell = q_{\ell-1} \xrightarrow{\mathbf{rec}(m_\ell, \neq)} iq_\ell$ then, by the previous claim, for all $j \in [1, r]$,

- $c_{\ell-1}(j) \neq d_\ell$ and by Claim A.7.1, $c^w_{\ell-1}(j) \neq d^w_\ell$
- $c_\ell(j) = c_{\ell-1}(j)$ and by Claim A.7.2, $c^w_\ell(j) = c_\ell j$

If δ_ℓ is a broadcast transition $q_{\ell-1} \xrightarrow{\mathbf{br}(m_\ell, i)} q_\ell$ then:

- As $c_{\ell-1}(i) = d_\ell$, by Claim A.7.1, $c^w_{\ell-1}(i) = d^w_\ell$.
- For all $j \in [1, r]$, $c_{\ell-1}(j) = c_\ell(j)$ thus by Claim A.7.2, we have $c_{\ell-1}(j) = c_\ell(j)$.

As a consequence, we have $(q_{\ell-1}, c_{\ell-1}) \xrightarrow{\mathbf{op}_\ell(m_\ell, d_\ell)}_{\delta_\ell} (q_\ell, c_\ell)$, for all ℓ.
Hence w is a valid local run, of length M and with the same transitions as u and u'.
Finally, let $d \in \mathbb{D}$, if d appears in u and u' then $\mathbf{In}_d(w) = \mathbf{In}_d(u) = \mathbf{In}_d(u')$. If d appears in u and not u' then $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u)$ (as $d^w_\ell = d$ implies $d_\ell = d$). Similarly, if d appears in u' and not u then $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u')$. This concludes our proof. □

Now that we know how to merge similar local runs, we can prove our main theorem of this section. Intuitively, we proceed as follows: We reason by induction on the number of active registers. Let A be the set of active registers.

If there is a datum d that appears continuously throughout a long part of the local run, then by the definition of local steps it stays in the same register throughout that part of the run. This register is thus passive. This gives a long local run with $|A|-1$ active registers, which we can reduce by induction hypothesis. If every register outside of A is active in every sufficiently long part of the local run, then if the local run is long enough we can cut it into many long parts. We will then be able to find two sections of the local run that satisfy the conditions of Lemma A.7, allowing us to shorten the local run by removing the part between those two sections and pasting them together.

Define φ as follows:

$$\varphi(\mathcal{R}, r, j) = \begin{cases} |\mathcal{R}| \text{ if } j = 0 \\ \psi(r\varphi(\mathcal{R}, r, j-1))|\mathcal{R}|^{\varphi(\mathcal{R},r,j-1)}\varphi(\mathcal{R}, r, j-1) \text{ if } j > 0 \end{cases}$$

Proof of Theorem A.1. We set $u = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} (q_1, c_1) \xrightarrow{\mathbf{op}_2(m_2,d_2)}_{\delta_2} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$.

Let A be the set of active registers in u. We prove the theorem by induction on $|A|$.

Base case: suppose u has no active register. This implies that no register is ever updated, hence all c_i are equal.

As a consequence, if $|u| > |Q|$ then there exist $k < \ell \in [0, n]$ such that $q_k = q_\ell$ and $c_k = c_\ell$. Hence we can define the local run $v = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_k(m_k,d_k)}_{\delta_k} (q_k, c_k) \xrightarrow{\mathbf{op}_{\ell+1}(m_{\ell+1},d_{\ell+1})}_{\delta_{\ell+1}} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$, which is shorter than u and clearly satisfies $\mathbf{In}_d(v) \sqsubseteq \mathbf{In}_d(u)$ for all $d \in \mathbb{D}$.

Induction: suppose the statement holds for all local run with at most a active registers and suppose u has $a+1$ active registers.

We distinguish two cases. The first case is when there exist indices $k, \ell \in [0, n]$ and $i \notin A$ such that $\ell - k > \varphi(|\Delta|, r-p-1)$ and i is passive in the local run $u' = (q_k, c_k) \xrightarrow{\mathbf{op}_{k+1}(m_{k+1},d_{k+1})}_{\delta_{k+1}} \cdots \xrightarrow{\mathbf{op}_\ell(m_\ell,d_\ell)}_{\delta_\ell} (q_\ell, c_\ell)$.

