U S PC
Université Sorbonne Paris Cité

université PARIS DIDEROT PARIS 7

THÈSE DE DOCTORAT DE L'UNIVERSITÉ SORBONNE PARIS CITÉ

PRÉPARÉE À L'UNIVERSITÉ PARIS DIDEROT

ÉCOLE DOCTORALE DE SCIENCES MATHÉMATIQUES DE PARIS-CENTRE - ED 386

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

LABORATOIRE D'INFORMATIQUE DE PARIS NORD

On Model-checking Pushdown System Models

Vérification de modèles de systèmes à pile

THÈSE DE DOCTORAT D'INFORMATIQUE

Par ADRIEN POMMELLET

Dirigée par TAYSSIR TOUILI

SOUTENUE PUBLIQUEMENT LE 5 JUILLET 2018 À VILLETANEUSE

Jury

Ahmed Bouajjani	Examinateur	Prof.	Université Paris Diderot
Didier Caucal	Président	Prof.	Université Paris-Est Marne-la-Vallée
Javier Esparza	Examinateur	Prof.	Technische Universität München
Jérôme Leroux	Rapporteur	DR	Université de Bordeaux
Stefan Schwoon	Rapporteur	MdC	ENS Paris-Saclay
Tayssir Touili	Directrice	DR	Université Paris 13
Tomáš Vojnar	Examinateur	Prof.	Brno University of Technology

Présentation de la thèse

Cette thèse introduit différentes méthodes de vérification (ou *model-checking*) sur des modèles de systèmes à pile. En effet, les *systèmes à pile* (*pushdown systems*) modélisent naturellement les programmes séquentiels grâce à une pile infinie qui peut simuler la *pile d'appel* du logiciel.

La première partie de cette thèse se concentre sur la vérification sur des systèmes à pile de la logique HyperLTL, qui enrichit la logique temporelle LTL de quantificateurs universels et existentiels sur des variables de chemin. Il a été prouvé que le problème de la vérification de la logique HyperLTL sur des systèmes d'états finis est décidable ; nous montrons que ce problème est en revanche indécidable pour les systèmes à pile ainsi que pour la sous-classe des *systèmes à pile visibles* (*visibly pushdown systems*). Nous introduisons donc des algorithmes d'approximation de ce problème, que nous appliquons ensuite à la vérification de politiques de sécurité.

Dans la seconde partie de cette thèse, dans la mesure où la représentation de la pile d'appel par les systèmes à pile est approximative, nous introduisons les *systèmes à surpile* (*pushdown systems with an upper stack*) ; dans ce modèle, les symboles retirés de la pile d'appel persistent dans la zone mémoire au dessus du pointeur de pile, et peuvent être plus tard écrasés par des appels sur la pile. Nous montrons que les ensembles de successeurs $post^*$ et de prédécesseurs pre^* d'un ensemble régulier de configurations ne sont pas réguliers pour ce modèle, mais que $post^*$ est toutefois contextuel (*context-sensitive*), et que l'on peut ainsi décider de l'accessibilité d'une configuration. Nous introduisons donc des algorithmes de sur-approximation de $post^*$ et de sous-approximation de pre^*, que nous appliquons à la détection de débordements de pile et de manipulations nuisibles du pointeur de pile.

Enfin, dans le but d'analyser des programmes avec plusieurs fils d'exécution, nous introduisons le modèle des *réseaux à piles dynamiques synchronisés* (*synchronized dynamic pushdown networks*), que l'on peut voir comme un réseau de systèmes à pile capables d'effectuer des changements d'états synchronisés, de créer de nouveaux systèmes à piles, et d'effectuer des actions internes sur leur pile. Le problème de l'accessibilité étant naturellement indécidable pour un tel modèle, nous calculons une abstraction des chemins d'exécutions entre deux ensembles

réguliers de configurations. Nous appliquons ensuite cette méthode à un processus itératif de raffinement des abstractions.

Mots-clés

— vérification

— systèmes à pile

— automates à pile

— LTL

— hyper-propriétés

— automates à surpile

— pointeur de pile

— réseaux d'automates à pile synchronisés

— abstraction de Kleene

Abstract

In this thesis, we propose different model-checking techniques for pushdown system models. *Pushdown systems* (PDSs) are indeed known to be a natural model for sequential programs, as they feature an unbounded stack that can simulate the assembly *stack* of an actual program.

Our first contribution consists in model-checking the logic HyperLTL that adds existential and universal quantifiers on path variables to LTL against pushdown systems (PDSs). The model-checking problem of HyperLTL has been shown to be decidable for finite state systems. We prove that this result does not hold for pushdown systems nor for the subclass of *visibly pushdown systems*. Therefore, we introduce approximation algorithms for the model-checking problem, and show how these can be used to check security policies.

In the second part of this thesis, as pushdown systems can fail to accurately represent the way an assembly stack actually operates, we introduce *pushdown systems with an upper stack* (UPDSs), a model where symbols popped from the stack are not destroyed but instead remain just above its top, and may be overwritten by later push rules. We prove that the sets of successors $post^*$ and predecessors pre^* of a regular set of configurations of such a system are not always regular, but that $post^*$ is context-sensitive, hence, we can decide whether a single configuration is forward reachable or not. We then present methods to overapproximate $post^*$ and under-approximate pre^*. Finally, we show how these approximations can be used to detect stack overflows and stack pointer manipulations with malicious intent.

Finally, in order to analyse multi-threaded programs, we introduce in this thesis a model called *synchronized dynamic pushdown networks* (SDPNs) that can be seen as a network of pushdown processes executing synchronized transitions, spawning new pushdown processes, and performing internal pushdown actions. The reachability problem for this model is obviously undecidable. Therefore, we compute an abstraction of the execution paths between two regular sets of configurations. We then apply this abstraction framework to a iterative abstraction refinement scheme.

Keywords

— model-checking

— pushdown systems

— pushdown automata

— LTL

— hyper-properties

— pushdown systems with an upper stack

— stack pointer

— synchronized dynamic pushdown networks

— Kleene abstraction

Résumé de la thèse

De la nécessité de la vérification automatique

Si les systèmes purement mécaniques souffrent de pannes parfois imprévisibles causées par l'usure et le vieillissement du matériel, l'utilisation de logiciels introduit la possibilité d'erreurs systématiques. Une seule faute au sein d'un système industriel complexe peut alors avoir des conséquences catastrophiques : une erreur de conversion d'un nombre flottant codé sur 64 bits vers un entier relatif codé sur 16 bits a provoqué en 1996 l'autodestruction de la fusée Arianne V, causant près de 500 millions de dollars de pertes.

La complexité toujours croissante des logiciels rend l'identification de telles erreurs de plus en plus compliquée. Les systèmes embarqués les plus récents utilisent plusieurs millions de lignes de code exécutées simultanément par des centaines de processeurs. Il est donc essentiel de concevoir des méthodes d'analyse des programmes qui soient fiables et efficaces.

Les techniques d'analyse *dynamique* reposent sur une étude du comportement d'un programme lors de son exécution. Les tests en sont l'une des formes les plus courantes et les plus simples : on compare le résultat de l'exécution d'un programme sur un jeu de données précis avec un résultat attendu. Afin d'empêcher un logiciel nuisible ou mal conçu d'endommager le système effectuant l'analyse, il est courant d'utiliser un environnement émulé.

On peut par exemple détecter des fuites de mémoire avec une telle méthode. Son efficacité toutefois dépend de l'adéquation du jeu de données de test utilisé: il n'est en particulier pas possible d'analyser l'exécution d'un programme ayant une infinité d'entrées possibles.

L'analyse *statique* s'effectue en revanche sans exécuter la moindre ligne de code, par une étude du code source ou du code machine si ce dernier n'est pas disponible. Idéalement, de telles méthodes doivent prendre en compte la sémantique du programme, et ne pas se résumer à une simple recherche de motifs syntaxiques.

Il est hélas impossible de toujours déterminer si un programme quelconque ayant la puissance de calcul d'une machine de Turing vérifie

une spécification précise telle que l'absence d'erreurs lors de l'exécution : un tel problème est indécidable, par une simple réduction au problème de l'arrêt.

Pour cette raison, le *model-checking* (ou vérification automatique) s'est imposé comme l'une des pierres angulaires des méthodes modernes d'analyse statique. Dans ce cadre, le programme est représenté par un *modèle* mathématique abstrait qui approxime le comportement du programme au prix de son incomplétude : toute propriété vérifiée par le programme ne l'est pas nécessairement par le modèle, mais ce dernier peut malgré tout s'avérer utile s'il ne vérifie que des propriétés qui sont également vraies pour le programme original.

Les comportements interdits et les propriétés désirables du programme sont ensuite exprimées par des formules dans un cadre *logique* clairement défini, puis l'on vérifie si elles s'appliquent bien au modèle abstrait du programme.

La logique temporelle

Le model-checking repose ainsi sur deux éléments: un modèle formel du programme assez précis pour représenter fidèlement certains aspects de son comportement tels que la récursion ou la propagation de variables, et une spécification précise des propriétés de sûreté ou de sécurité que l'on souhaite vérifier.

La logique temporelle a été introduite par Amir Pnueli dans [Pnu77]. Ce terme fait référence à différents systèmes de règles symboliques pour représenter et raisonner sur des propositions qualifiées en termes de temps. Pnueli a en particulier conçu la *logique temporelle linaire*, plus connue sous le nom de LTL, et montré comment elle pouvait être vérifiée sur des automates d'états finis. Les formules LTL représentent des propriétés sur le futur des chemins d'exécution, c'est-à-dire les suites de configuration que le modèle peut parcourir. LTL peut par exemple servir à représenter des propriétés de vivacité ou de sûreté.

Clarke et al. ont plus tard introduit et vérifié sur des automates d'états finis la *logique du temps arborescent* CTL. Cette logique s'applique a des arbres d'exécution, et peut imposer simultanément une contrainte à de multiples chemins d'exécution partant d'une même configuration. Certaines propriétés peuvent être exprimées par CTL mais pas par LTL, et vice-versa.

Le model-checking de LTL et de CTL a ainsi été le sujet de multiples études depuis 25 ans [GV08,Var96,KYV01,QS82].

Une formule LTL ne quantifie toutefois qu'un seul chemin d'exécution du système : ni LTL, ni CTL ne peuvent exprimer de propriétés sur de multiples traces simultanées et synchronisées d'un même programme. De tels propriétés sont appelées *hyperpropriétés*, et peuvent servir à exprimer de nombreuses règles de sécurité ou de sûreté, en particulier dans le cadre de l'analyse de flux d'informations. Ainsi, la règle de *non-interférence* impose de ne pas pouvoir distinguer les sorties publiques de deux exécutions ayant les mêmes variables publiques d'entrée, quand bien même leurs entrées privées diffèrent. Une telle relation entre deux exécutions du même modèle ne peut être exprimée par une simple formule LTL.

HyperLTL est une logique introduite par Clarkson et al. dans [CFK+14] qui étend LTL en autorisant la quantification universelle et existentielle de multiples *variables de chemin* évoluant dans l'ensemble des traces d'exécution du système, permettant ainsi l'expression d'hyperpropriétés. Par exemple, la formule $\forall \pi_1, \forall \pi_2, (a_{\pi_1} \wedge a_{\pi_2}) \Rightarrow X((b_{\pi_1} \wedge b_{\pi_2}) \vee (c_{\pi_1} \wedge c_{\pi_2}))$ signifie que pour deux variables de chemin π_1 et π_2 dans l'ensemble des traces d'exécution infinies d'un système, si π_1 et π_2 vérifient a à une étape donnée, alors elles doivent vérifier b ou c à l'étape suivante.

Clarkson et al. ont montré que le problème $S \models \psi$ du model-checking d'HyperLTL, c'est-à-dire vérifier si les traces d'un système S vérifient la formule HyperLTL ψ, est décidable si S est un système de transitions fini (équivalent à un automate d'états finis). Différents algorithmes ont plus tard été présentés par Finkbeiner et al. dans [FRS15].

Toutefois, les automates d'états finis ne peuvent simuler les programmes effectuant un nombre non-borné d'appels récursifs à d'autres procédures. Il faut pour cela utiliser la classe des *automates à pile* (ou PDSs, pour *pushdown systems*). Un tel automate dispose d'une pile non bornée qui lui permet de simuler la pile d'appel d'un programme, où sont stockées les informations sur les procédures actives du programme telles que les adresses de retour, les paramètres, et les variables locales.

Comme indiqué précédemment, la logique temporelle permet souvent d'exprimer des propriétés de correction des programmes. Il est donc important de proposer des techniques de model-checking pour les automates à pile. Des algorithmes efficaces pour LTL ont été introduits dans [BEM97, EHRS00, Wal01]. La vérification de CTL sur les automates à pile a d'abord été proposée par Walukiewicz et al. dans [Wal00] ; une approche basée sur l'emploi d'automates a plus tard été utilisée par Song et al. dans [ST11]. La vérification de HyperLTL sur les automates à pile n'a toutefois jamais encore été considérée.

Vérification de la logique HyperLTL sur les automates à piles

Nous nous intéressons dans la première partie de cette thèse au problème du model-checking de la logique HyperLTL sur les PDSs.

Nous montrons que ce problème est hélas indécidable : l'ensemble des traces d'un PDS est un langage hors-contexte, et il est indécidable de déterminer si l'intersection de deux langages hors-contexte est vide ; on réduit le précédent problème à celui du model-checking en utilisant une formule HyperLTL permettant de synchroniser des traces.

Il est toutefois décidable de déterminer si l'intersection de langages *visiblement hors-contexte* est vide. Cette famille de langages est engendrée par les *automates visiblement à pile*, une sous-classe des automates à pile introduite par Alur et al. dans [AM04] et qui dépend de ses entrées : à chaque étape d'une exécution, la prochaine opération effectuée sur la pile dépend de la lettre du mot d'entrée lue, selon une partition de l'alphabet d'entrée. Nous étudions le problème du model-checking de HyperLTL sur les systèmes visiblement à pile (*visibly pushdown systems*, ou VPDSs) et prouvons son indécidabilité par une réduction au problème du vide sur les automates visiblement à deux piles, que l'on sait indécidable grâce aux travaux de Carotenuto et al. dans [CMP07].

Toutefois, il est décidable de déterminer si l'intersection d'un langage hors-contexte et d'un ensemble régulier est vide ; nous considérons donc le cas d'une formule dont seule une unique variable appartient à l'ensemble des traces $Traces_\omega(\mathcal{P})$ d'un PDS, les autres variables appartenant à une sur-approximation régulière $\alpha(Traces_\omega(\mathcal{P}))$. En utilisant une approche basée sur la théorie des automates, nous pouvons partiellement décider le problème du model-checking de formules dont toutes les variables sont universellement quantifiées à l'exception d'au plus une : si une telle formule est vérifiée en sur-approximant les traces, il en sera de même pour le système original. De même, en considérant des sous-approximations régulières des traces sauf pour une seule variable au plus, on peut partiellement vérifier une formule HyperLTL n'utilisant que des quantificateurs existentiels $\exists$: si la sous-approximation ne vérifie pas la formule, il en est de même du programme originel.

Nous montrons également que le problème du model-checking sur des PDSs de formules HyperLTL n'utilisant que des quantificateurs existentiels $\forall$ peut être approximé en effectuant une vérification à *phases bornées* d'une formule LTL sur un *automate multi-piles* ; une phase est une partie d'un chemin d'exécution durant laquelle on ne peut retirer les symboles que d'une seule pile au plus, comme définie par La Torre et al. dans [TMP07].

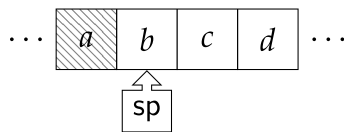

FIGURE 1: La pile originelle.

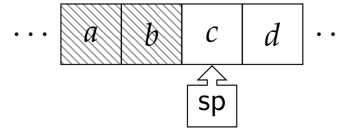

FIGURE 2: La pile après un *pop*.

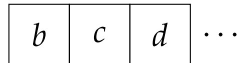

FIGURE 3: La pile originelle du PDS.

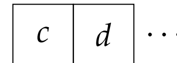

FIGURE 4: La pile du PDS après un *pop*.

Ces résultats ont été publiés dans [PT18].

Les automates à surpile

Dans la seconde partie de cette thèse, nous étendons le modèle des automates à pile de manière à prendre en compte la section de la mémoire localisée au-dessus de la pile.

Le but des PDSs est en effet de modéliser avec précision la *pile d'appel* du programme : il s'agit d'une structure de données sous forme de pile permettant de stocker les informations liées aux procédures actives du programme telles que les adresses de retour, les paramètres passés en argument, et les variables locales. Elle est généralement implémentée en utilisant un registre pour le *pointeur de pile (sp)* qui indique la position du sommet de la pile dans la mémoire.

Ainsi, si l'on considère que la pile s'étend vers des adresses mémoire décroissantes, quand l'on pose (*push*) un nouvel élément sur la pile, le pointeur *(sp)* est décrémenté avant d'écrire le nouvel élément. Par exemple, dans le cas de l'architecture $x86$, la valeur de *(sp)* est réduite de 4 (on ajoute 4 octets). Quand l'on retire (*pop*) un élément de la pile, *(sp)* est incrémenté, de 4 par exemple dans le cas de l'architecture $x86$.

Toutefois, dans un PDS, ni les *push*, ni les *pop* ne suivent vraiment le fonctionnement de la pile en assembleur. En effet, durant un vrai *pop*, l'élément retiré reste en mémoire et le pointeur de pile est simplement incrémenté, comme montré dans les Figures 1 et 2, tandis qu'un PDS se contente de détruire le sommet de la pile, comme montré dans les Figures 3 et 4

Cette différence subtile prend son importance si l'on veut analyser des programmes en assembleur manipulant directement le pointeur de

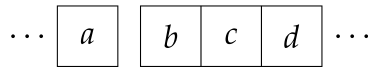

FIGURE 5: La pile originelle de l'UPDS.

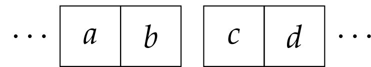

FIGURE 6: La pile de l'UPDS après un *pop*.

pile. En effet, il est possible de manipuler le registre contenant *sp*. Par exemple, l'instruction mov eax [sp − 4] permet de copier la valeur pointée par l'adresse sp − 4 dans le registre général eax. Dans la mesure où l'adresse sp − 4 est située après le pointeur de pile, il est impossible d'évaluer la valeur copiée vers eax sans une forme d'enregistrement des précédents éléments qui ont été retirés de la pile mais pas encore écrasés. De telles instructions peuvent être utilisées par des programmes nuisibles afin de dissimuler de manière inattendue leurs intentions aux méthodes d'analyse statique.

Il est donc important d'enregistrer la section de la mémoire située juste au-dessus du pointeur de pile. Nous étendons à cette fin les PDSs de manière à prendre en compte cette *surpile*: nous introduisons un nouveau modèle appelé *automate à surpile* (ou *pushdown system with an upper stack*, ou UPDS) qui étend la sémantique des PDSs. Dans un UPDS, quand l'on retire un symbole du sommet de la pile (que l'on précisera *du dessous* à partir de maintenant), il est ajouté à la base d'une surpile en lecture seule, ce qui permet de simuler la décrémentation du pointeur de pile, comme montré dans les Figures 5 et 6, où, après avoir été retiré de la pile du dessous (à droite), b est ajouté à la pile du dessus (à gauche) au lieu d'être simplement effacé. La surpile et la pile de dessous se rejoignent au niveau du pointeur de pile.

Nous prouvons les propriétés suivantes sur les UPDSs :

- l'ensemble des prédécesseurs et celui des successeurs d'un ensemble régulier de configurations ne sont pas toujours régulier ; celui des successeurs est toutefois hors-contexte ;
- l'ensemble des prédécesseurs est régulier si l'on impose une limite de k phases sur les exécutions, une phase étant une partie d'une exécution où l'on interdit soit les *push*, soit les *pop* ; c'est une sous-approximation du véritable ensemble ;
- on peut sur-approximer l'ensemble des successeurs à partir d'une abstraction des exécutions.

Nous montrons ensuite que les UPDSs et les approximations de leurs ensembles d'accessibilité décrites précédemment peuvent être utilisés pour localiser des erreurs et des brèches de sécurité dans des programmes.

Ces résultats ont été publiés dans [PDT17].

Les réseaux dynamiques à pile synchronisés

Dans la troisième partie de cette thèse, nous nous concentrons sur le problème du model-checking pour les programmes concurrents.

L'utilisation de programmes parallèles s'est popularisée ces quinze dernières années, mais ces derniers n'en restent pas moins souvent instables et vulnérables à des problèmes spécifiques tels que l'interblocage. Des méthodes d'analyse statique adaptées à de tels programmes sont donc plus nécessaires que jamais.

Comme précisé précédemment dans [EHRS00], les automates à pile sont un modèle naturel pour les programmes séquentiels utilisant des appels récursifs à des procédures. Il est donc naturel de vouloir modéliser un programmes concurrent par un réseau de systèmes à pile, chaque PDS modélisant l'une des composantes séquentielles du tout. C'est dans ce contexte que Bouajjani et al. ont introduit dans [BMOT05] les *réseaux dynamiques à pile* (*dynamic pushdown networks*, ou DPNs).

Intuitivement, cette famille d'automates est constituée d'un réseau de systèmes à piles exécutés indépendamment en parallèle. Chaque membre d'un DPN peut, après une transition, engendrer un nouveau PDS qui devient lui aussi membre du réseau. Les DPNs permettent ainsi de représenter un réseau de fils d'exécution, chaque fil pouvant appeler des procédures, effectuer des actions internes, et engendrer un nouveau fil.

Toutefois, ce modèle ne peut représenter la synchronisation de multiples fils ou composants parallèles. Afin de modéliser la communication inter-fils, Bouajjani et al. ont introduit dans [BET03] les *systèmes à pile communiquant*(*communicating pushdown systems*, ou CPDSs), que l'on peut présenter comme un tuple de systèmes à pile se synchronisant par rendez-vous sur leurs chemins d'exécution. Toutefois, les CPDSs ne disposent que d'un nombre constant de processus, et ne peuvent donc gérer la création dynamique de nouveaux fils.

Nous introduisons donc un modèle plus précis appelé *réseau dynamique à pile synchronisé* (*synchronized dynamic pushdown networks*, ou SDPNs) qui introduit dans les DPNs un mode de synchronisation par rendez-vous afin de gérer en même temps la communication inter-fils et la création dynamique de fils.

Un SDPN peut être vu comme un DPN dont les processus modélisés par des PDSs peuvent se synchroniser par rendez-vous en envoyant et recevant des messages : il est possible d'effectuer des actions internes

étiquetées par une lettre τ sans se synchroniser, comme dans un DPN, mais aussi de se synchroniser via des canaux dédiés.

Pour ce faire, chaque canal est représentée par une paire de lettres a et $\overline{a}$ qui étiquettent certaines transitions. Si un fil d'exécution peut effectuer une action étiquetée par a, et un autre fil, une action étiquetée par $\overline{a}$, alors les deux fils peuvent se synchroniser et effectuer simultanément leurs actions en une seule étape que l'on étiquettera par τ.

Nous considérons ensuite le problème de l'accessibilité pour la classe des SDPNs, c'est-à-dire, déterminer si une configuration critique peut être atteinte depuis les configurations initiales du programme. Il est équivalent de déterminer si l'ensemble $Paths(\mathcal{C}, \mathcal{C}')$ des chemins entre deux ensembles de configurations C et C' est vide ou non. Ce problème est hélas indécidable pour tous les modèles d'automates à pile synchronisés, comme démontré par Ramalingam dans [Ram00].

On ne peut donc calculer l'ensemble $Paths(\mathcal{C}, \mathcal{C}')$ des chemins d'exécution de manière exacte. Nous appliquons une méthode similaire à celle détaillée dans [BET03] pour surmonter ce problème : nous calculons une abstraction $\alpha(Paths(\mathcal{C}, \mathcal{C}'))$ des chemins. Pour ce faire, nous utilisons des techniques basées sur :

— la représentation des ensembles réguliers de configurations de SDPNs par des automates d'états finis ;

— l'utilisation de ces automates pour déterminer un ensemble de contraintes dont le plus petit point fixe permet de caractériser les chemins d'exécution du programme ; afin de calculer ces contrainte, (1) nous considérons une sémantique relâché sur les SDPNs qui autorise la synchronisation partielle de chemins, (2) nous approximons les ensembles de chemin d'exécution par des fonctions sur des algèbres de Kleene, et (3) nous définissons un produit de mélange (*shuffle product*) sur les chemins afin de représenter l'entrelacement des fils d'exécution et leur synchronisation potentielle ;

— la résolution de ces contraintes dans un domaine abstrait ; nous considérons en particulier le cas d'un domaine fini ; cet ensemble de contraintes peut alors être résolu par un calcul itératif de point fixe.

Notons que les principales contributions de cette approche par rapport aux méthodes définies dans [BET03, Tou05] sont l'emploi de fonctions pour représenter les ensembles de chemins approximés et la définition d'un produit de mélange adapté à l'entrelacement de fils. Le cadre théorique de ces articles est en effet insuffisant pour gérer le cas de la création dynamique de nouveaux fils, d'où ces ajouts.

Nous pouvons ensuite appliquer cette approximation du problème de l'accessibilité à une procédure *d'abstraction raffinée par itération* inspirée des travaux de Chaki et al. dans [CCK+06]. Notre idée est la suivante : (1) nous effectuons une analyse d'accessibilité d'un programme modélisé par un SDPN en utilisant une abstraction finie d'ordre n ; si l'ensemble cible de configuration n'est pas accessible par un chemin abstrait, il n'est également pas accessible dans le programme original ; dans le cas contraire, on dispose d'un contre-exemple ; (2) on vérifie si l'on peut associer ce contre-exemple à une véritable exécution du programme ; (3) si c'est le cas, l'ensemble cible est bel et bien accessible ; (4) dans le cas contraire, nous raffinons l'abstraction en utilisant un domaine fini d'ordre $n + 1$ à l'étape (1). Cette procédure est utilisée pour détecter un erreur dans un driver Windows, de manière semblable à [QW04].

Ces résultats ont été publiés dans [PT17].

Plan de la thèse

L'objectif du Chapitre 2 est de rappeler au lecteur des résultats importants sur les *automates à pile*, leurs propriétés d'accessibilité, et les algorithmes de *model-checking* utiles au reste de cette thèse. Dans le Chapitre 3, nous nous concentrons sur le *model-checking* de la logique HyperLTL sur les automates à pile. Nous introduisons dans le Chapitre 4 une nouvelle classe d'automates appelée *automates à surpile* dont nous étudions le problème de l'accessibilité. Nous présentons ensuite dans le Chapitre 5 un modèle concurrent combinant création dynamique de fil et synchronisation inter-fils appelé *réseau dynamique à pile synchronisé* conçu de manière à pouvoir calculer efficacement une abstraction des chemins d'exécution grâce au cadre théorique des algèbres de Kleene. Nous présentons enfin notre conclusion dans le chapitre 6.

Contents

Acknowledgements xix

1 Introduction 1
- 1.1 The need for model-checking . . . 1
- 1.2 Temporal logics . . . 2
- 1.3 Model-checking the logic HyperLTL for pushdown systems . . . 4
- 1.4 Pushdown systems with an upper stack . . . 5
- 1.5 Synchronized dynamic pushdown networks . . . 6
- 1.6 Thesis outline . . . 9

2 Model-checking Pushdown Systems 11
- 2.1 Pushdown systems . . . 11
  - 2.1.1 The model . . . 11
  - 2.1.2 $\mathcal{P}$-automata . . . 12
  - 2.1.3 From a program to a pushdown system . . . 13
- 2.2 Reachability sets . . . 14
  - 2.2.1 Computing pre^* . . . 14
  - 2.2.2 Computing $post^*$ . . . 14
- 2.3 Model-checking LTL on pushdown systems . . . 15
  - 2.3.1 The linear-time temporal logic LTL . . . 15
  - 2.3.2 Büchi automata . . . 16
  - 2.3.3 The model-checking problem . . . 16

3 Model-checking HyperLTL for Pushdown Systems 19
- 3.1 Visibly pushdown systems . . . 20
- 3.2 HyperLTL . . . 20
  - 3.2.1 The logic . . . 20
  - 3.2.2 HyperLTL and PDSs . . . 22
  - 3.2.3 HyperLTL and VPDSs . . . 24
- 3.3 Model-checking constrained HyperLTL . . . 26
  - 3.3.1 Regular over-approximations of context-free languages . . . 28
  - 3.3.2 With one context-free variable and n regular variables . . . 28
  - 3.3.3 With one visibly pushdown variable and n regular variables . . . 30

3.4 Model-checking HyperLTL with bounded phases 31
3.4.1 Multi-stack pushdown systems 31
3.4.2 Application to HyperLTL model-checking 33
3.5 Applications to security properties 34
3.5.1 Observational determinism 34
3.5.2 Declassification 36
3.5.3 Non-inference 38
3.6 Related work 38
3.7 Conclusion 39

4 Reachability Analysis of Pushdown Systems with an Upper Stack 41
4.1 Pushdown systems with an upper stack 42
4.2 Reachability properties 44
4.2.1 *post** is not regular 44
4.2.2 *pre** is not regular 46
4.2.3 *post** is context-sensitive 47
4.3 Under-approximating *pre** 51
4.4 Over-approximating *post** 52
4.4.1 A relationship between runs and the upper stack 53
4.4.2 Computing an over-approximation 56
4.5 Applications 56
4.5.1 Stack overflow detection 57
4.5.2 Reading the upper stack 57
4.5.3 Changing the stack pointer 58
4.6 Related work 59
4.7 Conclusion 59

5 Static Analysis of Multi-threaded Recursive Programs Communicating via Rendez-vous 61
5.1 Synchronized dynamic pushdown networks 62
5.1.1 Dynamic pushdown networks 62
5.1.2 The model and its semantics 63
The strict semantics. 64
The relaxed semantics. 65
5.1.3 From a program to a SDPN model 66
5.2 The reachability problem 67
5.2.1 From the strict to the relaxed semantics 68
5.2.2 Representing infinite sets of configurations 68
5.3 Representing the set of paths 70
5.3.1 Π-configurations 70
5.3.2 The shuffle product 70
5.3.3 Π-automata 71
The transition relation. 71
5.4 Characterizing the set of paths 72
5.4.1 Computing *pre*$^*(M, C)$ 73

5.4.2 From $pre^*(M, C)$ to $pre^*_\Pi(M, C)$ 73
The need for functions. 74
The constraints. 76
The intuition. 76
5.4.3 Proof of Theorem 22 78
Proof of Lemma 15 79
Proof of Lemma 16 81
5.5 An abstraction framework for paths 84
5.5.1 Abstractions and Galois connections 85
5.5.2 Kleene algebras 85
5.5.3 Kleene abstractions 86
Prefix abstractions. 86
Suffix abstractions. 87
5.6 Abstracting the set of paths 88
5.6.1 From the language of paths to the Kleene abstraction 89
5.6.2 Computing $pre^*_K(M, C)$ 92
5.6.3 Finding the abstraction 93
5.7 Using our framework in a iterative abstraction refinement scheme 94
5.8 A case study 95
5.8.1 The program 95
5.8.2 From the driver to the SDPN model 96
The process COUNTER. 97
The process STOPPING-FLAG. 97
The process STOPPING-EVENT. 98
The process STOP-D. 98
The process REQUEST. 98
The process GEN-REQ. 99
The function Increment. 99
The function Decrement. 99
5.8.3 An erroneous execution path 100
5.9 Related work 102
5.10 Conclusion 103

6 Conclusion and Future Work **105**
6.1 A brief summary of this thesis 105
6.2 Future work 105
6.2.1 Tools for the model-checking of hyperproperties . 105
6.2.2 Further reachability analysis of UPDSs 106
6.2.3 Tools for the model-checking of concurrent programs 106
6.2.4 Abstract model-checking for SDPNs 107

Acknowledgements

This thesis would not have been possible without the help and goodwill of many people.

I would first like to thank my supervisor Prof. Tayssir Touili for her continuous support of my Ph.D study and related research. I could not have written this thesis without her guidance.

I would also like to thank the members of my thesis committee for their time and effort: Ahmed Bouajjani, Didier Caucal, Javier Esparza, Jérôme Leroux, Stefan Schwoon, and Tomáš Vojnar.

I am grateful to the whole *Vérification* team of the IRIF. I praise Arnaud Sangnier, Constantin Enea, and Germán Delbianco for their commitment to our frequent seminaries.

I would also like to express my gratitude to the *LoVe* team of the LIPN. Its weekly seminaries, organized by César Rodríguez and Étienne André, provided me with many opportunities to hone my presentation skills.

I am indebted to my previous internship advisors Prof. Christophe Prieur, Prof. Fabien Mathieu, and Prof. David Naccache. Without them, I am not sure I would have mustered the will to start this thesis.

I thank my fellow labmate Marcio Diaz for his help on the trickiest points of some proofs.

Last but not the least, I would like to thank my family as well as my fencing club for their support.

List of Figures

1 La pile originelle. ix
2 La pile après un *pop*. ix
3 La pile originelle du PDS. ix
4 La pile du PDS après un *pop*. ix
5 La pile originelle de l'UPDS. x
6 La pile de l'UPDS après un *pop*. x

1.1 The original stack. 5
1.2 The stack after one pop. 5
1.3 The original PDS stack. 5
1.4 The PDS stack after one pop. 5
1.5 The original UPDS stacks. 6
1.6 The UPDS stacks after one pop. 6

3.1 Checking observational determinism on the PDS $\mathcal{P}$ . . . 35
3.2 Checking declassification on the PDS $\mathcal{P}$ 37

4.1 Semantics of pop rules. 43
4.2 Semantics of push rules. 43
4.3 Using an under-approximation. 51
4.4 Using an over-approximation. 53
4.5 Using $\top$ to bound the upper stack. 57
4.6 The stack being read. 58
4.7 The original stack. 58
4.8 After changing sp. 58

5.1 Representing configurations of a DPN. 62
5.2 A DPN with 3 threads after a pop from T_2 and a push on T_3. 63
5.3 A DPN with 3 threads after thread T_1 spawns a new thread T_4. 63
5.4 Semantics of synchronized actions. 64
5.5 Semantics of internal actions. 65
5.6 Semantics of unsynchronized actions. 65
5.7 Accepting a regular set $p_1\gamma_1^+ p_2\gamma_2\gamma_3$ with an M-automaton. 69
5.8 Using labels in Π. 75
5.9 Using labels in Π^Π. 75
5.10 Case of a switch rule. 77
5.11 Case of a pop rule. 77

5.12 Case of a push rule. 78
5.13 Case of a spawn rule. 78
5.14 Adding an edge if a new process is spawned. 80
5.15 Adding an edge if no new process is spawned. 81
5.16 Applying a second order prefix abstraction to an example. 87
5.17 Applying a second order suffix abstraction to an example. 88

Chapter 1

Introduction

1.1 The need for model-checking

Systems based entirely on 'hardwired technology' tend to suffer so-called random failures, which are typically age or wear-related, but software-based systems fail predominantly due to systematic errors. A single mistake in such complex designs may bear catastrophic consequences: a conversion error from a 64-bit floating point number to a 16-bit signed integer caused in 1996 the self-destruction of the European Space Agency's Ariane 5 rocket that carried a payload worth $500 million.

As the complexity of software grows, identifying these errors becomes harder and harder. Modern systems can depend on millions of lines of code running on hundreds of networked processors. Designing sound and efficient program analysis methods is therefore a matter of the utmost importance.

Dynamic techniques can be used to analyse a program by observing its behaviour during its execution. The simplest form of dynamic analysis is testing: the program is executed on a given set of inputs and matched against an expected result. In order to prevent ill-designed or malicious software from damaging the system on which the analysis is performed, the code is often run in an emulated environment.

Errors such as memory leaks can be found while the program is being monitored. To be effective, this method must be executed with adequate test inputs in order to produce interesting behaviour. However, if the set of inputs is very large or possibly infinite, the program's execution can't be analysed in all possible circumstances.

By using *static* techniques, on the other hand, one attempts to analyse programs without running any piece of code. The analysis is instead performed the source code or, if it's not available, the object code.

Ideally, such techniques should not rely on simple syntactic pattern-matching methods only, but the semantics of the program must be taken into account as well.

Unfortunately, deciding whether an arbitrary program with the computational power of a Turing machine may violate a specification such as the absence of runtime errors is an undecidable problem. It can be proven by a straightforward reduction to the halting problem.

For this reason, the *model-checking* framework has proven to be a cornerstone of modern static analysis techniques. The program is represented as a simpler abstract mathematical *model* that approximates the behaviour of the actual system at the expense of incompleteness, as not every property true in the actual system is true of its abstraction. This model can nonetheless be sound if every property true of the abstract system can be mapped to a true property of the original program.

Desirable properties and forbidden behaviours are then expressed using a well-defined *logical* framework, then checked against the abstract mathematical model of the program.

1.2 Temporal logics

In order to use model-checking techniques, one should on the one hand compute a formal model of the program accurate enough to simulate parts of its behaviour deemed relevant, be it recursion, synchronization, or accurate variable propagation, and on the other hand use a specification that allows one to express unambiguously the safety or security properties one wants to check.

Temporal logics were pioneered by Amir Pnueli in [Pnu77]. This term refers to different systems of symbolic rules for representing and reasoning about propositions qualified in terms of time. Pnueli introduced the *linear-time temporal logic*, also known as LTL, and solved its model-checking problem for finite automata. LTL formulas encode properties about the future of execution paths, that is, the sequence of configurations the model goes through. It can be used to express safety and liveness properties.

Clarke et al. in [CE82] later introduced the *computation tree logic* CTL and a related model-checking framework on finite-state systems. CTL is a branching time logic: it applies to execution trees, and can simultaneously constrain several paths starting from a same configuration of the model. There are properties expressible in CTL and not in LTL, as well as properties expressible in LTL but not in CTL.

Thus, LTL and CTL model-checking techniques were extensively studied over the past 25 years [GV08, Var96, KYV01, QS82].

However, a LTL formula only quantifies a single execution trace of a system; neither LTL nor CTL can express properties on multiple, simultaneous, synchronized executions of a program. These properties on sets of execution traces are known as *hyperproperties*. Many safety and security policies can be expressed as hyperproperties; this is in particular true of information-flow analysis. As an example, the *non-interference* policy states that if two computations share the same public inputs, they should have identical public outputs as well, even if their private inputs differ. This property implies a relation between computations that can't be expressed as a simple LTL formula.

HyperLTL is a logic extending LTL introduced by Clarkson et al. in [CFK+14] that allows the universal and existential quantifications of multiple *path variables* that range over traces of a system in order to define hyperproperties. As an example, the formula $\forall \pi_1, \forall \pi_2, (a_{\pi_1} \wedge a_{\pi_2}) \Rightarrow X((b_{\pi_1} \wedge b_{\pi_2}) \vee (c_{\pi_1} \wedge c_{\pi_2}))$ means that, given two path variables π_1 and π_2 in the set of infinite traces of a system, if π_1 and π_2 verify the same atomic property a at a given step, then they should both verify either b or c at the next step.

Clarkson et al. have shown that the model-checking problem $S \models \psi$ of HyperLTL, that is, knowing if the set of traces of a system S verifies the HyperLTL formula ψ, can be solved when S is a finite state transition system (i.e. equivalent to a finite state automaton). Further model-checking techniques were later presented by Finkbeiner et al. in [FRS15].

However, the class of finite automata can't simulate programs with unbounded recursive calls. To this end, another class called *pushdown systems* (PDSs) is more relevant. A PDS features an unbounded stack that can simulate the *call stack* of an actual program. The call stack stores information about the active procedures of a program such as return addresses, passed parameters and local variables.

As mentioned previously, correctness properties of programs are often expressed with the unifying framework of temporal logics. It is therefore important to propose model-checking algorithms for PDSs. Efficient algorithms for model-checking LTL on PDSs were introduced in [BEM97, EHRS00, Wal01]. CTL model-checking methods for PDSs were first designed by Walukiewicz et al. in [Wal00]; an automata-theoretic approach was later presented by Song et al. in [ST11]. However, model-checking techniques for HyperLTL were never considered.