As a consequence, u' has at most a active registers. By induction hypothesis, we have a run $v' : (q_k, c_k) \xrightarrow{*} (q_\ell, c_\ell)$ shorter than u'. We can set $u_- = (q_0, c_0) \xrightarrow{\mathbf{op}_1(m_1,d_1)}_{\delta_1} \cdots \xrightarrow{\mathbf{op}_k(m_k,d_k)}_{\delta_k} (q_k, c_k)$ and $u_+ = (q_\ell, c_\ell) \xrightarrow{\mathbf{op}_{\ell+1}(m_{\ell+1},d_{\ell+1})}_{\delta_{\ell+1}} \cdots \xrightarrow{\mathbf{op}_n(m_n,d_n)}_{\delta_n} (q_n, c_n)$. We then define v as the concatenation of u_-, v' and u_+. Clearly v is a local run from (q_0, c_0) to (q_n, c_n). Furthermore for all $d \in \mathbb{D}$,

$$\mathbf{In}_d(v) = \mathbf{In}_d(u_-)\mathbf{In}_d(v')\mathbf{In}_d(u_+) \sqsubseteq \mathbf{In}_d(u_-)\mathbf{In}_d(u')\mathbf{In}_d(u_+) = \mathbf{In}_d(u).$$

It remains to tackle the other case, in which every register $i \in A$ is active in every sequence of more than $\varphi(|\Delta|, r-p+1)$ local steps. Let $M = \varphi(|\Delta|, r-p-1)$. If $|u| > \varphi(|\Delta|, r-p) = \psi(rM)|\mathcal{R}|^M M$ then we can cut u into $\psi(rM)|\mathcal{R}|^M$ intervals of size M: For all $k \in [0, \psi(rM)|\mathcal{R}|^M - 1]$ we define u_k as the section of u between (q_{kM}, c_{kM}) and $(q_{(k+1)M-1}, c_{(k+1)M-1})$.

As each u_k is of length M, there are less than $|\mathcal{R}|^M$ possibilities for its sequence of transitions. We can thus find $\psi(rM)$ intervals which all have the same sequence of transitions: there is a set $K \subseteq [0, \psi(rM)|\mathcal{R}|^M - 1]$ such that $|K| \geq \psi(rM)$ and all $(u_k)_{k\in K}$ have the same sequence of transitions.

For each $k \in K$ we define $f_k : [0, r] \times [0, M-1] \to \mathbb{D}$ as $f_k(i,p) = c_{kM+p}(i)$ for $i \geq 1$ and $f_k(0,p) = d_{kM+p}$. We apply Lemma A.6 with $S = [0,r] \times [0, M-1]$: as there are $|K| \geq \psi(rM)$ such functions there are two indices $k, \ell \in K$ such that u_k and u_ℓ have the same transitions and f_k, f_ℓ are akin functions.

To apply Lemma A.7 we need to show that every datum that does not appear in u_ℓ vanishes in u_k. We do this by contraposition: take d a datum that does not vanish in u_k. Then, as record transitions require that the recorded datum is not already in a register, no record transition has stored d in u_k. Hence there is a register i that contains d in all configurations of u_k. This means i is passive in u_k, and thus $i \in A$. As a consequence, i contains the same datum in all of u, and in particular in all of u_ℓ, hence d appears in u_ℓ.

By Lemma A.7 there exists a run w of length M from (q_{kM}, c_{kM}) to $(q_{(\ell+1)M}, c_{(\ell+1)M})$ such that for all $d \in \mathbb{D}$, either $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u_k)$ or $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u_\ell)$. In particular, for all $d \in \mathbb{D}$, we have $\mathbf{In}_d(w) \sqsubseteq \mathbf{In}_d(u_k)\mathbf{In}_d(u_\ell)$.

Let $u_{<k}$ and $u_{>\ell}$ be respectively the prefix of u up to (q_{kM}, c_{kM}) and the suffix of u from $(q_{(\ell+1)M}, c_{(\ell+1)M})$.

By concatenating $u_{<k}$, w and $u_{>\ell}$, we obtain a run v from (q_0, c_0) to (q_n, c_n) which is shorter than u and such that for all $d \in \mathbb{D}$,

$\mathbf{In}_d(v) = \mathbf{In}_d(u_{<k})\mathbf{In}_d(w)\mathbf{In}_d(u_{>\ell}) \sqsubseteq \mathbf{In}_d(u_{<k})\mathbf{In}_d(u_k)\mathbf{In}_d(u_\ell)\mathbf{In}_d(u_{>\ell}) \sqsubseteq \mathbf{In}_d(u)$

This concludes our proof. □

We can now extend the previous result with little difficulty to preserve a part of the output. Say we have a local run in which a sequence of messages dw is broadcast. Then we can apply the previous result to reduce the runs between two broadcasts of dw. The resulting run still outputs dw, and is of size at most $|dw|\varphi(|\Delta|, r)$.