1.3 Model-checking the logic HyperLTL for pushdown systems

We consider in the first part of this thesis the model-checking problem of HyperLTL formulas against PDSs.

Unfortunately, we show that the model-checking problem of HyperLTL against PDSs is undecidable: the set of traces of a PDS is a context-free language, and deciding whether the intersection of two context-free languages is empty or not remains an undecidable problem that can be reduced to the model-checking problem by using a HyperLTL formula that synchronizes traces.

On the other hand, determining the emptiness of the intersection of two *visibly context-free languages* is decidable. This class of languages is generated by *visibly pushdown automata* (VPDA), an input-driven subclass of *pushdown automata* (PDA) first introduced by Alur et al. in [AM04]: at each step of a computation, the next stack operation will be determined by the input letter read, depending on a partition of the input alphabet. We study the model-checking problem of HyperLTL for *visibly pushdown systems* (VPDSs), and prove that it is also undecidable, by using a reduction of the emptiness problem for *two-stack visibly pushdown automata* (2-VPDA), which has been shown to be undecidable by Carotenuto et al. in [CMP07], to the model-checking problem.

To overcome these undecidability issues, since the emptiness of the intersection of a context-free langage with a regular set is decidable, one idea is to consider the case where only one path variable of the formula ranges over the set of traces $Traces_\omega(\mathcal{P})$ of a PDS or VPDS $\mathcal{P}$, while the other variables range over a regular abstraction $\alpha(Traces_\omega(\mathcal{P}))$. Using an automata-theoretic approach, we can approximate the model-checking problem of HyperLTL formulas that only use universal quantifiers $\forall$ with the exception of at most one path variable: if the HyperLTL formula holds for the abstract traces, it holds for the actual system as well. In a similar manner, if we under-approximate all path variables save one, and a HyperLTL formula that only use existential quantifiers $\exists$ with the exception of at most one path variable doesn't hold for this approximation, then it doesn't hold for the actual program.

We also show that the model-checking problem for PDSs of HyperLTL formulas that only use universal quantifiers $\forall$ can be approximated by performing a bounded-phase model-checking of a LTL formula for a *multi-stack pushdown system* (MPDS), where a *phase* is a part of a run during which there is at most one stack that is popped from, as defined by La Torre et al. in [TMP07].

These results were published in [PT18].

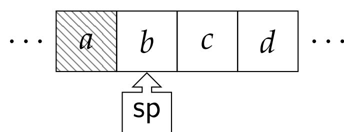

FIGURE 1.1: The original stack.

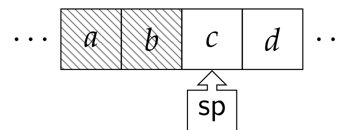

FIGURE 1.2: The stack after one pop.

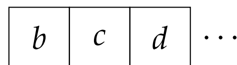

FIGURE 1.3: The original PDS stack.

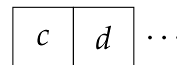

FIGURE 1.4: The PDS stack after one pop.

1.4 Pushdown systems with an upper stack

In the second part of this thesis, we extend pushdown systems in order to keep track of the content of the memory section just above the stack.

Indeed, PDSs were introduced to accurately model the *call stack* of a program. A *call stack* is a stack data structure that stores information about the active procedures of a program such as return addresses, passed parameters and local variables. It is usually implemented using a *stack pointer (sp)* register that indicates the head of the stack. Thus, assuming the stack grows downwards, when data is *pushed* onto the stack, *sp* is decremented before the item is placed on the stack. For instance, in the *x*86 architecture *sp* is decremented by 4 (pushing 4 bytes). When data is *popped* from the stack, *sp* is incremented. For instance, in the *x*86 architecture *sp* is incremented by 4 (popping 4 bytes).

However, in a PDS, neither push nor pop rules are truthful to the assembly stack. During an actual pop operation on the stack, the item remains in memory and the stack pointer is increased, as shown in Figures 1.1 and 1.2, whereas a PDS deletes the item on the top of the stack, as shown in Figures 1.3 and 1.4.

This subtle difference becomes important when we want to analyze programs that directly manipulate the stack pointer and use assembly code. Indeed, in most assembly languages, *sp* can be used like any other register. As an example, the instruction $\mathsf{mov\ eax\ [sp-4]}$ will put the value pointed to at address $\mathsf{sp}-4$ in the register eax (one of the general registers). Since $\mathsf{sp}-4$ is an address above the stack pointer, we do not know what is being copied into the register eax, unless we have a way to record the elements that had previously been popped from the stack and not overwritten yet. Such instructions may happen

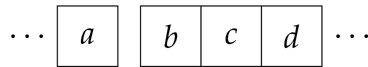

FIGURE 1.5: The original UPDS stacks.

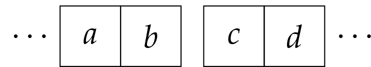

FIGURE 1.6: The UPDS stacks after one pop.

in malicious assembly programs: malware writers tend to do unusual things in order to obfuscate their payload and thwart static analysis.

Thus, it is important to record the part of the memory that is just above the stack pointer. To this end, we extend PDSs in order to keep track of this *upper stack*: we introduce a new model called *pushdown system with an upper stack* (UPDS) that extends the semantics of PDSs. In a UPDS, when a letter is popped from the top of the stack (*lower stack* from now on), it is added to the bottom of a write-only *upper stack*, effectively simulating the decrement of the stack pointer. This is shown in Figures 1.5 and 1.6, where after being popped, b is removed from the lower stack (on the right) and added to the upper stack (on the left) instead of being destroyed. The top of the lower stack and the bottom of the upper stack meet at the stack pointer.

We prove that the following properties hold for the class of UPDSs:

— the sets of predecessors and successors of a regular set of configurations are not regular; however, the set of successors of a regular set of configurations is context-sensitive;

— the set of predecessors is regular given a limit of k phases, a phase being a part of a run during which either pop or push rules are forbidden; this is an under-approximation of the actual set of predecessors;

— an over-approximation of the set of successors can be computed by abstracting the set of runs first;

We then show that the UPDS model and the approximations of its reachability sets described above can be used to find errors and security flaws in programs.

These results were published in [PDT17].

1.5 Synchronized dynamic pushdown networks

In the third part of this thesis, we tackle the model-checking problem for concurrent programs.

Indeed, the use of parallel programs has grown in popularity in the past fifteen years, but these remain nonetheless fickle and vulnerable to specific issues such as race conditions or deadlocks. Static analysis methods for this class of programs remain therefore more relevant than ever.

As mentioned previously, *pushdown systems* are a natural model for programs with sequential, recursive procedure calls [EHRS00]. Thus, networks of pushdown systems can be used to model multithreaded programs, where each PDS in the network models a sequential component of the whole program. In this context, *dynamic pushdown networks* (DPNs) were introduced by Bouajjani et al. in [BMOT05].

Intuitively, this class of automata consists in a network of pushdown systems running independently in parallel. Each member of a DPN can, after a transition, spawn a new PDS which is then introduced as a new member of the network. Thus, DPNs can be used to represent a network of threads where each thread can recursively call procedures, perform internal actions, or spawn a new thread.

However, this model cannot represent synchronization between different threads or parallel components. In order to handle communication in multithreaded programs, Bouajjani et al. introduced in [BET03] *communicating pushdown systems* (CPDSs), a model which consists of a tuple of pushdown systems synchronized by rendez-vous on execution paths. However, CPDSs have a constant number of processes and cannot therefore handle dynamic creation of new threads.

Hence, we introduce a more accurate model, namely, *synchronized dynamic pushdown networks* (SDPNs), that combines DPNs with synchronization by rendez-vous in order to handle dynamic thread creation and communication at the same time.

A SDPN can be seen as a DPN where PDS processes can synchronize via rendez-vous by sending and receiving messages. In a SDPN, pushdown processes can apply internal actions labelled by a letter τ without synchronization, just like a DPN, but can also synchronize through channels.

To do so, we represent each channel by a pair of letters, as an example a and $\overline{a}$, that can be used to label transitions. If one thread can execute an action labelled with a signal a, and another thread another action labelled with $\overline{a}$, then both threads can synchronize and execute their respective transitions simultaneously, in a single step labelled by τ.

We consider the reachability problem for SDPNs, that is, finding if a critical configuration can be reached from the set of starting configurations of the program. An equivalent problem is to compute the set $Paths(\mathcal{C}, \mathcal{C}')$ of execution paths leading from a configuration in $\mathcal{C}$ to a

configuration in C' and check if it is empty. Unfortunately, this problem remains undecidable for synchronized pushdown systems, as proven by Ramalingam in [Ram00].

Therefore, the set of execution paths $Paths(C, C')$ cannot be computed in an exact manner. To overcome this problem, we proceed in a manner similar to the method outlined in [BET03]: our approach is based on the computation of an abstraction $\alpha(Paths(\mathcal{C}, \mathcal{C}'))$ of the execution paths language. To this aim, we propose techniques based on:

— the representation of regular sets of configurations of SDPNs with finite word automata;

— the use of these automata to determine a set of constraints whose least fixpoint characterizes the set of execution paths of the program; to compute this set of constraints, (1) we consider a relaxed semantics on SDPNs that allows partially synchronized runs, (2) we abstract sets of execution paths as functions in a Kleene algebra, instead of simple elements of the abstract domain, and (3) we use a shuffle product on abstract path expressions to represent the interleaving and potential synchronization of parallel executions;

— the resolution of this set of constraints in an abstract domain; we consider in particular the case where the abstract domain is finite; the set of constraints can then be solved using an iterative fixpoint computation.

Note that the main contributions of our approach with regards to the methods outlined [BET03, Tou05] are the introduction of functions to represent sets of abstract path expressions and the use of a shuffle product to model the interleaving of threads. The abstraction framework as defined in these papers cannot be applied to SDPNs due to the presence of dynamic thread creation, hence, the need for functions and shuffling.

We can then apply this over-approximation framework for the reachability problem to a *iterative abstraction refinement* scheme inspired by the work of Chaki et al. in [CCK+06]. The idea is the following: (1) we do a reachability analysis of the program, using a finite domain abstraction of order n in our over-approximation framework; if the target set of configurations is not reachable by the abstract paths, it is not reachable by actual execution paths either; otherwise, we obtain a counter-example; (2) we check if the counter-example can be matched to an actual execution of the program; (3) if it does, then we have shown that the target set of configurations is actually reachable; (4) otherwise, we refine our abstraction and use instead a finite domain abstraction of order $n+1$ in step (1). This scheme is then used to prove that a Windows driver first presented in [QW04] can reach an erroneous configuration, using an abstraction of the original program.

These results were first published in [PT17].

1.6 Thesis outline

The purpose of Chapter 2 is to remind the reader of important definitions and results on *pushdown systems* (PDSs), reachability properties, and model-checking algorithms relevant for the rest of this thesis. In Chapter 3, we focus on model-checking the logic HyperLTL for pushdown systems. We introduce in Chapter 4 a new class of automata called *pushdown systems with an upper stack* (UPDSs) and focus on its reachability problem. We design in Chapter 5 a concurrent model featuring both thread spawns and synchronization between threads called *synchronized dynamic pushdown networks* (SDPNs) in such a manner that a convenient over-approximation of the execution paths can be computed thanks to a theoretical framework known as Kleene abstractions. We present our conclusion in Chapter 6.

Chapter 2

Model-checking Pushdown Systems

Pushdown systems (PDSs) were introduced to model the *call stack* of a program that stores information about the active procedures such as return addresses, passed parameters and local variables. Without such a stack, a finite-state automaton can't represent accurately programs with nested, recursive function calls, hence, the need for a more expressive model.

The purpose of this chapter, which features no new contributions of our own, is to present to the reader important definitions and model-checking techniques on PDSs that are relevant to the rest of this thesis.

Chapter outline. In Section 1 of this chapter, we define pushdown systems and explain why they are a relevant model. Then, in Section 2, we recall how the forward and backward reachability problems for PDSs can be solved using an automata-theoretic approach. Finally, in Section 3, we define the linear-time temporal logic and show how it can be applied to PDSs.

2.1 Pushdown systems

2.1.1 The model

Pushdown systems are a natural model for sequential programs with recursive procedure calls.

Definition 1 (Pushdown system)**.** *A* pushdown system *(PDS) is a tuple $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ where P is a finite set of control states, Σ a finite input alphabet, Γ a finite stack alphabet, and $\Delta \subseteq P \times \Gamma \times \Sigma \times P \times \Gamma^*$ a finite set of transition rules.*

If $d = (p, \gamma, a, p', w) \in \Delta$, we write $d = (p, \gamma) \xrightarrow{a} (p', w)$. We call a the *label* of d. We can assume without loss of generality that $\Delta \subseteq P \times \Gamma \times \Sigma \times P \times \Gamma^{\leq 2}$. We say that a PDS is *unlabeled* if $\Sigma = \emptyset$. We can then write $\mathcal{P} = (P, \Gamma, \Delta)$.

A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ is a control state and $w \in \Gamma^*$ a stack content. Let $Conf_{\mathcal{P}} = P \times \Gamma^*$ be the set of all configurations of $\mathcal{P}$. A set of configurations $\mathcal{C}$ of a PDS $\mathcal{P}$ is said to be *regular* if $\forall p \in P$, there exists a finite-state automaton $\mathcal{A}_p$ on the alphabet Γ such that $\mathcal{L}(\mathcal{A}_p) = \{w \mid \langle p, w \rangle \in \mathcal{C}\}$, where $\mathcal{L}(\mathcal{A})$ stands for the language recognized by an automaton $\mathcal{A}$.

To some PDSs we match an *initial configuration* $c_0 \in Conf_{\mathcal{P}}$ of the form $c_0 = \langle p_0, \bot \rangle$, where $\bot \in \Gamma$ is a special bottom stack symbol and $p_0 \in P$ a control state. We then introduce these PDSs as quintuplets of the form $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0)$.

For each $a \in \Sigma$, we define the transition relation $\xrightarrow{a}_{\mathcal{P}}$ on configurations as follows: if $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta$, for each $w' \in \Gamma^*$, $\langle p, \gamma w' \rangle \xrightarrow{a}_{\mathcal{P}} \langle p', w w' \rangle$. From these relations, we can then infer the *immediate successor* relation $\rightarrow_{\mathcal{P}} = \bigcup\limits_{a \in \Sigma} \xrightarrow{a}_{\mathcal{P}}$.

The *reachability* relation $\Rightarrow_{\mathcal{P}}$ is the reflexive and transitive closure of the immediate successor relation $\rightarrow_{\mathcal{P}}$. If $\mathcal{C}$ is a set of configurations, we introduce its set of *successors* $post^*(\mathcal{P}, \mathcal{C}) = \{c \in P \times \Gamma^* \mid \exists c' \in \mathcal{C}, c' \Rightarrow_{\mathcal{P}} c\}$ and its set of *predecessors* $pre^*(\mathcal{P}, \mathcal{C}) = \{c \in P \times \Gamma^* \mid \exists c' \in \mathcal{C}, c \Rightarrow_{\mathcal{P}} c'\}$. We may omit the variable $\mathcal{P}$ when only a single PDS is being considered.

A *run* r starting from a configuration c is a sequence of configurations $r = (c_i)_{i \geq 0}$ such that $c_0 = c$ and $\forall i \geq 0$, $c_i \xrightarrow{a_i}_{\mathcal{P}} c_{i+1}$. The word $(a_i)_{i \geq 0}$ is then said to be the *trace* of r. Traces and runs may be finite or infinite.

Let $Runs_\omega(\mathcal{P}, \mathcal{C})$ (resp. $Runs(\mathcal{P}, \mathcal{C})$) be the set of all infinite (resp. finite) runs of $\mathcal{P}$ starting from a configuration $c \in \mathcal{C}$. We define $Traces_\omega(\mathcal{P}, \mathcal{C})$ and $Traces(\mathcal{P}, \mathcal{C})$ in a similar manner.

If $\mathcal{P}$ has an initial configuration c_0, then we write $Runs_\omega(\mathcal{P}) = Runs_\omega(\mathcal{P}, \{c_0\})$. We define $Runs(\mathcal{P})$, $Traces(\mathcal{P})$, and $Traces_\omega(\mathcal{P})$ as well in a similar manner.

2.1.2 $\mathcal{P}$-automata

In order to represent regular sets of configurations, we consider the following structure:

Definition 2 (Bouajjani et al. [BEM97]). *Let $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ be a pushdown system. A $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, I, F)$ is a finite automaton on the stack alphabet Γ of $\mathcal{P}$ where Q is a set of states such that $P \subseteq Q$, $I = P$ the set of initial states, $F \subseteq Q$ the set of final states, and $\delta \subseteq Q \times \Gamma \cup \{\varepsilon\} \times Q$ a set of transitions.*

Intuitively, a $\mathcal{P}$-automaton is a finite automaton whose edges are labeled by stack symbols of $\mathcal{P}$ and whose initial states represent the states of $\mathcal{P}$.

Let $\rightarrow_{\mathcal{A}}$ be the transition relation inferred from δ. We say that $\mathcal{A}$ *accepts* a configuration $\langle p, w \rangle$ if there is a path $p \xrightarrow{w}{}^{*}_{\mathcal{A}} f$ such that $f \in F$. Let $L(\mathcal{A}) \subseteq Conf_{\mathcal{P}}$ be the set of configurations accepted by $\mathcal{A}$. Obviously, the following lemma holds:

Lemma 1 (Bouajjani et al. [BEM97]). *A set of configurations $\mathcal{C}$ of a PDS $\mathcal{P}$ is regular if and only if there exists a $\mathcal{P}$-automaton $\mathcal{A}$ such that $L(\mathcal{A}) = \mathcal{C}$.*

2.1.3 From a program to a pushdown system

PDSs can be used to model sequential problems with recursion, as shown by Esparza et al. in [EHRS00]. We abstract away data and variables and represent each procedure of the program by a *control flow graph* (CFG). The nodes of a CFG stand for control points, while its edges are labeled by statements such as calls to other procedures. If we abstract the value of variables, the CFG can be non-deterministic.

We consider a set of CFGs with a set of control points N. We build an unlabeled PDS with pushdown alphabet N and a single control point $\{p\}$: a configuration $\langle p, \gamma w \rangle$ stands for a situation where the program is at a control point γ and w represents the return addresses of the calling procedure on the stack. Its pushdown rules are the following:

— if the CFG moves from a control point γ to a control point γ' without calling procedure, we add a rule $(p, \gamma) \rightarrow (p, \gamma')$;

— if the CFG moves from a control point γ to a control point γ' while calling a procedure starting in control point γ'', we add a rule $(p, \gamma) \rightarrow (p, \gamma''\gamma')$;

— if an edge in the CFG leaves the control point γ with a *return* statement, we add a rule $(p, \gamma) \rightarrow (p, \varepsilon)$.

2.2 Reachability sets

Many static analysis methods rely on being able to determine whether a given critical state is reachable or not from the starting configuration of a program, hence, the need for reachability analysis techniques.

Let $\mathcal{C}$ be a regular set of configurations of an unlabeled PDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$, and let $\mathcal{A}$ be a $\mathcal{P}$-automaton accepting $\mathcal{C}$. Labels are not relevant in this situation as reachability sets do not depend on them. It has been proven by Didier Caucal in [Cau92] that the sets $pre^*(\mathcal{P}, \mathcal{C})$ and $post^*(\mathcal{P}, \mathcal{C})$ are regular. In this section, we present the automata-theoretic approach used in [BEM97, EHRS00] to compute them.

2.2.1 Computing pre^*

We compute a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ accepting $pre^*(\mathcal{C})$ by applying the saturation procedure introduced by Bouajjani et al. in [BEM97] to $\mathcal{A}$:

> if $(p, \gamma) \rightarrow (p', w) \in \Delta$ and there is a path $p' \xrightarrow{w}{}^* q$ in the current automaton, add a transition $p \xrightarrow{\gamma} q$ to the automaton.