Recall that $\sqsubseteq$ is the subword relation over words.

Corollary A.8 ▶ Register transducer run reduction with output

Let $\mathcal{R}$ be a register transducer with r registers.
Let u be a local run of $\mathcal{R}$ and dw a word over $\mathcal{M} \times \mathbb{D}$ such that $dw \sqsubseteq \mathbf{Out}(u)$.
Then there exists a local run v of length at most $|dw|\varphi(\mathcal{R}, r, r)$ such that $dw \sqsubseteq \mathbf{Out}(v)$ and for all $d \in \mathbb{D}$, $\mathbf{In}_d(v) \sqsubseteq \mathbf{In}_d(u)$.

Proof. Let $dw = (m_1, d_1) \cdots (m_k, d_k)$. We can decompose u as

$$u = (q_0^-, c_0^-) \xrightarrow{u_0} (q_0^+, c_0^+) \xrightarrow{\mathbf{br}(m_1,d_1)}_{\delta_1} (q_1^-, c_1^-) \cdots \xrightarrow{\mathbf{br}(m_k,d_k)}_{\delta_k} (q_k^-, c_k^-) \xrightarrow{u_k} (q_k^+, c_k^+).$$

By Theorem A.1, for all $\ell \in [0, k-1]$ there is a run v_ℓ of length at most $\varphi(|\Delta|, r, r)$ from (q_ℓ^-, c_ℓ^-) to (q_ℓ^+, c_ℓ^+) such that $\mathbf{In}_d(v_\ell) \sqsubseteq \mathbf{In}_d(u_\ell)$ for all $d \in \mathbb{D}$.

We then define

$$v = (q_0^-, c_0^-) \xrightarrow{v_0} (q_0^+, c_0^+) \xrightarrow{\mathbf{br}(m_1,d_1)}_{\delta_1} (q_1^-, c_1^-) \xrightarrow{v_1} (q_1^+, c_1^+) \cdots \xrightarrow{\mathbf{br}(m_k,d_k)}_{\delta_k} (q_k^-, c_k^-).$$

Note that we replaced every u_ℓ by v_ℓ except for u_k, which we removed.

Clearly v is a valid run and $dw \sqsubseteq \mathbf{Out}(v)$. Furthermore, for all $d \in \mathbb{D}$, we have $\mathbf{In}_d(v) = \mathbf{In}_d(v_0) \cdots \mathbf{In}_d(v_{k-1}) \sqsubseteq \mathbf{In}_d(u_0) \cdots \mathbf{In}_d(u_{k-1}) \sqsubseteq \mathbf{In}_d(u)$. □

Finally, we obtain the decidability of Cover for signature BNRA as a by-product of the construction.

Corollary A.9

Cover is decidable for signature BNRA.

Proof. The problem is clearly recursively enumerable, as it sufficesto enumerate runs until we find one that covers the error state (or broadcasts the error message).

It remains to show that it is co-recursively enumerable. Consider Corollary 3.21. To show that a message cannot be broadcast, it suffices to find an invariant satisfying both conditions. As invariants are downward-closed, their complement have a finite basis. We can thus enumerate finite sets of words B, and check each time if $B\uparrow^{\mathsf{c}}$ satisfies both conditions.

Let $I = B\uparrow^{\mathsf{c}}$. The first condition is straightforward to check. For the second one, if the condition is not satisfied then there is a local run u such that $\mathbf{In}_d(u) \in B\uparrow^{\mathsf{c}}$ for all d and $\mathbf{Out}_{sign}(u) \notin I$.

As a consequence, there is a word $w \in B$ such that $w \sqsubseteq \mathbf{Out}_{sign}(u)$. By Corollary A.8, there is a local run v of length at most $|w|\varphi(\mathcal{R}, r, r)$ such that $w \sqsubseteq \mathbf{Out}_{sign}(v)$ and for all d, $\mathbf{In}_d(v) \sqsubseteq \mathbf{In}_d(u)$.

As I is downward-closed and does not contain w, we have $\mathbf{Out}_{sign}(v) \notin I$ and for all d, $\mathbf{In}_d(v) \in I$.

In conclusion, if $B\uparrow^{\mathsf{c}}$ does not satisfy the second condition of Corollary 3.21 then there is a local run of size at most $||B||\varphi(\mathcal{R}, r, r)$ witnessing it. This means that we can enumerate sets of words B and check the second condition. This proves that Cover for signature BNRA is co-recursively enumerable, and thus decidable. □

Appendix B

Population games

The line of research surrounding the Population control problem is very dear to me, as one of the first research problems I worked on came from it, and I kept encountering it throughout my research years. This is why this section is dedicated to the history of the model, the problems solved so far, the work in progress, and the most prominent open problems.

The story starts in 2017: Bertrand, Dewaskar, Genest and Gimbert introduce a population games, inspired by questions from biology on the control of yeast populations [**BertrandDGG17**].

A population game is described by an NFA with one initial and one final state. There are two players, Laetitia (who chooses letters) and Terrence (who chooses transitions). Some number N of tokens are placed on an initial state of an NFA. The two players then play alternately: Laetitia chooses a letter x, then Terrence moves each token along an x-labelled transitions. Laetitia wins if all tokens end up on the same final state eventually. The question is then: does Laetitia have a winning strategy against any number of tokens? The authors showed that this problem is ExpTime-complete, by converting this question into an ω-regular game using an involved automaton construction.

See an example below.

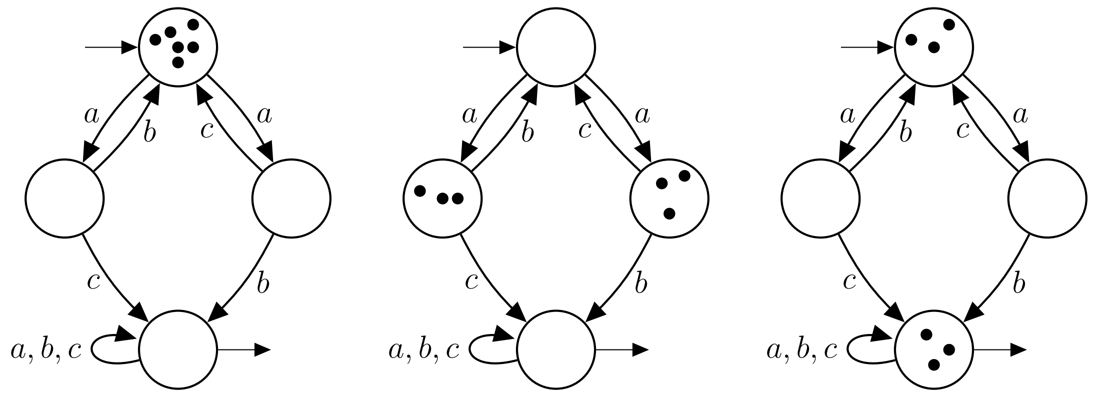


Figure B.1: Example of a population game where Laetitia wins no matter the number of tokens. She plays a, Terrence splits the tokens in two, and Laetitia picks b or c to send the larger half to the final state. Iterating this strategy lets her win in $O(log(N))$ steps.

Since this model was meant to represent populations of yeasts, the adversarial setting seemed too restrictive: it is unlikely that yeasts would apply a complex strategy to avoid synchronisation. A natural question arose: what if Terrence simply picks the transition taken by each token at random, independently? Then we want to know if Laetitia can win with probability 1 against any number of tokens? This game is quite different as Laetitia can now repeat sequences of actions many times until tokens behave a certain way.