The following theorem holds:

Theorem 1 (Bouajjani et al. [BEM97])**.** *The $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ accepts $pre^*(\mathcal{C})$.*

Intuitively, if a configuration $\langle p', ww' \rangle$ belongs to $pre^*(\mathcal{C})$, and there is a rule $(p, \gamma) \rightarrow (p', w) \in \Delta$, then $\langle p, \gamma w' \rangle \rightarrow_{\mathcal{P}} \langle p', ww' \rangle$ and therefore $\langle p, \gamma w' \rangle \in pre^*(\mathcal{C})$. Hence, if $\mathcal{A}_{pre^*}$ accepts $\langle p', ww' \rangle$, it should accept $\langle p, \gamma w' \rangle$ as well.

2.2.2 Computing $post^*$

We compute a $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ accepting $post^*(\mathcal{C})$ by applying the two-step procedure introduced by Esparza et al. in [EHRS00] to $\mathcal{A}$:

1. for each transition rule of the form $(p, \gamma) \rightarrow (p', \gamma' \gamma'') \in \Delta$, add a state $q_{p',\gamma'}$ to $\mathcal{A}$ and a transition $p' \xrightarrow{\gamma'} q_{p',\gamma'}$;
2. apply the following saturation rules:

   — if $(p, \gamma) \rightarrow (p', \gamma') \in \Delta$ and there is a transition $p \xrightarrow{\gamma} q$ in the current automaton, add a transition $p' \xrightarrow{\gamma'} q$ to the automaton;

— if $(p,\gamma) \to (p',\varepsilon) \in \Delta$ and there is a transition $p \xrightarrow{\gamma} q$ in the current automaton, add a transition $p' \xrightarrow{\varepsilon} q$ to the automaton;

— if $(p,\gamma) \to (p',\gamma'\gamma'') \in \Delta$ and there is a transition $p \xrightarrow{\gamma} q$ in the current automaton, add a transition $q_{p',\gamma'} \xrightarrow{\gamma''} q$ to the automaton.

The following theorem holds:

Theorem 2 (Esparza et al. [EHRS00]). *The $\mathcal{P}$-automaton $\mathcal{A}_{post^*}$ accepts $post^*(\mathcal{C})$.*

Intuitively, if a configuration $\langle p, \gamma w' \rangle$ belongs to $post^*(\mathcal{C})$, and there is a rule $(p,\gamma) \to (p',w) \in \Delta$, then $\langle p, \gamma w' \rangle \to_{\mathcal{P}} \langle p', ww' \rangle$ and therefore $\langle p, ww' \rangle \in post^*(\mathcal{C})$. Hence, if $\mathcal{A}_{post^*}$ accepts $\langle p, \gamma w' \rangle$, it should accept $\langle p', ww' \rangle$ as well.

2.3 Model-checking LTL on pushdown systems

The most widely used variant of temporal logics is the *linear-time temporal logic* LTL introduced by Pnueli in [Pnu77].

2.3.1 The linear-time temporal logic LTL

Let AP be a finite set of *atomic propositions* used to express facts about a program. A *path* is an infinite word $\pi = (\pi_i)_{\geq 0}$ in the set $(2^{AP})^{\omega}$.

Definition 3 (LTL). *The set of* LTL *formulas is given by the following grammar:*

$$\varphi, \psi ::= \bot \mid p \in \mathrm{AP} \mid \neg\varphi \mid \varphi \vee \psi \mid \mathrm{X}\,\varphi\,(Next) \mid \varphi\,\mathrm{U}\,\psi\,(Until)$$

$\bot$ stands for the predicate 'always true'. X and U are called the *next* and *until* operators: the former means that a formula should happen at the next step, the latter, that a formula should hold at least until another formula becomes true. We consider the following semantics on paths:

Definition 4 (Semantics of LTL). *Let φ be a* LTL *formula, $\pi \in Paths$, and $i \in \mathbb{N}$. We define inductively the semantics of the relation $\pi, i \models \varphi$:*

$$\begin{aligned} \pi, i \models \rho \text{ where } \rho \in AP &\Leftrightarrow \rho \in \pi_i \\ \pi, i \models \mathrm{X}\, \varphi &\Leftrightarrow \pi, i+1 \models \varphi \\ \pi, i \models \varphi \,\mathrm{U}\, \psi &\Leftrightarrow \exists j \geq i \text{ such that } \pi, j \models \psi \text{ and} \\ & \quad\ \forall k \in \{i, \ldots, j-1\}, \pi, k \models \varphi \end{aligned}$$

as well as the obvious interpretation of the boolean operators.

Intuitively, $\pi, i \models \varphi$ means that the path π verifies ϕ from its i-th symbol onward. We consider the language $L(\varphi) = \{w \mid w \in (2^{AP})^\omega$ and $w, 0 \models \varphi\}$ of a LTL formula φ, that is, the set of all paths verifying φ according to the semantics outlined previously.

2.3.2 Büchi automata

We consider the following class of finite state automata:

Definition 5 (Büchi automaton). *A* Büchi automaton *(BA) $\mathcal{B}$ is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite input alphabet, $\delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$ a set of transitions, $F \subseteq Q$ a set of accepting states, and $q_0 \in Q$ the initial state.*

The language $L(\mathcal{B})$ accepted by $\mathcal{B}$ is the set of all infinite sequences w in Σ^ω such that there is an infinite run r of $\mathcal{B}$ with trace w starting in state q_0 that visits accepting states from F infinitely often. A language is called ω-regular if and only if there is a Büchi automaton accepting it.

BAs can be used in the following fashion:

Theorem 3 (Kesten et al. [KMMP93]). *Given a LTL formula φ, there is a Büchi automaton $\mathcal{B}$ on the alphabet $\Sigma = 2^{AP}$ such that $L(\mathcal{B}) = L(\varphi)$.*

2.3.3 The model-checking problem

Let $\nu : Conf_{\mathcal{P}} \to 2^{AP}$ be a valuation function on configurations of a PDS $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$. It is said to be *simple* if for all $w, w' \in \Gamma^*$, $p \in P$, and $\gamma \in \Gamma$, we have $\nu(\langle p, \gamma w\rangle) = \nu(\langle p, \gamma w'\rangle)$. Intuitively, a simple valuation is equivalent to a function $\nu : P \times \Gamma \to 2^{AP}$ that only depends on the control state and the top stack symbol.

Let $r = (r_i)_{i \geq 0}$ be an infinite run of $\mathcal{P}$. We define the image $\nu(r) = (\nu(r_i))_{i \geq 0}$ in $(2^{AP})^\omega$ of r by the valuation function ν. We write that $r \models_\nu \varphi$ if $\nu(r), 0 \models \varphi$.

The *model-checking* problem is defined as follows:

Definition 6 (The model-checking problem)**.** *Given a LTL formula φ, a PDS $\mathcal{P}$ with a starting configuration c_0, and a simple valuation ν on configurations of $\mathcal{P}$, the model-checking problem consists in determining whether $\forall r \in Runs(\mathcal{P})$, $r \models_\nu \varphi$.*

An automata-theoretic approach has been used in [BEM97, EHRS00] to answer the model-checking problem of LTL against PDSs.

Chapter 3

Model-checking HyperLTL for Pushdown Systems

In this chapter, we focus on model-checking HyperLTL against pushdown systems (PDSs). Temporal logics such as LTL are often used to express safety or correctness properties of programs. However, they cannot model complex formulas known as hyperproperties introducing relations between different execution paths of a same system. In order to do so, the logic *HyperLTL* adds existential and universal quantifications of path variables to LTL. The model-checking problem, that is, determining if a given representation of a program verifies a HyperLTL property, has been shown to be decidable for finite state systems by Clarkson et al. in [CFK+14].

Unfortunately, we prove that this result does not hold for pushdown systems nor for the subclass of *visibly pushdown systems*. Therefore, we introduce algorithms in order to approximate the model-checking problem, using either an automata-theoretic approach or a bounded-phase analysis of a *multi-stack pushdown system*. We then show how these approximations can be used to verify security policies.

Chapter outline. In Section 1 of this chapter, we provide background on *visibly pushdown systems* (VPDSs). We define in Section 2 the hyper linear-time logic *HyperLTL*, and prove that its model-checking problem against PDSs and VPDSs is undecidable. Then, in Section 3, we propose an approximation of the model-checking problem for PDSs. In Section 4, we use *multi-stack pushdown systems* (MPDSs) and *bounded phase analysis* in order to compute another approximation of the model-checking problem. In Section 5, we apply the logic HyperLTL to express security properties. Finally, we describe the related work in Section 6 and present our conclusion in Section 7.

These results were published in [PT18].

3.1 Visibly pushdown systems

We consider a particular subclass of PDSs introduced by Alur et al. in [AM04]. Let $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ be a partition of the input alphabet, where Σ_c, Σ_r, and Σ_l stand respectively for the *call*, *return*, and *local* alphabets.

Definition 7 (Alur et al. [AM04]). *A* visibly pushdown system *(VPDS) over a partition $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ of Σ is a PDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0)$ verifying the following properties:*

— *if $(p, \gamma_1) \xrightarrow{a} (p', \gamma_2) \in \Delta$, then $a \in \Sigma_l$, $\gamma_1 = \gamma_2$, and $\forall \gamma \in \Gamma$, $(p, \gamma) \xrightarrow{a} (p', \gamma) \in \Delta$;*

— *if $(p, \gamma) \xrightarrow{a} (p', \varepsilon) \in \Delta$, then $a \in \Sigma_r$;*

— *if $(p, \gamma_1) \xrightarrow{a} (p', \gamma_2\gamma_1) \in \Delta$, then $a \in \Sigma_c$, and $\forall \gamma \in \Gamma$, $(p, \gamma) \xrightarrow{a} (p', \gamma_2\gamma) \in \Delta$;*

VPDSs are an *input-driven* subclass of PDSs: at each step of a computation, the next stack operation will be determined by the input letter in Σ read, depending on which subset of the partition $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$ the aforementioned letter belongs to.

Visibly pushdown automata accept the class of *visibly pushdown languages*. If a BPDA $\mathcal{BP}$ is visibly pushdown according to a partition of Σ, we say it's a *Büchi visibly pushdown automata* (BVPDA). The class of languages accepted by BVPDA is called *ω visibly pushdown languages*.

Unlike context-free languages, the emptiness of the intersection of visibly pushdown languages is a decidable problem [AM04] and the complement of a visibly pushdown language is a visibly pushdown language that can be computed. The same properties also hold for ω visibly pushdown languages.

3.2 HyperLTL

3.2.1 The logic

Let AP be a finite set of atomic propositions used to express facts about a program; a *path* is an infinite word in $(2^{AP})^\omega = \mathcal{T}$. Let $\mathcal{V}$ be a finite set of *path variables*. The *HyperLTL* logic relates multiple paths by introducing quantifiers on path variables.

Definition 8 (Syntax of HyperLTL)**.** *Unquantified HyperLTL formulas are defined according to the following syntax equation:*

$$\varphi ::= \bot \mid (a, \pi) \in AP \times \mathcal{V} \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathrm{X}\, \varphi \mid \varphi \,\mathrm{U}\, \varphi \mid \mathrm{G}\, \varphi \mid \mathrm{F}\, \varphi$$

From then on, we write $a_\pi = (a, \pi)$. HyperLTL formulas are defined according to the following syntax equation:

$$\psi ::= \exists \pi, \varphi \mid \forall \pi, \varphi \mid \varphi$$

where $\pi \in \mathcal{V}$ is a path variable.

The existential $\exists$ and universal quantifiers $\forall$ are used to define path variables, to which atomic propositions in AP are bound. A HyperLTL formula is said to be *closed* if there is no free variable: each path variable is bound by a path quantifier exactly once.

As an example, the closed formula $\forall \pi_1, \exists \pi_2, \varphi$ means that for all paths π_1, there exists a path π_2 such that the formula φ holds for π_1 and π_2. Simple LTL formulas can be considered as a subclass of closed HyperLTL formulas of the form $\forall \pi, \varphi$ with a single path variable.

Let $\Pi : \mathcal{V} \to \mathcal{T}$ be a *path assignment function* of $\mathcal{V}$ that matches to each path variable π a path $\Pi(\pi) \in \mathcal{T}$. If $\Pi(\pi) = (t_j)_{j \geq 0}$, for all $i \geq 0$, we define the i-th value of the path $\Pi(\pi)[i] = t_i$ and a suffix assignment function $\Pi[i, \infty]$ such that $\Pi[i, \infty](\pi) = (t_j)_{j \geq i}$.

We first define the semantics of this logic for path assignment functions.

Definition 9 (Semantics of unquantified HyperLTL formulas)**.** *Let φ be an unquantified HyperLTL formula. We define by induction on φ the following semantics on path assignment functions:*

$$\begin{aligned}
\Pi \models a_\pi &\Leftrightarrow a \in \Pi(\pi)[0] \\
\Pi \models \neg\varphi &\Leftrightarrow \Pi \not\models \neg\varphi \\
\Pi \models \varphi_1 \vee \varphi_2 &\Leftrightarrow (\Pi \models \varphi_1) \vee (\Pi \models \varphi_2) \\
\Pi \models \varphi_1 \wedge \varphi_2 &\Leftrightarrow (\Pi \models \varphi_1) \wedge (\Pi \models \varphi_2) \\
\Pi \models \mathrm{X}\, \varphi &\Leftrightarrow \Pi[1, \infty] \models \varphi \\
\Pi \models \varphi \,\mathrm{U}\, \psi &\Leftrightarrow \exists j \geq 0, \Pi[j, \infty] \models \psi \textit{ and } \forall i \in \{0, \ldots, j-1\}, \\
&\qquad \Pi[i, \infty] \models \varphi \\
\Pi \models \mathrm{G}\, \varphi &\Leftrightarrow \forall i \geq 0, \Pi[i, \infty] \models \varphi \\
\Pi \models \mathrm{F}\, \varphi &\Leftrightarrow \exists i \geq 0, \Pi[i, \infty] \models \varphi
\end{aligned}$$

$\Pi \models \varphi$ if φ holds for a given assignment of path variables defined according to Π.

Let $T : \mathcal{V} \to 2^{\mathcal{T}}$ be a *set assignment function* of $\mathcal{V}$ that matches to each path variable $\pi \in \mathcal{V}$ a set of paths $T(\pi) \subseteq \mathcal{T}$. We can now define the semantics of closed HyperLTL formulas for set assignment functions.

Definition 10 (Semantics of closed HyperLTL formulas)**.** *We consider a closed HyperLTL formula $\psi = \chi_0 \pi_0, \ldots, \chi_n \pi_n, \varphi$, where each $\chi_i \in \{\forall, \exists\}$ is an universal or existential quantifier, and φ an unquantified HyperLTL formula using trace variables $\pi_0, \ldots, \pi_n$.*

For a given set assignment function T, we write that $T \models \psi$ if for $\chi_0 t_0 \in T(\pi_0)$, ..., $\chi_n t_n \in T(\pi_n)$, we have $\Pi \models \varphi$, where Π is the path assignment function such that $\forall i \in \{0, \ldots, n\}$, $\Pi(\pi_i) = t_i$.

As an example, if $\psi = \forall \pi_1, \exists \pi_2, \varphi$ is a closed HyperLTL formula and T is a set assignment function of $\mathcal{V}$, then $T \models \psi$ if $\forall t_1 \in T(\pi_1)$, $\exists t_2 \in T(\pi_2)$ such that $\Pi \models \varphi$, where $\Pi(\pi_1) = t_2$ and $\Pi(\pi_2) = t_2$. Intuitively, $T \models \psi$ if, assuming path variables belong to the path sets defined by T, the closed formula ψ holds. From then on, we assume that *every HyperLTL formula considered in this chapter is closed*.

3.2.2 HyperLTL and PDSs

Let $\mathcal{P}$ be a PDS on the input alphabet $\Sigma = 2^{AP}$ and ψ a closed HyperLTL formula. We write that $\mathcal{P} \models \psi$ if and only if $T \models \psi$ where the set assignment function T is such that $\forall \pi \in \mathcal{V}$, $T(\pi) = Traces_\omega(\mathcal{P})$. Determining whether $\mathcal{P} \models \psi$ for a given PDS $\mathcal{P}$ and a given HyperLTL formula ψ is called the *model-checking* problem of ψ against $\mathcal{P}$. The following theorem holds:

Theorem 4. *The model-checking problem of HyperLTL against PDSs is undecidable.*

The intuition. We can prove this result by reducing the emptiness of the intersection of two context-free languages, a well-known undecidable problem, to the model-checking problem. Our intuition is to consider two context-free languages $\mathcal{L}_1$ and $\mathcal{L}_2$ on the alphabet Σ. As HyperLTL formulas apply to infinite words, we define two BPDA $\mathcal{BP}_1$ and $\mathcal{BP}_2$ that accept $\mathcal{L}_1 f^\omega$ and $\mathcal{L}_2 f^\omega$ respectively, where $f \notin \Sigma$ is a special ending symbol. We then define a PDS $\mathcal{P}$ that can simulate either $\mathcal{BP}_1$ or $\mathcal{BP}_2$.

We now introduce the HyperLTL formula $\psi = \exists \pi_1, \exists \pi_2, \varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$: the unquantified formula φ_{start} expresses that trace variables π_1 and π_2 represent runs of $\mathcal{BP}_1$ and $\mathcal{BP}_2$ respectively, φ_{sync} means that

the two traces are equal from their second letter onwards, and φ_{end} implies that the two traces are accepting. Hence, if $\mathcal{P} \models \psi$, then $\mathcal{BP}_1$ and $\mathcal{BP}_2$ share a common accepting run, and $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$.

On the other hand, if $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$, there is an accepting trace π common to $\mathcal{BP}_1$ and $\mathcal{BP}_2$ and we can define two traces π_1 and π_2 of $\mathcal{P}$ such that the formula $\varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$ holds.

Since the emptiness problem is undecidable, so must be the model-checking problem.

Proof of Theorem 4. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two context-free languages, and $\mathcal{P}_1 = (P_1, \Sigma, \Gamma, \Delta_1, \langle p_0^1, \perp\rangle, F_1)$ and $\mathcal{P}_2 = (P_2, \Sigma, \Gamma, \Delta_2, \langle p_0^2, \perp\rangle, F_2)$ two PDA accepting $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively. Without loss of generality, we can consider that $P_1 \cap P_2 = \emptyset$. Let $e_1 \notin P_1$, $e_2 \notin P_2$, and $f \notin \Sigma$.

We define two BPDA $\mathcal{BP}_i = (P_i \cup \{e_i\}, 2^{\Sigma \cup \{f\}}, \Gamma, \Delta'_i, \langle p_0^i, \perp\rangle, \{e_i\})$ for $i = 1, 2$, where Δ'_i is such that $(e_i, \gamma) \xrightarrow{\{f\}} (e_i, \gamma) \in \Delta'_i$ and $(p_f, \gamma) \xrightarrow{\{f\}} (e_i, \gamma) \in \Delta'_i$ for all $\gamma \in \Gamma$ and $p_f \in F_i$, and if $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta_i$, then $(p, \gamma) \xrightarrow{\{a\}} (p', w) \in \Delta'_i$. If we consider that $\{a\}$ is equivalent to the label $a \in \Sigma \cup \{f\}$, $\mathcal{BP}_1$ and $\mathcal{BP}_2$ accept the languages $\mathcal{L}_1 f^\omega$ and $\mathcal{L}_2 f^\omega$ respectively. Since HyperLTL formulas apply to infinite words in $2^A P$, for $i = 1, 2$, we have designed a BPDA $\mathcal{BP}_i$ that extends words in $\mathcal{L}_i$ by adding a final looping state e_i from which the automaton can only output an infinite sequence of the special ending symbol f.

We consider now the PDS $\mathcal{P} = (\{p_0\} \cup P_1 \cup P_2, \{\{\iota^1\}, \{\iota^2\}\} \cup 2^{\Sigma \cup \{f\}}, \Gamma, \Delta, c_0)$, where $p_0 \notin P_1 \cup P_2$, $\iota^1, \iota^2 \notin \Sigma \cup \{f\}$, $c_0 = \langle p_0, \perp\rangle$, and $\Delta = \{(p_0, \perp) \xrightarrow{\{\iota^1\}} (p_0^1, \perp), (p_0, \perp) \xrightarrow{\{\iota^2\}} (p_0^2, \perp)\} \cup \Delta'_1 \cup \Delta'_2$. The PDS $\mathcal{P}$ can simulate either $\mathcal{BP}_1$ or $\mathcal{BP}_2$, depending on whether it applies first a transition labelled by $\{\iota^1\}$ or $\{\iota^2\}$ from the initial configuration c_0.