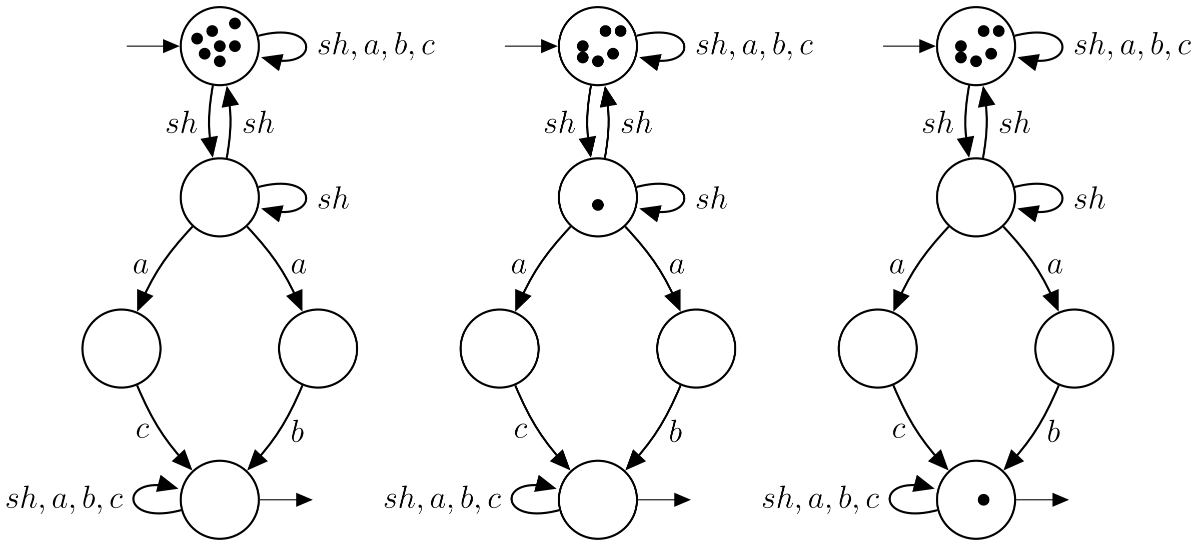


Figure B.2: Example of a randomised population game where Laetitia wins. She plays sh until exactly one token is in the second state (which happens eventually with probability 1), and then plays a and b or c to send that token to the final state. This is the only way to progress: if she plays a while more than one tokens are in the second state, then tokens may be sent to the left and right states, and she is stuck. Iterating this strategy lets her win in $2^{O(N)}$ steps, which is optimal.

In 2019, this was the subject of a paper by Colcombet, Fijalkow and Ohlmann. They showed that the problem is decidable, by translating it into a two-player game which they solve using well quasi-orders and distance automata [**ColcombetFO20**]. However, the use of well quasi-orders induces a non-elementary (maybe even Ackermanian) complexity. Meanwhile, we showed with Mahsa Shirmohammadi and Patrick Totzke that the problem is ExpTime-hard [**MascleST19**].

We made various attempts to close the complexity gap at the time, making several conjectures on the form of winning strategies, which were all eventually disproved (with increasingly large counter-examples).

We started working again on this problem in 2022, during a visit to Liverpool, with Hugo Gimbert and Patrick Totzke. We then found a new structure for strategies, in which we try to go to the final configuration by moving around large crowds of tokens and isolated tokens. If we get unlucky and move away from the path, we try to recover the isolated tokens by bringing them back into a crowd. Once they are recovered, we attempt again a lucky path. This new strategy shape was very convincing, but it was only over a year later that we were able to prove it, thanks to a key probabilistic argument from Hugo. From what we have so far, it appears that the problem is in fact ExpTime-complete, closing this line of research.

Another natural question, suggested by Blaise Genest, is the *speed of victory*: since Laetitia may require exponential expected time in the number of tokens, we may want to look for more efficient strategies. From the examples of games that we have, it seems that we can classify NFAs for which Laetitia wins in four categories:

- The ones that can be won in constant time. It is not hard to prove that those are exactly the ones for which there is a word w such that every path reading w from the initial state ends in the final state.
- The ones that require $log(N)^{O(1)}$ steps.
- The ones that require $N^{O(1)}$ steps.
- The ones that require $2^{N^{O(1)}}$ steps.