We introduce the formula $\psi = \exists \pi_1, \exists \pi_2, \varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$ on $AP = \{\iota^1, \iota^2\} \cup \Sigma \cup \{f\}$, where:

— $\varphi_{start} = \iota^1_{\pi_1} \wedge \iota^2_{\pi_2}$;

— $\varphi_{sync} = \text{XG} \bigwedge_{a \in AP} (a_{\pi_1} \Leftrightarrow a_{\pi_2})$;

— $\varphi_{end} = \text{FG}\,(f_{\pi_1} \wedge f_{\pi_2})$.

We suppose that $\mathcal{P} \models \psi$; φ_{start} expresses that trace variables π_1 and π_2 represent runs of $\mathcal{BP}_1$ and $\mathcal{BP}_2$ respectively. φ_{sync} means that the two traces are equal from their second letter onwards. φ_{end} implies that the two traces are accepting runs.

Therefore, if $\mathcal{P} \models \psi$, then $\mathcal{BP}_1$ and $\mathcal{BP}_2$ share a common accepting run and $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$. On the other hand, if $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$, there is

an accepting run π common to $\mathcal{B}_1$ and $\mathcal{B}_2$, and we can then find two traces π_1 and π_2 of $\mathcal{P}$ such that the formula $\exists \pi_1, \exists \pi_2, \varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$ holds. The emptiness problem is undecidable, and therefore so must be the model-checking problem. □

As a consequence of Theorem 4, determining whether $T \models \psi$ for a generic set assignment function T and a given HyperLTL formula ψ is an undecidable problem.

3.2.3 HyperLTL and VPDSs

Since the emptiness of the intersection of visibly pushdown languages is decidable, the previous proof does not apply to VPDSs and one might wonder if the model-checking problem of HyperLTL for this particular subclass is decidable. Unfortunately, we can show that this is not the case:

Theorem 5. *The model-checking problem of HyperLTL against VPDSs is undecidable.*

The intuition. In order to prove this theorem, we will rely on a class of two-stack automata called *2-visibly pushdown automata* (2-VPDA) introduced in [CMP07]. In a 2-VPDA, each stack is input driven, but follows its own partition of Σ. The same input letter may result in different pushdown rules being applied to the first and second stack: as an example, a transition can push a word on the first stack and pop the top letter of the second stack, depending on which partition is used by each stack. Moreover, in a manner similar to VPDA, transitions of 2-VPDA do not depend on the top stack symbols unless they pop them.

It has been shown in [CMP07] that the emptiness problem is undecidable for 2-VPDA. Our intuition is therefore to prove Theorem 5 by reducing the emptiness problem for 2-VPDA to the model-checking problem of HyperLTL against VPDSs. To do so, for a given 2-VPDA $\mathcal{D}$, we define a VPDS $\mathcal{P}$ and a HyperLTL formula ψ on two trace variables such that $\mathcal{P} \models \psi$ if and only if $\mathcal{D}$ has an accepting run.

$\mathcal{P}$ is such that it can simulate either stack of the 2-VPDA. However, both stacks must be synchronized in order to properly represent the whole automaton: the content of one stack can lead to a control state switch that may enable a transition modifying the other stack. The HyperLTL formula ψ determines which trace variable is related to which stack, synchronizes two runs of $\mathcal{P}$ in such a manner that they can be used to define an execution path of $\mathcal{D}$, and ensure that this path is an accepting one.

Introducing 2-visibly pushdown automaton. Let Σ be a finite input alphabet with two partitions $\Sigma = \Sigma_{c_j} \cup \Sigma_{r_j} \cup \Sigma_{l_j}$, $j \in \{1,2\}$. We then introduce a *2-pushdown alphabet* $\aleph = \left\langle \left(\Sigma_{c_1}, \Sigma_{r_1}, \Sigma_{l_1}\right), \left(\Sigma_{c_2}, \Sigma_{r_2}, \Sigma_{l_2}\right) \right\rangle$ on Σ.

Definition 11 (Carotenuto et al. [CMP07]). *A* 2-visibly pushdown automaton *(2-VPDA) over $\aleph$ is a tuple $\mathcal{D} = (P, \Sigma, \Gamma, \Delta, c_0, F)$ where P is a finite set of control states, Σ a finite input alphabet, Γ a finite stack alphabet, $\Delta \subseteq (P \times \Gamma \times \Gamma) \times \Sigma \times (P \times \Gamma^* \times \Gamma^*)$ a finite set of transition rules, $c_0 = \langle p_0, \perp, \perp \rangle \in P \times \Gamma \times \Gamma$ an initial configuration, and $F \subseteq P$ a set of final states. Moreover, Δ is such that $\forall d \in \Delta$, and for $i \in \{1,2\}$:*

- *if d is labelled by a letter in Σ_{c_i}, d pushes a word on the i-th stack regardless of its top stack symbol;*
- *if d is labelled by a letter in Σ_{r_i}, d pops the top letter of of the i-th stack;*
- *if d is labelled by a letter in Σ_{l_i}, d does not modify the i-th stack.*

The semantics of 2-VPDA is defined in a manner similar to PDA, and so are configurations, runs, execution paths, languages, and 2-Büchi visibly pushdown automata (2-BVPDA). The following theorem holds:

Theorem 6 (Carotenuto et al. [CMP07]). *The emptiness problem for 2-VPDA is undecidable.*

Proof of Theorem 5. Let $\mathcal{D} = (P, \Sigma, \Gamma, \Delta, \langle p_0, \perp, \perp \rangle, F)$ be a 2-VPDA on an input alphabet Σ according to a partition $\aleph = \langle (\Sigma_{c_1}, \Sigma_{r_1}, \Sigma_{l_1}), (\Sigma_{c_2}, \Sigma_{r_2}, \Sigma_{l_2}) \rangle$. Let $e \notin P$ and $f \notin \Sigma$. We introduce a 2-BVPDA $\mathcal{BD} = (P \cup \{e\}, 2^{\Sigma \cup \{f\}}, \Gamma, \Delta', \langle p_0, \perp, \perp \rangle, \{e\})$ such that $(e, \gamma, \gamma') \xrightarrow{\{f\}} (e, \gamma, \gamma') \in \Delta'$ and $(p_f, \gamma, \gamma') \xrightarrow{\{f\}} (e, \gamma, \gamma') \in \Delta'$ for all $\gamma, \gamma' \in \Gamma$ and $p_f \in F$, and if $(p, \gamma, \gamma') \xrightarrow{a} (p', w, w') \in \Delta$, then $(p, \gamma, \gamma') \xrightarrow{\{a\}} (p', w, w') \in \Delta'$ on the input alphabet $\Sigma \cup \{f\}$. Obviously, $\mathcal{BD}$ is visibly if we add the symbol f to Σ_{l_1} and Σ_{l_2} in the partition of $\Sigma \cup \{f\}$; it accepts $\mathcal{L}(\mathcal{D}) f^\omega$, assuming the label $\{a\}$ is equivalent to the label $a \in \Sigma \cup \{f\}$.

Let P^1 and P^2 (resp. Δ^1 and Δ^2) be two disjoint copies of P (resp. Δ'). To each $p \in P$ (resp. $d \in \Delta'$), we match its copies $p^1 \in P^1$ and $p^2 \in P^2$ (resp. $d^1 \in \Delta^1$ and $d^2 \in \Delta^2$). Let $\iota^1, \iota^2 \notin \Delta_1 \cup \Delta_2$. We define a PDS $P = (\{\sigma\} \cup P^1 \cup P^2, \{\{\iota^1\}, \{\iota^2\}, \{f\}\} \cup 2^{\Delta^1} \cup 2^{\Delta^2}, \Gamma, \delta, \langle \sigma, \perp, \rangle)$. The set δ is such that, for each transition $d = (p, \gamma_1, \gamma_2) \xrightarrow{a} (p', w_1, w_2) \in \Delta$ such that $a \neq f$, we add two transitions $(p^1, \gamma_1) \xrightarrow{\{d^1\}} (p'^1, w_1)$ and $(p^2, \gamma_2) \xrightarrow{\{d^2\}} (p'^2, w_2)$ to δ. If $a = f$, we add instead $(p^1, \gamma_1) \xrightarrow{\{f\}} (p'^1, w_1)$ and $(p^2, \gamma_2) \xrightarrow{\{f\}} (p'^2, w_2)$. Transitions in δ are projections of the original transitions of the 2-BVPDA on one of its two stacks; their label depends on the original transition in Δ, unless they are labelled by

f. Moreover, the transitions $(\sigma, \bot) \xrightarrow{\{\iota_1\}} (p_0^1, \bot)$ and $(\sigma, \bot) \xrightarrow{\{\iota_2\}} (p_0^2, \bot)$ both belong to δ.

$\mathcal{P}$ is such that it can either simulate the first or the second stack of the 2-BVPDA $\mathcal{BD}$, depending on which transition was used first. $\mathcal{P}$ is indeed a VPDS: a suitable partition of its input alphabet can be computed depending on which operation on the i-th stack transitions in Δ perform. As an example, if $d \in \Delta$ pushes a symbol on the first stack and pops from the second, d^1 belongs to the call alphabet and d^2, to the return alphabet.

Given a set of trace variables $\mathcal{V} = \{\pi_1, \pi_2\}$ and a predicate alphabet $AP = \{\iota^1, \iota^2, f\} \cup \Delta^1 \cup \Delta^2$, we then consider an unquantified HyperLTL formula φ of the form $\varphi = \varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$, where φ's sub-formulas are defined as follows:

Initialization formula: $\varphi_{start} = \iota^1_{\pi_1} \wedge \iota^2_{\pi_2}$; $\Pi \models \varphi_{start}$ if and only if for $i \in \{1, 2\}$, $\Pi[1, \infty](\pi_i)$ is a run that simulates the i-th stack of $\mathcal{BD}$;

Synchronization formula: $\varphi_{sync} = XG \bigwedge_{d \in \Delta} (d^1_{\pi_1} \Leftrightarrow d^2_{\pi_2})$; $\Pi \models \varphi_{start} \wedge \varphi_{sync}$ if and only if $\Pi[1, \infty](\pi_1)$ and $\Pi[1, \infty](\pi_2)$ can be matched to a common run of the 2-BVPDA $\mathcal{BD}$;

Acceptation formula: $\varphi_{end} = FG\,(f_{\pi_1} \wedge f_{\pi_2})$; $\Pi \models \varphi_{start} \wedge \varphi_{sync} \wedge \varphi_{end}$ if and only if $\Pi[1, \infty](\pi_1)$ and $\Pi[1, \infty](\pi_2)$ can be used to define an accepting run of the 2-BVPDA $\mathcal{BD}$.

Therefore, if $\Pi \models \varphi$, we have $\Pi(\pi_i) = (\iota^i, d^i_1, d^i_2 \ldots)$ for $i = 1, 2$, and the sequence of transitions $(d_1, d_2, \ldots) \in \Delta^\omega$ defines an accepting run on $\mathcal{BD}$. Therefore, we can solve the model-checking problem $\mathcal{P} \models \exists \pi_1, \exists \pi_2, \varphi$, if and only if we can determine whether $\mathcal{L}(\mathcal{BD})$ is empty or not, hence, $\mathcal{L}(\mathcal{D})$ as well. There is a contradiction and the former problem is undecidable. □

3.3 Model-checking constrained HyperLTL

By Theorem 4, the model-checking problem of HyperLTL against PDSs is undecidable. Intuitively, this issue stems from the undecidability of the intersection of context-free languages. However, since the emptiness problem of the intersection of a context-free language with regular sets is decidable, one can think of a way to abstract the set of runs of a PDS for some - but not all - path variables of a HyperLTL formula as a mean of regaining decidability.

As shown in [BS90, PW91] and detailed in Section 3.3.1, runs of a PDS can be over-approximated in a regular fashion. Hence, for a given PDS

$\mathcal{P}$, if we consider a regular abstraction of the set of runs $\alpha(Traces_\omega(\mathcal{P}))$, we can change the set assignment function for a path variable π in such a manner that $T(\pi) = \alpha(Traces_\omega(\mathcal{P}))$ instead of $T(\pi) = Traces_\omega(\mathcal{P})$.

For a set assignment function T on a set of path variables $\mathcal{V}$ and a variable $\pi \in \mathcal{V}$, we say that π is context-free w.r.t. to T if $T(\pi) = Traces_\omega(\mathcal{P})$ for some PDS $\mathcal{P}$. We define regular and visibly pushdown variables in a similar manner.

Let $\psi = \chi_0\pi_0, \ldots, \chi_n\pi_n, \varphi$ be a closed HyperLTL formula on the alphabet AP with $n+1$ trace variables $\pi_0, \ldots, \pi_n$, where $\chi_0 \ldots, \chi_n \in \{\forall, \exists\}$ are either universal or existential quantifiers. In this section, we will present a procedure to determine whether $T \models \psi$ in two cases.

1. If the variable π_0 is context-free w.r.t. T, and all the other variables are regular, then we can determine whether $T \models \psi$ or not. We can then apply this technique in order to approximate the model-checking problem if $T(\pi_0) = Traces_\omega(\mathcal{P})$, $\chi_1, \ldots, \chi_n = \forall$, and $T(\pi_j) = \alpha(Traces_\omega(\mathcal{P}))$ for $j = 1 \ldots n$. The last n variables can only be universally quantified.

   $T \models \psi$ then implies that $\mathcal{P} \models \psi$: indeed, the universal quantifiers on the path variables that range over the abstracted traces are such that, if the formula φ holds for every run in the over-approximation, then it also holds for every run in the actual set of traces. This is an approximation of the actual model-checking problem.

2. If there exists a variable π_i such that π_i is visibly context-free w.r.t. T, and all the other variables are regular, then we can determine whether $T \models \psi$ or not. A single path variable at most can be visibly context-free (not necessarily π_0, though), and all the others must be regular. We can then apply this technique in order to approximate the model-checking problem if $\mathcal{P}$ is a VPDS, $T(\pi_i) = Traces_\omega(\mathcal{P})$, $T(\pi_j) = \alpha(Traces_\omega(\mathcal{P}))$ and $\chi_j = \forall$ for $j \neq i$. Each path variable with the exception of the visibly context-free one must be universally quantified.

   Because of the universal quantifiers on the regular path variables, $T \models \psi$ implies again that $\mathcal{P} \models \psi$. This is an approximation of the model-checking problem.

Moreover, these approximations are accurate for at least one variable in the trace variable set, as the original, ω context-free (or ω visibly pushdown) set of runs is assigned to this variable instead of an ω regular over-approximation.

In a similar manner, if we compute a regular under-approximation $\beta(Traces_\omega(\mathcal{P}))$ of the set of traces, and if $T(\pi_0) = Traces_\omega(\mathcal{P})$, $T(\pi_j) =$

$\beta(Traces_\omega(\mathcal{P}))$ for $j = 1 \dots n$, and $\chi_1, \dots, \chi_n = \exists$, then $T \not\models \psi$ implies that $\mathcal{P} \not\models \psi$

3.3.1 Regular over-approximations of context-free languages

The infinite set $\mathcal{C}$ of configurations of $\mathcal{P}$ has to be reduced to a smaller, finite approximation. In order to do so, a common intuition is to define a congruence relation $\sim$ on $\mathcal{C}$ with a finite set of equivalence classes $\mathcal{C}/\sim$.

Let $\mathcal{I}_\sim = \{E \in \mathcal{C}/\sim \mid \exists c_i \in I, c_i \in E\}$ and let δ be a relation transition on $\mathcal{C}/\sim$ labelled by Δ such that $E_1 \xrightarrow{\lambda}_\delta E_2$ if and only $\exists c_1 \in E_1, \exists c_2 \in E_2, c_1 \overset{\lambda}{\Rightarrow} c_2$. The finite automaton $\mathcal{O} = (\Delta, \delta, \mathcal{C}/\sim, \mathcal{I}_\sim, \mathcal{C}/\sim)$ is then introduced as an over-approximation of $\mathcal{P}$.

One such congruence relation introduced by Bermudez et al. in [BS90] is called *fixed-depth investigation*: for a given integer d, $c_1 = \langle p_1, w_1\rangle \sim_d c_2 = \langle p_2, w_2\rangle$ if and only if $p_1 = p_2$ and $\exists w \in \Gamma^*$ such that either $|w| = d$ and $\exists w'_1, w'_2 \in \Gamma^*$, $w_1 = ww'_1$ and $w_2 = ww'_2$ or $|w| < d$ and $w_1 = w_2 = w$. Two configurations belong to the same class if they share the same state and the same topmost d stack letters; stacks that contains less than d elements each have a separate class.

Another method considered by Pereira and al. in [PW91] is a *reduction to a shorter stack*. If $c_1 = \langle p_1, w_1\rangle$ and w_1 is of the form $awaw'$, $a \in \Gamma$, $w, w' \in \Gamma^*$, then $c_1 \sim \langle p_1, aw'\rangle$. If a stack contains two occurrences of the same stack symbol $a \in \Gamma$, it is identified to a shorter stack where the part of the stack between the first and the second occurrence of a has been removed (including one of the a).

3.3.2 With one context-free variable and n regular variables

Let $\mathcal{P}$ be a PDS such that $T(\pi_0) = Traces_\omega(\mathcal{P})$, and $\mathcal{K}_1, \dots, \mathcal{K}_n$, finite state transition systems (i.e. finite automata without final states) such that for $i = 1, \dots, n$, $T(\pi_i) = Traces_\omega(\mathcal{K}_i)$.

Theorem 7. *If π_0 is context-free w.r.t. T and the other variables are regular, we can decide whether $T \models \chi_0\pi_0, \dots, \chi_n\pi_n, \varphi$ or not.*

To do so, we use the following result mentioned in Chapter 2:

Lemma 2. *Let φ be an LTL formula. There exists a Büchi automaton $\mathcal{B}_\varphi$ on the alphabet 2^{AP} such that $L(\mathcal{B}_\varphi) = \{w \in (2^{AP})^\omega \mid w \models \varphi\}$. We say that $\mathcal{B}_\varphi$ accepts φ.*

An *unquantified* HyperLTL formula with m trace variables $\pi_1, \ldots, \pi_m$ can be considered as a LTL formula on the alphabet $(2^{AP})^m$: given a word w on $(2^{AP})^m$ and $a \in AP$, we say that $w \models a_{\pi_i}$ if $a \in w_i(0)$, where w_i is the i-th component of w. We then apply Lemma 2 and introduce a Büchi automaton $\mathcal{B}_\varphi$ on the alphabet $(2^{AP})^{n+1}$ accepting φ. We denote $\Sigma = 2^{AP}$.

We then compute inductively a sequence of Büchi automata $\mathcal{B}_{n+1}, \ldots, \mathcal{B}_1$ such that:

— $\mathcal{B}_{n+1}$ is equal to the Büchi automaton $\mathcal{B}_\varphi$ on the alphabet Σ^{n+1};

— if the quantifier χ_i is equal to $\exists$ and $\mathcal{B}_{i+1} = (Q, \Sigma^{i+1}, \delta, q_0, F)$ is a Büchi automaton on the alphabet Σ^{i+1}, let $\mathcal{K}_i = (S, \Sigma, \delta', s_0)$ be the finite state transition system generating $T(\pi_i)$; we now define the Büchi automaton $\mathcal{B}_i = (Q \times S, \Sigma^i, \rho, (q_0, s_0), F \times S)$ where the set ρ of transitions is such that if $q \xrightarrow{(a_0,\ldots,a_i)} q' \in \delta$ and $s \xrightarrow{a_i} s' \in \delta'$, then $(q, s) \xrightarrow{(a_0,\ldots,a_{i-1})} (q', s') \in \rho$. Intuitively, the Büchi automaton $\mathcal{B}_i$ represents the formula $\exists \pi_i, \chi_{i+1}\pi_{i+1}, \ldots, \chi_n \pi_n, \varphi$; its input alphabet Σ^i depends on the number of variables that are not quantified yet;

— if the quantifier χ_i is equal to $\forall$, we consider instead the complement $\mathcal{B}'_{i+1}$ of $\mathcal{B}_{i+1}$ and compute its product with $\mathcal{K}_i$ in a similar manner to the previous construction; $\mathcal{B}_i$ is then equal to the complement of this product; intuitively, $\forall \pi, \psi = \neg(\exists \pi, \neg \psi)$.