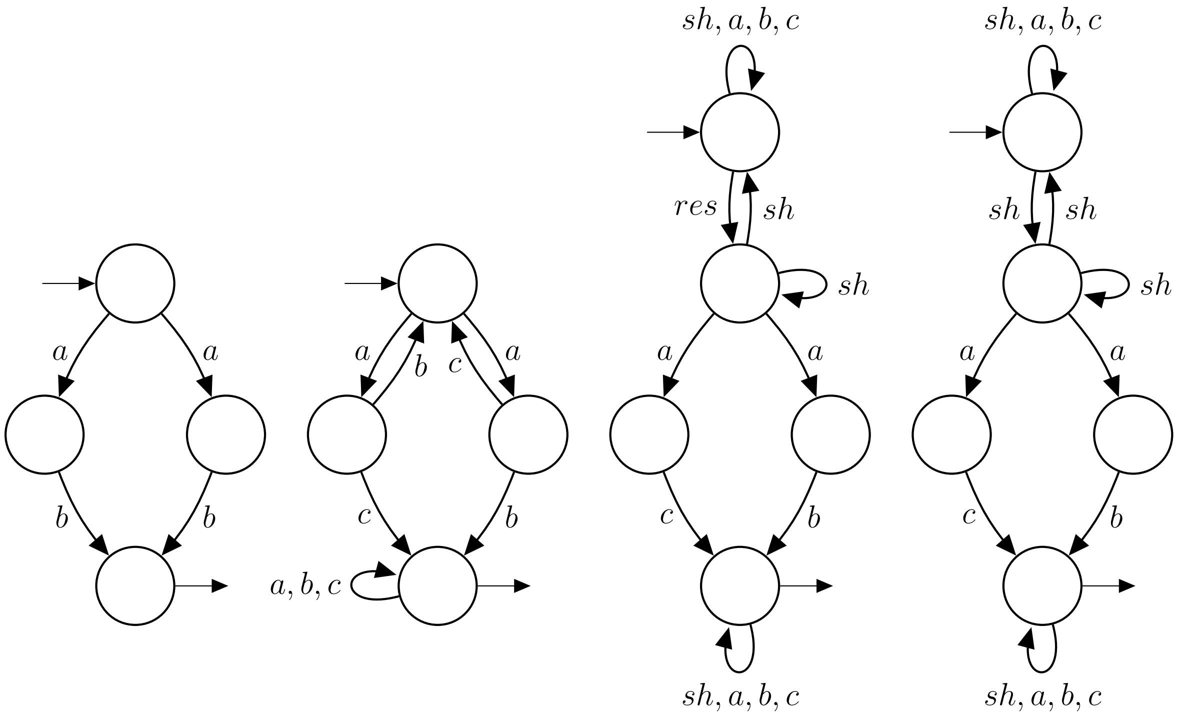


Figure B.3: Four randomised population games won by Laetitia. Sink states are omitted. The first one requires $O(1)$ steps, as the word ab synchronises all tokens to the final state. The second one requires $O(log(N))$ steps. The third one takes $O(Nlog(N))$ steps: play res and then sh until there is one token left in the second state (if all tokens leave at once before one remains, play res to reset). After $O(log(N))$ steps, one token remains in the second state, and we can play a and b or c to take it to the final state. We need to do this for each token, hence $O(Nlog(N))$ steps. The last one takes $2^{O(N)}$ steps.

Conjecture B.1

Let $\mathcal{A}$ be an NFA. Consider the associated randomised population game. Let $f : \mathbb{N} \to \mathbb{N}$ be such that $f(N)$ is the optimal winning time for Laetitia against N tokens. Then one of the following properties holds:

- $f \in O(1)$
- $f = log(N)^{O(1)}$ and $f = \Omega(log(N))$
- $f = N^{O(1)}$ and $f = \Omega(N^\alpha)$ for some $\alpha > 0$
- $f = 2^{N^{O(1)}}$ and $f = 2^{\Omega(N)}$
- $f(N) = +\infty$ for some N.

Furthermore this classification is decidable.

We also conjecture that a similar classification exists in the adversarial setting, with NFAs requiring either constant, polylogarithmic or polynomial time. It was shown in [**BertrandDGGG19**] that whenever Laetitia wins she has a polynomial-time strategy. Take a look at Figure B.4. The intuition is that in the second game, when tokens split, Laetitia can choose the part of tokens that makes progress. By contrast, in the third game she can make part of the tokens progress, but she does not pick that part.

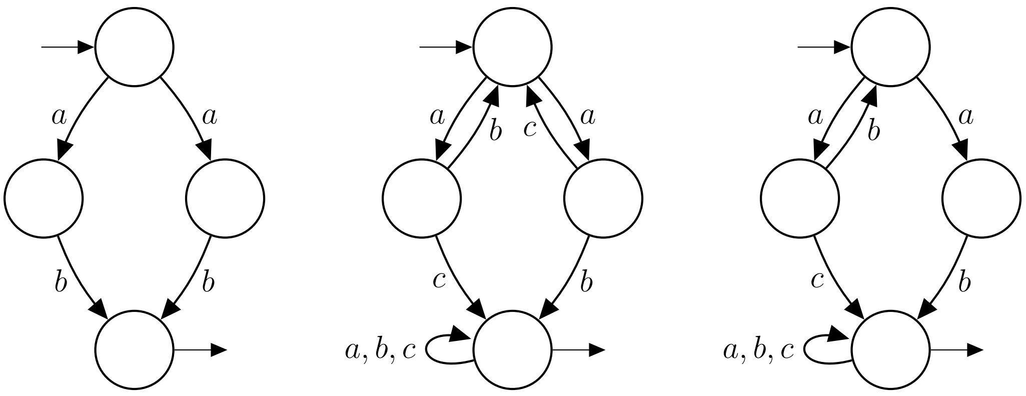


Figure B.4: Three population games won by Laetitia. The first one requires $O(1)$ steps, as the word ab synchronises all tokens to the final state. The second one requires $O(log(N))$ steps. The third one takes $O(N)$ steps: play a, if all tokens go left play c, if some go right play b. In the worst case this takes $2N$ steps.