Having computed $\mathcal{B}_1 = (Q, \Sigma, \delta, q_0, F)$, let $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, \langle p_0, \bot \rangle)$ be the PDS generating $T(\pi_0)$. We assume that $\chi_0 = \exists$. Let $\mathcal{BP} = (P \times Q, \Sigma, \Delta', \langle (p_0, q_0), \bot \rangle, P \times F)$ be a Büchi pushdown automaton, where the set of transitions Δ' is such that if $q \xrightarrow{a} q' \in \delta$ and $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta$, then $((p, q), \gamma) \xrightarrow{a} ((p', q'), w) \in \Delta'$. $\mathcal{BP}$ represents the fully quantified formula $\exists \pi_0, \chi_1 \pi_1, \ldots, \chi_n \pi_n, \varphi$. Obviously, $\mathcal{BP}$ is not empty if and only if $T \models \psi$.

If $\chi_0 = \forall$, we consider instead the complement $\mathcal{B}'_1$ of $\mathcal{B}_1$, then define a Büchi pushdown automaton $\mathcal{BP}$ in a similar manner. $\mathcal{BP}$ is empty if and only if $T \models \psi$.

It has been proven in [BEM97, EHRS00] that the emptiness problem is decidable for Büchi pushdown automata. Hence, given our initial constraints on T and ψ, we can determine whether $T \models \psi$ or not. □

The Büchi automaton $\mathcal{B}_\varphi$ has $O(2^{|\varphi|})$ states; if we assume that all variables are existentially quantified, the BPDS $\mathcal{BP}$ has $\nu = O(2^{|\varphi|}|\mathcal{P}||\mathcal{K}_1| \dots |\mathcal{K}_n|)$ states. According to [EHRS00], checking the emptiness of $\mathcal{BP}$ can be done in $O(\nu^2 k)$ operations, where k is the number of transitions of $\mathcal{BP}$, hence, in $O(\nu^4|\Gamma|^2)$.

Complementation of a Büchi Automaton may increase its size exponentially; hence, this technique may incur an exponential blow-up depending on the number of universal quantifiers.

Application. If we consider that π_0 range over $Traces_\omega(\mathcal{P})$ and that $\pi_1, \dots, \pi_n$ range over a regular abstraction $\alpha(Traces_\omega(\mathcal{P}))$ of the actual set of traces, and we assume that $\chi_1, \dots, \chi_n = \forall$, we can apply this result to approximate the model-checking problem, as detailed earlier in this section.

It is worth noting that the complement of an ω context-free language is not necessarily an ω context-free language. Hence, we can't use the previous procedure to check a HyperLTL formula of the form $\psi = \exists\pi, \forall\pi' \varphi$ where π' is a context-free variable and π is regular. We know, however, that ω visibly pushdown languages are closed under complementation. We therefore consider the case of a single visibly pushdown variable in the following subsection.

3.3.3 With one visibly pushdown variable and n regular variables

Let $\mathcal{P}$ be a VPDS such that $T(\pi_i) = Traces_\omega(\mathcal{P})$, and $(\mathcal{K}_j)_{j \neq i}$ finite state transition systems such that for $j \neq i$, $T(\pi_j) = Traces_\omega(\mathcal{K}_j)$. Unlike the previous case, the visibly context-free variable no longer has to be the first one π_0.

Theorem 8. *If a variable π_i is visibly pushdown w.r.t. T and the other variables are regular, we can decide whether $T \models \chi_0\pi_0, \dots, \chi_n\pi_n, \varphi$ or not.*

The proof of this theorem is similar to the proof of Theorem 7. We first build a sequence of Büchi automata $\mathcal{B}_{n+1}, \dots, \mathcal{B}_{i+1}$ in a similar manner to the proof of Theorem 7, starting from a finite state automaton $\mathcal{B}_{n+1} = \mathcal{B}_\varphi$ on the alphabet Σ^{n+1} representing the unquantified formula φ then computing products with the transition systems $\mathcal{K}_{n+1}, \dots, \mathcal{K}_{i+1}$ until we end up with a Büchi automaton $\mathcal{B}_{i+1}$ on the alphabet Σ^{i+1}.

Having computed $\mathcal{B}_{i+1} = (Q, \Sigma^{i+1}, \delta, q_0, F)$, we define the VPDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, \langle p_0, \bot\rangle)$ generating $T(\pi_i)$. We assume that $\chi_i = \exists$. Let $\mathcal{BP}_i = (P \times Q, \Sigma^{i+1}, \Delta', \langle (p_0, q_0), \bot\rangle, P \times F)$ be a visibly Büchi pushdown automaton, where Δ' is such that if $q \xrightarrow{(a_0, \dots, a_{i-1}, a)} q' \in \delta$ and

$(p,\gamma) \xrightarrow{a} (p',w) \in \Delta$, then $((p,q),\gamma) \xrightarrow{(a_0,\ldots,a_{i-1},a)} ((p',q'),w) \in \Delta'$. $\mathcal{BP}_i$ is indeed a BVPDA on the alphabet Σ^{i+1} as its stack operations only depend on its $i+1$-th variable. If $\chi_i = \forall$, we consider instead the complement $\mathcal{B'}_{i+1}$.

From the i-th variable onwards, we compute a sequence of visibly Büchi pushdown automata $\mathcal{BP}_i, \ldots, \mathcal{BP}_0$ on the alphabets $\Sigma^{i+1}, \ldots, \Sigma^i$ respectively. For $i \geq k \geq 1$, if $\mathcal{BP}_k = (P', \Sigma^{k+1}, \Delta', \langle p'_0, \bot\rangle, F')$, $\mathcal{K}_i = (S, \Sigma, \delta, s_0)$, and $\chi_k = \exists$, let $\mathcal{BP}_{k-1} = (P' \times S, \Sigma^k, \Delta'', \langle (p'_0, s_0), \bot\rangle, F' \times S)$ be a visibly Büchi pushdown automaton, where the set of transitions Δ'' is such that if $(p,\gamma) \xrightarrow{(a_0,\ldots,a_{k-1},a_i)} (p',w) \in \Delta'$ and $q \xrightarrow{a_{k-1}} q' \in \delta$, then $((p,q),\gamma) \xrightarrow{(a_0,\ldots,a_{k-2},a_i)} ((p',q'),w) \in \Delta''$. The last letter of each tuple always stands for the visibly pushdown path variable π_i: $\mathcal{BP}_{k-1}$ is visibly pushdown as its stack operations only depend on this variable. If $\chi_k = \forall$, we consider the complement $\mathcal{BP}'_k$ of $\mathcal{BP}_k$ instead, which is a visibly pushdown automaton as well, as proven in [AM04].

We can check the emptiness of $\mathcal{BP}_0$. If it is indeed empty, then $T \models \psi$. □

It has been proven in [AM04] that the complement of a VPDA incurs an exponential blow-up in terms of states. Hence, the technique shown here is exponential (in terms of time) in the size of $\mathcal{P}$ and φ.

Application. If we consider that π_i range over $Traces_\omega(\mathcal{P})$ and that $\pi_j, j \neq i$ range over a regular abstraction $\alpha(Traces_\omega(\mathcal{P}))$ of the actual set of traces, and we assume that $\chi_j = \forall$ for $j \neq i$, we can apply this result to approximate the model-checking problem, as detailed earlier in this section.

3.4 Model-checking HyperLTL with bounded phases

In this section, we use results on *multi-stack pushdown automata* to approximate the model-checking problem of HyperLTL formulas with universal quantifiers against PDSs.

3.4.1 Multi-stack pushdown systems

Multi-stack pushdown systems (MPDSs) are pushdown systems with multiple stacks. Their semantics are defined in a manner similar to PDSs, and so are configurations, traces, runs, *multi-stack pushdown automata* (MPDA), and the semantics of LTL.

Definition 12 (La Torre et al. [TMP07])**.** *A* multi-stack pushdown system *(or MPDS) is a quadruplet $\mathcal{M} = (P, \Gamma, l, \Delta)$ where P is a finite set of control states, Γ is a finite stack alphabet, l is the number of stacks, and $\Delta \subset P \times \Gamma \times \{1, \ldots, l\} \times P \times \Gamma^*$ a finite set of transition rules.*

For a given transition of a MPDS, in a given control state, only one stack is read and modified. A rule of the form $(p, w, n) \rightarrow (p', w')$ is applied to the n-th stack with semantics similar to those of common pushdown systems.

A *configuration* of $\mathcal{M}$ is an element of $P \times (\Gamma^*)^l$. A set of configurations $\mathcal{C}$ is said to be *regular* if for all $p \in P$, there exists a finite-state automaton $\mathcal{A}_p$ on the alphabet $\{\#\} \cup \Gamma$ such that $\mathcal{L}(\mathcal{A}_p) = \{w_1\# \ldots \#w_l \mid \langle p, w_1, \ldots, w_l \rangle \in \mathcal{C}\}$.

We define a successor relation $\hookrightarrow_{\mathcal{M}}$ on configurations. If $\delta = (p, a, i) \rightarrow (p', w) \in \Delta$, then for each configuration $c = \langle p, w_1, \ldots, w_l \rangle$ such that $w_i = ax$, we have $c \overset{\delta}{\hookrightarrow} \langle p', w'_1, \ldots, w'_l \rangle$ where $w'_i = wx$ and $w'_j = w_j$ if $j \neq i$. $\hookrightarrow^*_{\mathcal{M}}$ is the reflexive and transitive closure of the relation $\hookrightarrow_{\mathcal{M}} = (\bigcup_{\delta \in \Delta} \overset{\delta}{\hookrightarrow})$. We may ignore the variable $\mathcal{M}$ if only a single MPDS is being considered. Without loss of generality, we can assume n stacks can be modified by a single transition instead of using n different transitions in a row.

For a given set of configurations $\mathcal{C}$ of a MPDS $\mathcal{M}$, we define its set of *predecessors* $pre^*_{\text{MPDS}}(\mathcal{M}, \mathcal{C}) = \{c \in P \times (\Gamma^*)^l \mid \exists c' \in \mathcal{C}, c \hookrightarrow^* c'\}$.

A *run* r of $\mathcal{M}$ from a configuration c_0 is a sequence of configurations $r = (c_i)_{i=1,\ldots,n} \in \Delta^*$ such that $c_0 \overset{t_1}{\hookrightarrow} c_1 \overset{t_2}{\hookrightarrow} c_2 \ldots \overset{t_n}{\hookrightarrow} c_n$, where $t = (t_i)_{i=1,\ldots,n}$ is a sequence of transition rules of $\mathcal{M}$ called the *trace* of r. We then write $c_0 \overset{t}{\hookrightarrow} c_n$.

Multi-stack automata are unfortunately Turing powerful, even with only two stacks. La Torre et al. thus introduced in [TMP07] a new restriction called *phase-bounding*:

Definition 13. *A configuration c' of $\mathcal{M}$ is said to be reachable from another configuration c in* k phases *if there exists a sequence of runs $r_1, r_2, \ldots r_k$ with matching traces $t_1, t_2, \ldots t_k$ such that $c_0 \overset{t_1}{\hookrightarrow} c_1 \ldots \overset{t_k}{\hookrightarrow} c_k$ where $c_0 = c$, $c_k = c'$, $(c_i)_{i=1,\ldots,k}$ is a sequence of configurations on $\mathcal{M}$ and where during the execution of a given run r_i, at most a single stack is popped from. We note $c \hookrightarrow^*_{\mathcal{M},k} c'$.*

We then define $pre^*_{\text{MPDS}}(\mathcal{M}, \mathcal{C}, k) = \{c \in P \times (\Gamma^*)^l \mid \exists c' \in \mathcal{C}, c \hookrightarrow^*_{\mathcal{M},k} c'\}$. The following theorem has been proven in [Set10]:

Theorem 9. *Given a MPDS $\mathcal{M}$ and a regular set of configurations $\mathcal{C}$, the set $pre^*_{MPDS}(\mathcal{M}, \mathcal{C}, k)$ is regular and effectively computable.*

This property can then be used to show that the following theorem holds:

Theorem 10 (Anil Seth [Set10])**.** *The model-checking problem of the logic LTL against MPDSs with bounded phases is decidable.*

3.4.2 Application to HyperLTL model-checking

Phase-bounding can be used to under-approximate the set of traces of a MPDS. If a given LTL property φ does not hold for a MPDS $\mathcal{M}$ with a phase-bounding constraint, it does not hold for the MPDS $\mathcal{M}$ w.r.t. the usual semantics as well. We write $\mathcal{M} \models_k \varphi$ if the LTL formula φ holds for traces of $\mathcal{M}$ with at most k phases.

We can use decidability properties of MPDSs with bounded phases to approximate the model-checking problem of HyperLTL against pushdown systems. Let $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0\rangle)$ be a PDS on the input alphabet $\Sigma = 2^{AP}$, and $\psi = \forall \pi_1, \ldots, \forall \pi_n, \varphi$, a HyperLTL formula on n trace variables with only universal quantifiers.

Our intuition is to define a MPDS $\mathcal{M}$ such that each stack represents a path variable of the HyperLTL formula. This MPDS is the product of n copies of $\mathcal{P}$. Because ψ features *universal quantifiers* only, the model-checking problem of the LTL formula φ for $\mathcal{M}$ is then equivalent to the model-checking problem of ψ for $\mathcal{P}$: $\mathcal{M}$ simulates n runs of $\mathcal{P}$ simultaneously, hence, LTL formulas on $\mathcal{M}$ can be used to synchronize these runs. We can therefore use a phase-bounded approximation of the former problem to approximate the latter.

We introduce the MPDS $\mathcal{M} = (P^n, \Sigma^n, \Gamma^n, n, \Delta', c'_0)$, with an initial configuration $c'_0 = \langle (p_0, \ldots, p_0), \bot, \ldots, \bot \rangle) \in P^n \times \Gamma^n$ and a set of transitions Δ' defined as follows: $\forall d_1, \ldots, d_n \in \Delta^n$ where $d_i = (p_i, \gamma_i) \xrightarrow{a_i} (p'_i, w_i)$ for $i = 1, \ldots, n$, the transition $((p_1, \ldots, p_n), \gamma_1, \ldots, \gamma_n) \xrightarrow{(a_1, \ldots, a_n)} ((p'_1, \ldots, p'_n), w_1, \ldots, w_n)$ belongs to Δ'. The following lemma holds:

Lemma 3. *$\mathcal{M} \models \varphi$ if and only if $P \models \psi$.*

As a consequence, if $\mathcal{M} \not\models \varphi$, then $P \not\models \psi$. We then consider a phase-bounded analysis of $\mathcal{M}$: for a given integer k, if $\mathcal{M} \not\models_k \varphi$, then $\mathcal{M} \not\models \varphi$, hence $P \not\models \psi$. We can therefore compute an approximation of the model-checking problem of HyperLTL formulas with universal quantifiers only.

3.5 Applications to security properties

We apply in this section our results to information flow security, and remind how, as shown in [CFK+14], security policies can be expressed as HyperLTL formulas. If we model a given program as a PDS or a VPDS $\mathcal{P}$ following the method outlined in [EHRS00], we can either approximate an answer to the model-checking problem $\mathcal{P} \models \psi$ of a policy represented by a HyperLTL formula ψ for this program.

3.5.1 Observational determinism

The *strict non-interference* security policy is the following: an attacker should not be able to distinguish two computations from their outputs if they only differ in their secret inputs. Few actual programs meet this requirement, and different versions of this policy have thus been defined.

We partition variables of a program into high and low security variables, and into input and output variables. The *observational determinism* property holds if, assuming two starting configurations have identical low security input variables, their low security output variables will be equal as well.

We model the program as a PDS $\mathcal{P}$ on the input alphabet 2^{AP}, where atomic propositions in AP contain variable values: if a variable x can take a value a, then $(x, a) \in AP$. We can express the observational determinism policy as the following HyperLTL formula:

$$\psi_{OD} = \forall \pi_1, \forall \pi_2, \left(\bigwedge_{a \in LS_i} (a_{\pi_1} \Leftrightarrow a_{\pi_2}) \right) \Rightarrow G \left(\bigwedge_{b \in LS_o} (b_{\pi_1} \Leftrightarrow b_{\pi_2}) \right)$$

where LS_i (resp. LS_o) is the set of low security input (resp. output) variables values. Using our techniques detailed in Sections 3.4 and 3.3.2, we can approximate the problem $\mathcal{P} \models \psi_{OD}$ that is otherwise undecidable.

A context-free example. Let $AP = \{i, o, h_1, h_2\}$, $LS_i = \{i\}$, $LS_o = \{o\}$, and let $HS_i = \{h_1, h_2\}$ be a set of high security inputs. We suppose we are given a program that can be abstracted by the following PDS $\mathcal{P}$ on the alphabet $\Sigma = 2^{AP}$, the stack alphabet $\Gamma = \{\gamma, \bot\}$, and the set of states $P = \{p_0, p_1, p_2, p_3, p_4\}$, with the following set of transitions, as

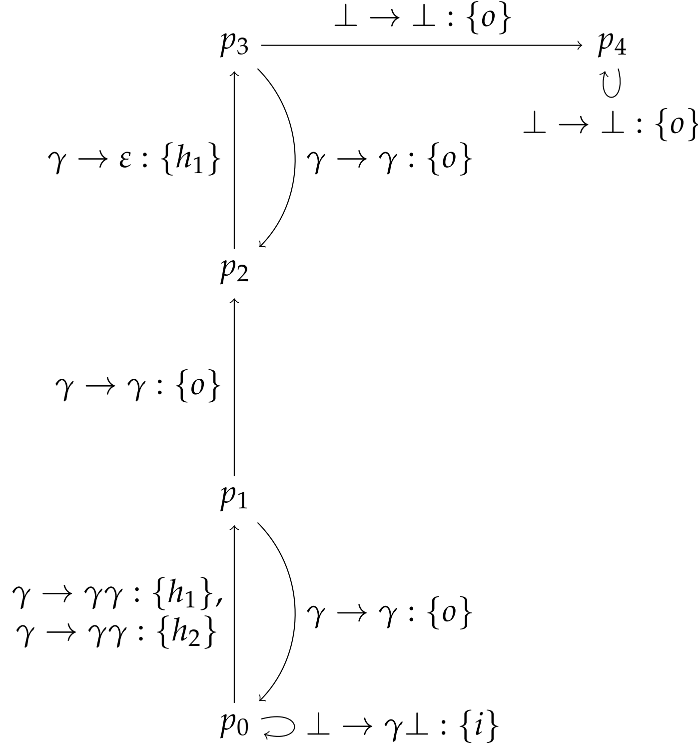

FIGURE 3.1: Checking observational determinism on the PDS $\mathcal{P}$

represented by Figure 3.1:

$$
\begin{array}{llll}
(init) & (p_0, \bot) \xrightarrow{\{i\}} (p_0, \gamma\bot) & (\mu_2) & (p_2, \gamma) \xrightarrow{\{h_1\}} (p_3, \varepsilon) \\
(\lambda_1) & (p_0, \gamma) \xrightarrow{\{h_1\}} (p_1, \gamma\gamma) & (\mu_3) & (p_3, \gamma) \xrightarrow{\{o\}} (p_2, \gamma) \\
(\lambda_2) & (p_0, \gamma) \xrightarrow{\{h_2\}} (p_1, \gamma\gamma) & (\nu_1) & (p_3, \bot) \xrightarrow{\{o\}} (p_4, \bot) \\
(\lambda_3) & (p_1, \gamma) \xrightarrow{\{o\}} (p_0, \gamma) & (\nu_2) & (p_4, \bot) \xrightarrow{\{o\}} (p_4, \bot) \\
(\mu_1) & (p_1, \gamma) \xrightarrow{\{o\}} (p_2, \gamma) & &
\end{array}
$$

We would like to check if $P \models \psi_{OD}$, where ψ_{OD} is the observational determinism HyperLTL formula outlined above. Intuitively, it will not hold: two runs always have the same input i but, if they do not push the same number of symbols on the stack, their low-security outputs will differ.

Since transitions of $\mathcal{P}$ are only labelled by singletons, we can write ρ instead of $\{\rho\}$ when describing traces. The set $Traces_\omega(\mathcal{P})$ of infinite traces of $\mathcal{P}$ is equal to $\bigcup_{n \in \mathbb{N}} i \cdot ((h_1 + h_2) \cdot o)^n \cdot (h_1 \cdot o)^{n+1} \cdot o^*$: from the bottom symbol $\bot$, rules $(init)$, (λ_1), (λ_2), and (λ_3) push $n+1$ symbols γ on the stack, then rules (μ_1), (μ_2), and (μ_3) pop these $(n+1)$

symbols, and finally rule (ν_2) loop in state p_4 once the bottom of the stack is reached again and rule (ν_1) has been applied. $Traces_\omega(\mathcal{P})$ is context-free, hence, we can't model-check the observational determinism policy on $\mathcal{P}$ using the algorithms outlined in [CS10].

Using the approximation technique outlined in Section 3.4, we can show that ψ_{OD} does not hold if we bound the number of phases to 2: we find a counter-example $\pi_1 = i \cdot h_2 \cdot o \cdot h_1 \cdot o \cdot o^*$ and $\pi_2 = i \cdot (h_2 \cdot o)^2 \cdot (h_1 \cdot o)^2 \cdot o^*$. We can therefore reach the conclusion that $\mathcal{P} \not\models \psi_{OD}$; the observational determinism security policy therefore does not hold for the original program.

3.5.2 Declassification

The strict non-interference security policy is very hard to enforce as many programs must, one way or another, leak secret information during their execution. Thus, we must relax the security properties defined previously.

We introduce instead a *declassification* policy: at a given step, leaking a specific high security variable is allowed, but the observational determinism must otherwise holds. As an example, let's consider a program accepting a password as a high security input in its initial state, whose correctness is then checked during the next execution step. The program's behaviour then depends on the password's correctness. We express this particular declassification policy as the following HyperLTL formula:

$$\begin{aligned}\psi_D = \quad & \forall \pi_1, \forall \pi_2, ((\bigwedge_{a \in LS_i} (a_{\pi_1} \Leftrightarrow a_{\pi_2})) \wedge X\,(\rho_{\pi_1} \Leftrightarrow \rho_{\pi_2})) \\ & \Rightarrow G\,(\bigwedge_{b \in LS_o} (b_{\pi_1} \Leftrightarrow b_{\pi_2}))\end{aligned}$$

where ρ is a high security atomic proposition specifying that an input password is correct. Again, using our techniques detailed in Sections 3.4 and 3.3.2, we can both approximate the model-checking problem $\mathcal{P} \models \psi_D$.

Checking a password. We consider a program where the user can input a low-security username and a high-security password, then get different outputs depending on whether the password is true or not.

Let $AP = \{u, pw_1, pw_2, pw_3, o, \rho, h_1, h_2\}$, $LS_i = \{u\}$, $LS_o = \{o\}$, let ρ be a variable that is allowed to leak, and let $HS_i = \{pw_1, pw_2, pw_3, h_1, h_2\}$

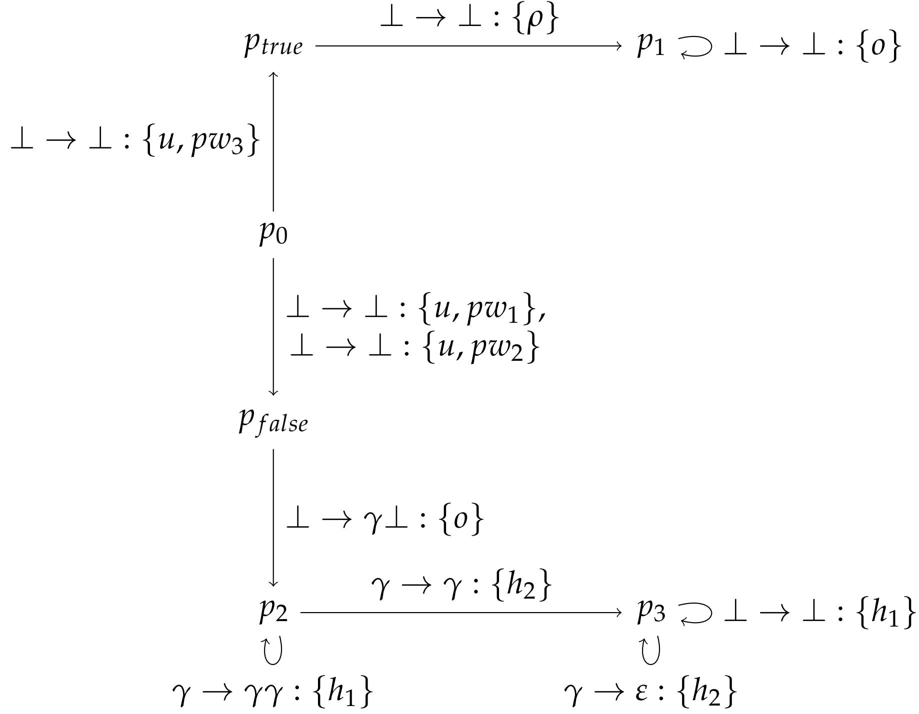

FIGURE 3.2: Checking declassification on the PDS $\mathcal{P}$

be a set of high security inputs. Assuming there is only a single username u and three possible passwords pw_1, pw_2, pw_3, the last password pw_3 being the only right answer, we can consider the following PDS $\mathcal{P}$ on the alphabet $\Sigma = 2^{AP}$, the stack alphabet $\Gamma = \{\gamma, \bot\}$, the set of states $P = \{p_0, p_1, p_2, p_3, p_{true}, p_{false}\}$, with the following set of transitions, as represented by Figure 3.2:

$$
\begin{array}{llll}
(init_1) & (p_0, \bot) \xrightarrow{\{u,pw_1\}} (p_{false}, \bot) & (\mu_1) & (p_1, \bot) \xrightarrow{\{o\}} (p_1, \bot) \\
(init_2) & (p_0, \bot) \xrightarrow{\{u,pw_2\}} (p_{false}, \bot) & (\mu_2) & (p_2, \gamma) \xrightarrow{\{h_1\}} (p_2, \gamma\gamma) \\
(init_3) & (p_0, \bot) \xrightarrow{\{u,pw_3\}} (p_{true}, \bot) & (\mu_3) & (p_2, \gamma) \xrightarrow{\{h_2\}} (p_3, \gamma) \\
(pw_{true}) & (p_{true}, \bot) \xrightarrow{\{\rho\}} (p_1, \bot) & (\mu_4) & (p_3, \gamma) \xrightarrow{\{h_2\}} (p_3, \varepsilon) \\
(pw_{false}) & (p_{false}, \bot) \xrightarrow{\{o\}} (p_2, \gamma\bot) & (\mu_2) & (p_3, \bot) \xrightarrow{\{h_1\}} (p_3, \bot)
\end{array}
$$

We would like to check if $P \models \psi_D$, where ψ_D is the declassification HyperLTL formula outlined above. Obviously, if we consider that $\rho \in LS_o$, then observational determinism does not hold: given the same username u, depending on whether the high-security password p_i chosen is right or not, the low-security output will differ. However, intuitively, the declassification policy should hold: given two different input passwords, the PDS will behave in the same manner as long as both are either true or false.

The set $Traces_\omega(\mathcal{P})$ of infinite traces of $\mathcal{P}$ is equal to $(\{u, pw_3\} \cdot \{\rho\} \cdot \{o_1\}^*) \cup \bigcup_{n \in \mathbb{N}} ((\{u, pw_1\} + \{u, pw_2\}) \cdot \{o\} \cdot \{h_1\}^n \cdot \{h_2\}^{n+2} \cdot \{h_1\}^*)$: if the right password pw_3 has been input from the bottom symbol $\bot$, then rules $(init_3)$ and (p_{true}) lead to state p_1 where the PDS loops; otherwise, if the password is wrong, rules $(init_1)$, $(init_2)$ and (p_{false}) push a symbol γ and lead to state p_2, where rule (μ_2) pushes n symbols γ on the stack, then the PDS switches to state p_3 where it pops these $(n+1)$ symbols with rules (μ_3) and (μ_4) then loops with rule (μ_5) once the bottom of the stack has been reached. $Traces_\omega(\mathcal{P})$ is context-free, hence, we can't model-check the declassification policy on $\mathcal{P}$ using the algorithms outlined in [CS10].

Using the approximation techniques detailed in Section 3.3.2, we can consider the regular abstraction $\alpha(Traces_\omega(\mathcal{P})) = (\{u, p_3\} \cdot \{\rho\} \cdot \{o_1\}^*) \cup ((\{u, p_1\} + \{u, p_2\}) \cdot \emptyset \cdot \{h_1\}^* \cdot \{h_2\}^* \cdot \{h_1\}^*)$ of the actual set of traces. We can then reach the conclusion that $\mathcal{P} \models \psi_D$, since this property holds for the approximation as well; the declassification security policy therefore holds for this example.

3.5.3 Non-inference

Non-inference is a variant of the non-interference security policy. It states that, should all high security input variables be replaced by a dummy input λ, the behaviour of low security variables should not change.

We express this property as the following HyperLTL formula:

$$\psi_{NI} = \forall \pi_1, \exists \pi_2, G\,(\bigwedge_{x \in HS_i} (x, \lambda)_{\pi_2}) \wedge G\,(\bigwedge_{b \in LS} (b_{\pi_1} \Leftrightarrow b_{\pi_2}))$$

where LS stands for the set of all low security variables values, HS_i for the set of high security input variables, and (x, λ) means that variable x has value λ. We can't rely on the method outlined in 3.3.2 because π_2 is existentially quantified, but an approximation can nonetheless be found using the method detailed in Section 3.3.3, if we model the program as a VPDS $\mathcal{P}$, choose π_2 as the visibly context-free path variable, and make it so that π_1 ranges over a regular abstraction of the traces.

3.6 Related work

Clarkson and Schneider introduced *hyperproperties* in [CS10] to formalize security properties, using second-order logic. Unfortunately, this

logic isn't verifiable in the general case.

However, some fragments of it can be verified: in [CFK+14], Clarkson et al. formalized the temporal logics HyperLTL and HyperCTL*, extending the widespread and flexible framework of linear-time and branching time logics to hyperproperties. The model-checking problem of these logics against finite state systems has been shown to be decidable by a reduction to the satisfiability problem for the quantified propositional temporal logic QPTL defined in [SVW87].

Proper model-checking algorithms were introduced by Finkbeiner et al. in [FRS15]. These algorithms follow the automata-theoretic framework defined by Vardi et al. in [Var96], and can be used to verify security policies in circuits. However, while circuits can be modelled as finite state systems, actual programs can feature recursive procedure calls and infinite recursion. Hence, a more expressive model such as PDSs is needed.

In [BEM97, EHRS00], the forward and backward reachability sets of PDSs have been shown to be regular and effectively computable. As a consequence, the model-checking problem of LTL against PDSs is decidable; an answer can be effectively computed using an automata-theoretic approach. We try to extend this result to HyperLTL.

Multi-stack pushdown systems (MPDSs) are unfortunately Turing powerful. Following the work of Qadeer et al. in [QR05], La Torre et al. introduced in [TMP07] MPDSs with *bounded phases*: a run is split into a finite number of phases during which there is at most one stack that is popped from. Anil Seth later proved in [Set10] that the backward reachability set of a multi-stack pushdown system with bounded phases is regular; this result can then be used to solve the model checking problem of LTL against MPDSs with bounded phases. We rely on a bounded-phase analysis of a MPDS to approximate an answer to the model-checking problem of HyperLTL against PDSs.

3.7 Conclusion

In this chapter, we study the model-checking problem of hyper properties expressed by the logic HyperLTL against PDSs. We show that it is undecidable, even for the sub-class of visibly pushdown automata. We therefore design an automata-theoretic framework to abstract the model-checking problem given some constraints on the use of universal quantifiers in the HyperLTL formula. We also use phase-bounding

constraints on multi-stack pushdown automata to approximate the actual answer. Finally, we show some relevant examples of security properties that cannot be expressed with LTL but can be checked using our approximation algorithms on a HyperLTL formula.

Chapter 4

Reachability Analysis of Pushdown Systems with an Upper Stack

As mentioned previously, *pushdown systems* (PDSs) are a natural model for sequential programs, but they can fail to accurately represent the way an assembly stack actually operates. Indeed, one may want to access the part of the memory that is below the current stack or base pointer, hence the need for a model that keeps track of this part of the memory.

To this end, we introduce *pushdown systems with an upper stack* (UPDSs), an extension of PDSs where symbols popped from the stack are not destroyed but instead remain just above its top, and may be overwritten by later push rules. We prove that the sets of successors $post^*$ and predecessors pre^* of a regular set of configurations of such a system are not always regular, but that $post^*$ is context-sensitive, so that we can decide whether a single configuration is forward reachable or not.

In order to under-approximate pre^* in a regular fashion, we consider a bounded-phase analysis of UPDSs, where a phase is a part of a run during which either push or pop rules are forbidden. We then present a method to over-approximate $post^*$ that relies on regular abstractions of runs of UPDSs. Finally, we show how these approximations can be used to detect stack overflows and stack pointer manipulations with malicious intent.

Chapter outline. We define in Section 1 a new class of pushdown systems called *pushdown systems with an upper stack*. We prove in Section 2 that neither the set of predecessors nor the set of successors of a regular set of configurations are regular, but that the set of successors is nonetheless context-sensitive. Then, in Section 3, we prove that the set of predecessors of an UPDS is regular given a *phase-bounding* constraint. In Section 4, we give an algorithm to compute an over-approximation of the set of successors. In Section 5, we show how

these approximations could be applied to find errors or security flaws in programs. Finally, we describe the related work in Section 6 and show our conclusion in Section 7.

These results were published in [PDT17].

4.1 Pushdown systems with an upper stack

Definition 14. *A* pushdown system with an upper stack *(UPDS) is a triplet $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of control states, Γ is a finite stack alphabet, and $\Delta \subseteq P \times \Gamma \times P \times (\{\varepsilon\} \cup \Gamma \cup \Gamma^2)$ a finite set of transition rules.*

We further note $\Delta_{pop} = \Delta \cap P \times \Gamma \times P \times \{\varepsilon\}$, $\Delta_{switch} = \Delta \cap P \times \Gamma \times P \times \Gamma$, and $\Delta_{push} = \Delta \cap P \times \Gamma \times P \times \Gamma^2$. If $\delta = (p, w, p', w') \in \Delta$, we write $\delta = (p, w) \to (p', w')$. In a UPDS, a write-only *upper stack* is maintained above the stack used for computations (from then on called the *lower stack*), and modified accordingly during a transition.

For $x \in \Gamma$ and $w \in \Gamma^*$, $|w|_x$ stands for the number of times the letter x appears in the word w, and w^R for the mirror image of w. Let $\overline{\Gamma}$ be a disjoint copy (bijection) of the stack alphabet Γ. If $x \in \Gamma$ (resp. Γ^*), then its associated letter (resp. word) in $\overline{\Gamma}$ (resp. $\overline{\Gamma}^*$) is written $\overline{x}$.

A *configuration* of $\mathcal{P}$ is a triplet $\langle p, w_u, w_l \rangle$ where $p \in P$ is a control state, $w_u \in \Gamma^*$ an *upper stack content*, and $w_l \in \Gamma^*$ a *lower stack content*. Let $Conf_{\mathcal{P}} = P \times \Gamma^*$ be the set of configurations of $\mathcal{P}$.

A set of configurations $\mathcal{C}$ of a UPDS $\mathcal{P}$ is said to be *regular* if for all $p \in P$, there exists a finite-state automaton $\mathcal{A}_p$ on the alphabet $\overline{\Gamma} \cup \Gamma$ such that $\mathcal{L}(\mathcal{A}_p) = \{\overline{w_u} w_l \mid \langle p, w_u, w_l \rangle \in \mathcal{C}\}$, where $\mathcal{L}(\mathcal{A})$ stands for the language recognized by an automaton $\mathcal{A}$.

From the set of transition rules Δ, we can infer an *immediate successor relation* $\to_{\mathcal{P}} = (\bigcup_{\delta \in \Delta} \xrightarrow{\delta})$ on configurations of $\mathcal{P}$, which is defined as follows:

Switch rules: if $\delta = (p, \gamma) \to (p', \gamma') \in \Delta_{switch}$, then $\forall w_u \in \Gamma^*$ and $\forall w_l \in \Gamma^*$, $\langle p, w_u, \gamma w_l \rangle \xrightarrow{\delta} \langle p', w_u, \gamma' w_l \rangle$. The top letter γ of the lower stack is replaced by γ', but the upper stack is left untouched (the stack pointer doesn't move).

Pop rules: if $\delta = (p, \gamma) \to (p', \varepsilon) \in \Delta_{pop}$, then $\forall w_u \in \Gamma^*$ and $\forall w_l \in \Gamma^*$, $\langle p, w_u, \gamma w_l \rangle \overset{\delta}{\Rightarrow} \langle p', w_u \gamma, w_l \rangle$. The top letter γ popped from the lower stack is added to the bottom of the upper stack (the stack pointer moves to the right), as shown in Figure 4.1.

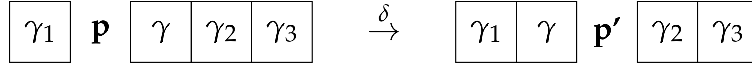

FIGURE 4.1: Semantics of pop rules.

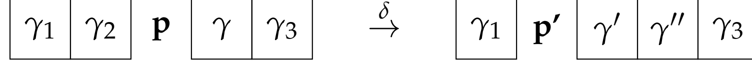

FIGURE 4.2: Semantics of push rules.

Push rules: if $\delta = (p, \gamma) \rightarrow (p', \gamma'\gamma'') \in \Delta_{push}$, then $\forall w_l \in \Gamma^*$, $\forall w_u \in \Gamma^*$, $\langle p, \varepsilon, \gamma w_l \rangle \xrightarrow{\delta} \langle p', \varepsilon, \gamma'\gamma'' w_l \rangle$ and $\forall x \in \Gamma$, $\langle p, w_u x, a w_l \rangle \xrightarrow{\delta} \langle p', w_u, \gamma'\gamma'' w_l \rangle$. A new letter b is pushed on the lower stack, and a single letter is deleted from the bottom of the upper stack in order to make room for it, unless the upper stack was empty (the stack pointer moves to the left), as shown in Figure 4.2.

The *reachability* relation $\Rightarrow_{\mathcal{P}}$ is the reflexive and transitive closure of the immediate successor relation $\rightarrow_{\mathcal{P}}$. If $\mathcal{C}$ is a set of configurations, we introduce its set of *successors* $post^*(\mathcal{P}, \mathcal{C}) = \{c \in P \times \Gamma^* \times \Gamma^* \mid \exists c' \in \mathcal{C}, c' \Rightarrow_{\mathcal{P}} c\}$ and its set of *predecessors* $pre^*(\mathcal{P}, \mathcal{C}) = \{c \in P \times \Gamma^* \times \Gamma^* \mid \exists c' \in \mathcal{C}, c \Rightarrow_{\mathcal{P}} c'\}$. We may omit the variable $\mathcal{P}$ when only a single UPDS is being considered. Note that the combined size of the two stacks keeps growing after each transition.

For a set of configurations $\mathcal{C}$, let $\mathcal{C}_{low} = \{\langle p, w_l \rangle \mid \exists w_u \in \Gamma^*, \langle p, w_u, w_l \rangle \in \mathcal{C}\}$ and $\mathcal{C}_{up} = \{\langle p, w_u \rangle \mid \exists w_l \in \Gamma^*, \langle p, w_u, w_l \rangle \in \mathcal{C}\}$. We then define $post^*_{up}(\mathcal{P}, \mathcal{C}) = (post^*(\mathcal{P}, \mathcal{C}))_{up}$, as well as $post^*_{low}(\mathcal{P}, \mathcal{C})$, $pre^*_{up}(\mathcal{P}, \mathcal{C})$ and $pre^*_{low}(\mathcal{P}, \mathcal{C})$ in a similar fashion.