Conjecture B.2

Let $\mathcal{A}$ be an NFA. Consider the associated adversarial population game. Let $f : \mathbb{N} \to \mathbb{N}$ be such that $f(N)$ is the optimal winning time for Laetitia against N tokens. Then one of the following properties holds:

- $f \in O(1)$
- $f = log(N)^{O(1)}$ and $f = \Omega(log(N))$
- $f = N^{O(1)}$ and $f = \Omega(N)$
- $f(N) = +\infty$ for some N.

Furthermore this classification is decidable.

Explorable automata Hazard and Kuperberg introduced *explorable automata*, which can be seen as an extension of both history-deterministic automata and population games [**HazardK23**][1]. Consider an automaton on infinite words, say a parity automaton. An automaton is N-explorable if Terrence wins the following game:

- N tokens are placed on the initial state (we can assume that there is only one)
- at each turn, Laetitia picks a letter x and Terrence moves each token along an x-transition
- Laetitia wins if:
  - the sequence of letters she picked is in the language of the automaton, and
  - for all tokens, the sequence of transition taken by that token is not an accepting run.

An automaton is explorable if it is N-explorable for some N.

Hazard and Kuperberg showed that explorability is decidable for [0,2]-automata, and that the explorability for [i,j]-automata reduces to explorability for [1,3]-automata for all i, j such that $j - i > 2$. Solving the [1,3] case would thus yield a complete picture.

They also defined ω-explorability: an automaton id ω-explorable if Terrence wins the game above with an infinite countable set of tokens. The ω-explorability property is decidable for [0,1]-automata (co-Büchi), and the explorability for [i,j]-automata reduces to explorability for [1,2]-automata for all i, j such that $j - i > 1$. Here, solving the [1,2] case would thus yield a complete picture.

Open problem B.3

Is explorability decidable for [1,3]-parity automata?

Open problem B.4

Is ω-explorability decidable for Büchi automata?

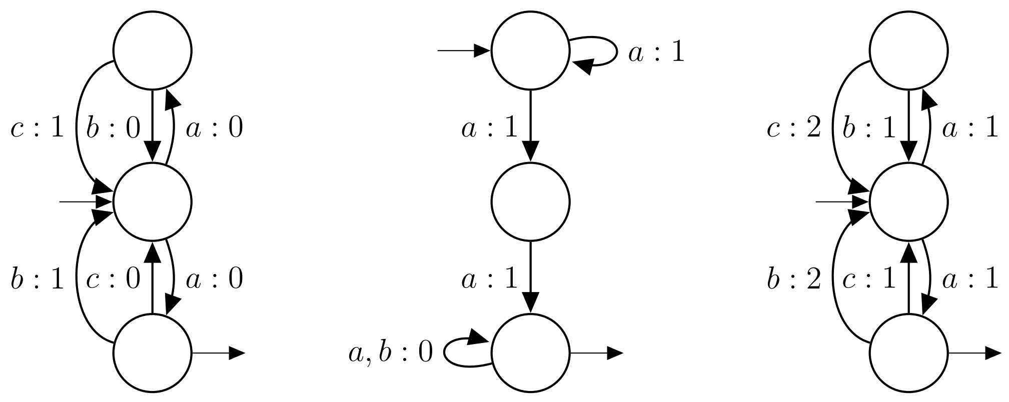


Figure B.5: The left automaton is 2-explorable, the middle one is not explorable but is ω-explorable, the third one is not ω-explorable. These examples are taken from [**HazardK23**].

Another open problem concerns the strategies for Terrence: In general, in population games, we allow him to have a global view of the system. If we want to strengthen the link to distributed synthesis, we could allow only local strategies. Of course, if all tokens follow the same deterministic strategy, then the game is trivial. However, if we allow randomisation, then we obtain an intermediate case between the fully-randomised setting and the adversarial one with full view of the configuration.

[1]The mentioned results do not all appear in the cited paper, some of them are from personal communication and a presentation, see here

Other projects

During my PhD I worked on several projects that are not included in this manuscript. Here is a list of the most fruitful ones.

Simplifying ω-regular conditions with the Alternating Cycle Decomposition In a joint work with Antonio Casares, following his PhD thesis [**CasaresThesis23**], we investigated the complexity of simplifying the acceptance condition of various automata classes, such as Muller, Rabin or parity, without changing their structure [**CasaresM24**]. We studied two modes of simplification: one is replacing conditions with ones from a more restricted class (e.g. Muller conditions by parity ones), the other is replacing a condition with one that uses less colours. We considered those problems in two versions: either we try to find a simpler condition that is equivalent on all automata, or we try to find one that is equivalent on a given automaton structure. Our conclusions were that if we do not take into account the automaton structure all those problems can be solved in P. In the case where we are given an automaton structure, we can simplify the condition type in polynomial time using the Alternating Cycle Decomposition (ACD), which is an object introduced by Casares, Colcombet and Fijalkow in 2021 [**CasaresCF21**]. We showed that the ACD of a Muller automaton can be computed in polynomial time. By contrast, we showed that reducing the number of colours used on a given automaton is NP-hard for Rabin and Muller conditions.

Minimisation of (history-)deterministic (co-)Büchi automata An automaton is history-deterministic (HD) if its non-determinism can be resolved by a controller based only on the sequence of transitions seen so far. In 2019, Abu Radi and Kupferman showed that transition-based HD coBüchi automata can be minimised in polynomial time [**RadiK22**]. In a joint work with Antonio Casares, Denis Kuperberg, Olivier Idir and Aditya Prakash (conducted in part at Highlights 2023), we extended the result of Abu Radi and Kupferman to HD generalised coBüchi automata [**CasaresIKMP2024arxiv**]. On the other hand, we showed that state minimisation is NP-complete for (history-)deterministic generalised Büchi automata and deterministic coBüchi automata. This helps us understand the difficulties in minimising ω-automata.

LTL learning In 2021, Fijalkow and Lagarde started to investigate the complexity of the following problem: given two finite sets of words P and N and a bound $k \in \mathbb{N}$, is there an LTL formula of size at most k that is satisfied by all words of P and no word of N? They showed that the problem is NP-hard in some cases, and gave tight bounds on the polynomial-time approximations in the case where one only has X and $\wedge$

operators [**FijalkowL21**]. We later extended their results by showing that this problem is NP-complete and hard to approximate in almost all fragments of LTL, even over a fixed alphabet. While this result is not so surprising, the techniques we use to prove it lead us to understand much better the power of separation of many fragments of LTL [**MascleFL23**].

Immediate-observation population protocols with unordered data Population protocols are a model of distributed computation in which a crowd of anonymous finite-state agents communicate via pairwise interactions. Together they decide whether their initial configuration, i. e., the initial distribution of agents in each state, satisfies a property. In 2023, Blondin and Ladouceur extended this model by introducing Population Protocols with Unordered Data (PPUD) [**BlondinL23**]. In PPUD, each agent carries a fixed data value, and the interactions between agents depend on the equality of their data. Blondin and Ladouceur also identified the interesting subclass of Immediate Observation PPUD (IOPPUD), where in every transition one of the two agents remains passive and does not move, and they characterised its expressive power. In a collaboration with Steffen van Bergerem, Roland Gutenberg, Sandra Kiefer, Nicolas Waldburger and Chana Weil-Kennedy, following the Autobóz 2023 workshop, we studied the decidability and complexity of formally verifying these protocols [**BergeremGKMWW24**]. We showed that checking if a PPUD is well-specified, i. e., whether it correctly computes some function, is undecidable in general. By contrast, for IOPPUD, we exhibited a large yet natural class of problems, which includes well-specification among other classic problems, and establish that these problems are decidable in ExpSpace.

A trichotomy on the number of necessary samples needed for property-testing regular languages Property testing is a class of problems where we look for very efficient algorithms that distinguish objects with a given property from ones which are very far from having it. More precisely, we consider a class of structures (for instance graphs) equipped with a distance (for instance the edit distance) and a property (for instance being bipartite). We fix a threshold $\varepsilon \in [0, 1[$ and search for an algorithm that, given an object of size n, answers yes if that object has the property, and answers no with probability at least $\frac{1}{3}$ if that object is at distance $> \varepsilon n$ from all objects with that property. In 2001, Alon, Krivelevich, Newman, and Szegedy showed that all regular languages could be tested using $O(\varepsilon^{-1} log^3(1/\varepsilon))$ samples, with the Hamming distance [**AlonKNS00**]. Those bounds were later improved and generalised to other distances, such as the edit distance. In 2021, Bathie and Starikovskaya provided an algorithm using only $O(\varepsilon^{-1} log(1/\varepsilon))$ samples with the edit distance, and showed that for some languages this could not be improved [**BathieS21**].

In a collaboration with Gabriel Bathie and Nathanaël Fijalkow, we built upon these results to show that, for the Hamming distance, every regular language falls into one of three categories: those testable with $\Theta(1)$ samples, those testable with $\Theta(\varepsilon^{-1})$ samples, and those testable with $\Theta(\varepsilon^{-1} log(1/\varepsilon))$. Given an NFA, it is PSPACE-complete to decide which class it belongs to.