A *finite run* r of $\mathcal{P}$ from a configuration $c \in Conf_{\mathcal{P}}$ is a finite sequence of configurations $(c_i)_{i=0,\ldots,n}$ such that $c_0 = c$ and $c_0 \xrightarrow{t_1} c_1 \xrightarrow{t_2} c_2 \ldots \xrightarrow{t_n} c_n$, where $t = (t_i)_{i=1,\ldots,n}$ is a sequence of transitions in Δ^*, also called the *trace* of r. We then write $c_0 \overset{t}{\Rightarrow}_{\mathcal{P}} c_n$, or $c_0 \rightarrow^n_{\mathcal{P}} c_n$ (c_n is reachable from c_0 in n steps).

We say that r is a run of $\mathcal{P}$ from a set of configurations $\mathcal{C}$ if and only if $\exists c \in \mathcal{C}$ such that r is a run of $\mathcal{P}$ from c. Let $Runs(\mathcal{P}, \mathcal{C})$ (resp. $Traces(\mathcal{P}, \mathcal{C})$) be the set of all finite runs (resp. traces) of $\mathcal{P}$ from a set of configurations $\mathcal{C}$.

These definitions are related to similar concepts on unlabelled PDSs detailed in Chapter 2. A UPDS and a PDS indeed share the same definition, but the semantics of the former expand the latter's. For a set $\mathcal{C} \subseteq P \times \Gamma^*$ of lower stack configurations (the upper stack is ignored) and a UPDS $\mathcal{P}$, let $post^*_{_{PDS}}(\mathcal{P}, \mathcal{C})$ and $pre^*_{_{PDS}}(\mathcal{P}, \mathcal{C})$ be the set of forward and backward reachable configurations from $\mathcal{C}$ using the PDS semantics. The following lemmas hold:

Lemma 4. *Given a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a set of configurations $\mathcal{C}$, t is a trace from $\mathcal{C}$ with respect to the UPDS semantics if and only if t is a trace from $\mathcal{C}_{low}$ with respect to the standard PDS semantics.*

Lemma 5. *Given a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a set of configurations $\mathcal{C}$, $post^*_{low}(\mathcal{P}, \mathcal{C}) = post^*_{PDS}(\mathcal{P}, \mathcal{C}_{low})$ and $pre^*_{low}(\mathcal{P}, \mathcal{C}) = pre^*_{PDS}(\mathcal{P}, \mathcal{C}_{low})$.*

Lemmas 4 and 5 are true because, if we ignore the upper stack, a PDS and a UPDS share the same semantics.

4.2 Reachability properties

As shown in Chapter 2, we know that pre^*_{PDS} and $post^*_{PDS}$ are regular for a regular set of starting configurations. We prove that these results cannot be extended to UPDSs, but that $post^*$ is still context-sensitive. This implies that reachability of a single configuration is decidable for UPDSs.

4.2.1 $post^*$ is not regular

The following counterexample proves that, unfortunately, $post^*(\mathcal{P}, \mathcal{C})$ is not always regular for a given regular set of configurations $\mathcal{C}$ and a UPDS $\mathcal{P}$. The intuition behind this statement is that the upper stack can be used to store symbols in a non-regular fashion. The counterexample should be carefully designed in order to prevent later push operations from overwriting these symbols.

Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a UPDS with $P = \{p, p'\}, \Gamma = \{a, b, x, y, \bot\}$, and Δ the following set of pushdown transitions:

$$\begin{array}{cccc} (S_x) & (p, x) \rightarrow (p, a) & (R_a) & (p, a) \rightarrow (p, \varepsilon) \\ (S_y) & (p, y) \rightarrow (p, b) & (R_b) & (p, b) \rightarrow (p, \varepsilon) \\ (C) & (p, a) \rightarrow (p, ab) & (E) & (p, \bot) \rightarrow (p', \bot) \end{array}$$

Let $\mathcal{C} = \{p\} \times \{\varepsilon\} \times (xy)^* x\bot$ be a regular set of configurations. We can compute a relevant subset L of $post^*(\mathcal{C})$:

Lemma 6. $L = \{\langle p', a^{n+1} b^n, \bot \rangle, n \in \mathbb{N}\} \subseteq post^*(\mathcal{C})$.

Proof. We prove that $\langle p, \varepsilon, x(yx)^n \bot \rangle \Rightarrow \langle p, a^{n+1} b^n, \bot \rangle$ by induction on n.

Basis: $\langle p, \varepsilon, x\bot \rangle \rightarrow \langle p, \varepsilon, a\bot \rangle \rightarrow \langle p, a, \bot \rangle$.

Induction step: if $\langle p, \varepsilon, (xy)^n x\bot\rangle \Rightarrow \langle p, a^{n+1}b^n, \bot\rangle$, since the only rule able to read or modify the symbol $\bot$ is (E) but it has not been applied as the PDS would end up in state p', we have $\langle p, \varepsilon, (xy)^n x\rangle \Rightarrow \langle p, a^{n+1}b^n, \varepsilon\rangle$, hence, $\langle p, \varepsilon, (xy)^{n+1}x\bot\rangle \Rightarrow \langle p, a^{n+1}b^n, yx\bot\rangle$.

However, $\langle p, a^{n+1}b^n, yx\bot\rangle \overset{S_y R_b S_x C^{n+1}}{\Rightarrow} \langle p, a^{n+1}, ab^{n+1}\bot\rangle$ and $\langle p, a^{n+1}, ab^{n+1}\bot\rangle \overset{R_a R_b^{n+1}}{\Rightarrow} \langle p, a^{n+2}b^{n+1}, \bot\rangle$. From there, we have $\langle p, a^{n+1}b^n, \bot\rangle \overset{E}{\Rightarrow} \langle p', a^{n+1}b^n, \bot\rangle$.

Hence, $\langle p', a^{n+1}b^n, \bot\rangle \in post^*(C), \forall n \in \mathbb{N}$. □

Then, we prove an inequality holding for any configuration in $post^*$:

Lemma 7. *$\forall\langle p, w_u, w_l\rangle \in post^*(\mathcal{C})$, let $w = \overline{w_u}w_l$; then $|w|_b + |w|_{\overline{b}} + 1 \geq |w|_a + |w|_{\overline{a}}$.*

Proof. The only rule in Δ that can add a letter a to the whole stack is S_x. However, in order to apply it more than once, a x deeper in the lower stack must be reached beforehand, and the only way to do so is by switching a y to a b and popping said b, hence, adding a b to the whole stack.

Moreover, the number of b in the whole stack keeps growing during a computation, since no rule can switch a b on the lower stack or delete it from the upper stack. The inequality therefore holds. □

If we suppose that $post^*(\mathcal{C})$ is regular, then so is the language $L^{p'}$, where $L^{p'} = \{\overline{w_u}w_l \mid \langle p', w_u, w_l\rangle \in post^*(\mathcal{C})\}$, and by the pumping lemma, it admits a pumping length k. We will apply the pumping lemma to an element of L in order to generate a configuration that should be in $post^*$ but does not comply with the previous inequality.

According to Lemma 6, $L \subseteq post^*(\mathcal{C})$ and as a consequence the word $w = \overline{a^{k+1}b^k}\bot$ is in $L^{p'}$. Hence, if we apply the pumping lemma to w, there exist $x, y, z \in (\Gamma \cup \overline{\Gamma})^*$ such that $w = xyz$, $|xy| \leq k$, $|y| \geq 1$, and $xy^iz \in post^*(\mathcal{C})$, $\forall i \geq 1$. As a consequence of w's definition, $x, y \in \overline{a}^*$ and $z \in (\overline{a} + \overline{b})^*$.

Hence, for i large enough, $w_i = xy^iz \in L^{p'}$ and $|w_i|_{\overline{a}} > |w_i|_{\overline{b}} + 1$. By Lemma 7, this cannot happen and therefore neither $L^{p'}$ nor $post^*(\mathcal{C})$ are regular.

It should be noted that $L_{up}^{p'}$ is not regular either. Indeed, from the definition of $\mathcal{P}$ and $\mathcal{C}$, it is clear that $\forall\langle p', w_u, w_l\rangle \in post^*(\mathcal{C}), w_l = \bot$, so $L_{up}^{p'}$ and $L^{p'}$ are in bijection. We have therefore proven the following theorem:

Theorem 11. *There exist a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$ for which neither $post^*(\mathcal{C})$ nor $post^*_{up}(\mathcal{C})$ are regular.*

4.2.2 *pre** is not regular

We now prove that pre^* is not regular either. Let $\mathcal{P} = (P, \Gamma, \Delta)$ be a UPDS with $P = \{p\}, \Gamma = \{a, b, c\}$, and Δ the following set of push-down transitions:

$$\begin{array}{llll} (C_0) & (p,c) \rightarrow (p,ab) & (R_a) & (p,a) \rightarrow (p,\varepsilon) \\ (C_1) & (p,c) \rightarrow (p,cb) & (R_b) & (p,b) \rightarrow (p,\varepsilon) \end{array}$$

We define the regular set of configurations $\mathcal{C} = \{p\} \times (ab)^* \times \{c\}$ and again, compute a relevant subset of $pre^*(\mathcal{C})$:

Lemma 8. $L = \{\langle p, b^n, c^n c\rangle, n \in \mathbb{N}\} \subseteq pre^*(\mathcal{C})$.

Proof. By induction on n, we can prove that $\langle p, b^n, c^n c\rangle \Rightarrow \langle p, (ab)^n, c\rangle$, proving the induction step by using the fact that $\langle p, b^{n+1}, c^{n+2}\rangle \Rightarrow \langle p, abb^n, c^n c\rangle$. □

Given the rules of $\mathcal{P}$, the following lemma is verified:

Lemma 9. *If $\langle p, b^m, c^n\rangle \Rightarrow^* \langle p, w_u, w_l\rangle$, then $|w_u|_a + |w_l|_a \leq n$.*

Proof. The only rule that can add an a to the whole stack is C_0 and it replaces a c on the lower stack by ab. Hence, during a computation, one cannot create more a than there were c in the initial configuration. The inequality therefore holds. □

If $pre^*(\mathcal{C})$ is regular, so is $L^p = \{\overline{w_u} w_l \mid \langle p, w_u, w_l\rangle \in pre^*(\mathcal{C})\}$, and by the pumping lemma, it admits a pumping length k. Moreover, by lemma 8, $w = \overline{b^k} c^k c \in L^p$.

If we apply the pumping lemma to w, there exist $x, y, z \in (\Gamma \cup \overline{\Gamma})^*$ such that $w = xyz$, $|xy| \leq k$, $|y| \geq 1$ and $w_i = xy^i z \in pre^*(\mathcal{C})$, $\forall i \geq 1$. As a consequence of w's definition, $x, y \in \overline{b}^*$ and $z \in \overline{b}^* c^k c$.

Since $w_i \in L^p$, $\forall i \geq 1$, there exists an integer n_i such that $w_i \Rightarrow c_i = \overline{(ab)^{n_i}} c$. Moreover, the size of the stack must grow or remain constant during a computation, hence $|c_i| \geq |w_i|$ and $n_i \geq \frac{|w_i|-1}{2}$. Since words in the sequence $(w_i)_i$ are unbounded in length, the sequence $(n_i)_i$ must be unbounded as well. However, by Lemma 9, $n_i = |c_i|_{\overline{a}} \leq |w_i|_c = k + 1$.

Hence, there is a contradiction and $pre^*(\mathcal{C})$ is not regular.

Theorem 12. *There exist a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$ for which $pre^*(\mathcal{C})$ is not regular.*

4.2.3 *post** is context-sensitive

We prove that, if $\mathcal{C}$ is a regular set of configurations of a UPDS $\mathcal{P}$, then $post^*(\mathcal{P},\mathcal{C})$ is context-sensitive. This implies that we can decide whether a single configuration is reachable from $\mathcal{C}$ or not.

We show that the problem of computing $post^*(\mathcal{P},\mathcal{C})$ can be reduced without loss of generality to the case where $\mathcal{C}$ contains a single configuration. To do so, we define a new UPDS $\mathcal{P}'$ by adding new states and rules to $\mathcal{P}$ in such manner that any configuration c in $\mathcal{C}$ can be reached from a single configuration $c_\$ = \langle p_\$, \varepsilon, \$\rangle$. Once a configuration in $\mathcal{C}$ is reached, $\mathcal{P}'$ follow the same behaviour as $\mathcal{P}$.

Theorem 13. *For each UPDS $\mathcal{P} = (P,\Gamma,\Delta)$ and each regular set of configurations $\mathcal{C}$ on $\mathcal{P}$, there exists a UPDS $\mathcal{P}' = (P',\Gamma\cup\overline{\Gamma}\cup\{\$\},\Delta')$, $P\subseteq P'$, and $p_\$ \in P'\setminus P$ such that $post^*(\mathcal{P},\mathcal{C}) = post^*(\mathcal{P}',\{\langle p_\$,\varepsilon,\$\rangle\})\cap(P\times\Gamma^*\times\Gamma^*)$.*

Proof. Our intuition is to build configurations in $\mathcal{C}$ in three steps: from $c_\$$, push the word $\overline{w_u}w_l$ on the stack by using push rules mimicking a finite automaton accepting the regular set $\mathcal{C}$, switch each symbol in $\overline{\Gamma}$ to its equivalent letter in Γ and then pop it in order to write w_u on the upper stack, then move to the right state p.

Since $\mathcal{C}$ is regular, so is $\forall p\in P$ the language $\{\overline{w_u}^R w_l \mid \langle p,w_u,w_l\rangle\in\mathcal{C}\}$. Consider $\mathcal{A}_p = (\Gamma\cup\overline{\Gamma},Q_p,E_p,I_p,F_p)$ such that $\mathcal{L}(\mathcal{A}_p) = \{(\overline{w_u}w_l)^R \mid \langle p,w_u,w_l\rangle\in\mathcal{C}\}$. The mirror image is needed because the bottom of lower stack should be pushed first and the top of upper stack last. Without loss of generality, we suppose that $I_p = \{i_p\}$, $F_p = \{f_p\}$, $p_\$ \notin Q_p$, $Q_p\cap P = \emptyset$ and that no edge in E_p ends in i_p nor starts in f_p.

We define the UPDS $\mathcal{P}'_p = (P'_p,\Gamma\cup\overline{\Gamma}\cup\{\$\},\Delta'_p)$, where $P'_p = Q_p\cup\{p_\$,p,p_\tau\}$, $p_\tau\notin Q_p$ and the following rules belong to Δ'_p:

Rules from $\mathcal{A}_p$: for all $x\in\Gamma\cup\overline{\Gamma}$, if $q \xrightarrow[E_p]{x}{}^{*} q'$ in the automaton $\mathcal{A}_p$, then $(p_\$,\$)\to(q,x)\in\Delta'_p$ if $q=i_p$, and $(q,y)\to(q',xy)\in\Delta'_p$ $y\in\Gamma\cup\overline{\Gamma}$ otherwise. These rules are used to build the stack and mimicks transitions in $\mathcal{A}_p$; symbols that will end on the upper stack are stored on the lower stack.

Setting the upper stack: for all $\overline{x}\in\overline{\Gamma}$, $(f_p,\overline{x})\to(p_\tau,x)\in\Delta'_p$ and $(p_\tau,x)\to(f_p,\varepsilon)\in\Delta'_p$. Each symbol $\overline{x}$ on the top of the lower

stack is switched to its equivalent symbol in Γ then popped in order to end on the upper stack.

Moving to state p: for all $x \in \Gamma$, $(f_p, x) \to (p, x) \in \Delta'_p$. Once the upper stack has been defined and the lower stack is being read, the UPDS moves to state p in order to end in a configuration in $\mathcal{C}$.

The UPDS $\mathcal{P'}_p$ follows the three steps previously outlined: push $\overline{w_u} w_l$ on the lower stack, so that it is in a configuration $\langle f_p, \varepsilon, \overline{w_u} w_l \rangle$, then move to $\langle p_\tau, w_u, w_l \rangle$ by switching and popping the symbols in $\overline{\Gamma}$, and end in $\langle p, w_u, w_l \rangle$.

We then introduce $\mathcal{P}' = (\bigcup_{p \in P} P'_p \cup P, \Gamma \cup \overline{\Gamma} \cup \{\$\}, \bigcup_{p \in P} \Delta'_p \cup \Delta)$. From $c_\$$, $\mathcal{P}'$ can reach any configuration of $\mathcal{C}$ using the rules of the automata $(\mathcal{P'}_p)_{p \in P}$, then follow the rules of $\mathcal{P}$. $\mathcal{P}'$ therefore satisfies Theorem 13. □

Using this theorem, we can focus on the single starting configuration case. We assume for the rest of this subsection that $\mathcal{C} = \langle p_\$, \varepsilon, \$ \rangle$, $p_\$ \in P$, and $\$ \in \Gamma$. We now formally define context-sensitive grammars:

Definition 15. *A* grammar *$\mathcal{G}$ is a quadruplet (N, Σ, R, S) where N is a finite set of non terminal symbols, Σ a finite set of terminal symbols with $N \cap \Sigma = \emptyset$, $R \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ a finite set of production rules, and $S \in N$ a start symbol.*

We define the one-step derivation relation $\dashrightarrow_{\mathcal{G}}$ on a given grammar $\mathcal{G}$: if $\exists p, q \in (N \cup \Sigma)^*$, $p \to q \in R$ then for $x = upv$ and $y = uqv$, where $u, v \in (N \cup \Sigma)^*$, $x \dashrightarrow_{\mathcal{G}} y$. The derivation relation $\dashrightarrow^*_{\mathcal{G}}$ is its transitive closure. The language $\mathcal{L}(\mathcal{G})$ of a grammar is the set $\{w \in \Sigma^* \mid S \dashrightarrow^*_{\mathcal{G}} w\}$. We may omit the variable $\mathcal{G}$ when only a single grammar is being considered.

A grammar is said to be *context-sensitive* if each productions rule $r \in R$ is of the form $\alpha A \beta \to \alpha \gamma \beta$ where $\alpha, \beta \in (N \cup \Sigma)^*, \gamma \in (N \cup \Sigma)^+$, and $A \in N$. A language L is said to be context-sensitive if there exists a context-sensitive grammar $\mathcal{G}$ such that $\mathcal{L}(\mathcal{G}) = L$.

The following theorem is a well-know property of context-sensitive languages detailed in [HMRU00]:

Theorem 14. *Given a context-sensitive language L and a word $w \in \Sigma^*$, we can effectively decide whether $w \in L$ or not.*

We can compute a context-sensitive grammar recognizing *post**. Our intuition is to represent a configuration $\langle p, w_u, w_l \rangle$ of $\mathcal{P}$ by a word $\top w_u p w_l \bot$ of a grammar $\mathcal{G}$. We use Theorem 13 so that the single start

symbol of $\mathcal{G}$ can be matched to a single configuration $c_{\$}$. The context-sensitive rules of $\mathcal{G}$ mimic the transitions of the UPDS. As an example, a rule $\delta = (p, a) \rightarrow (p', \varepsilon) \in \Delta_{pop}$ can be modelled by three rules $pa \dashrightarrow_{\mathcal{G}} pg_\delta$, $pg_\delta \dashrightarrow_{\mathcal{G}} ag_\delta$, and $ag_\delta \dashrightarrow_{\mathcal{G}} ap'$ such that $pa \dashrightarrow^*_{\mathcal{G}} ap'$, where $\dashrightarrow_{\mathcal{G}}$ stands for the one-step derivation relation and g_δ is a non-terminal symbol of $\mathcal{G}$.

Let us define this context-sensitive grammar $\mathcal{G} = (N, \Sigma, R, S)$ more precisely:

Start symbol: S is the start symbol.

Nonterminal symbols: let $N = \{S\} \cup \overline{\Gamma} \cup \overline{P} \cup \Delta_{switch} \cup \Delta_{pop} \cup \Delta_{push} \times \{0, 1\}$. $\overline{P}$ is a disjoint copy (bijection) of the state alphabet P. In order to properly simulate transitions rules in Δ with context-sensitive production rules, non-terminal symbols related to these transitions are needed.

Terminal symbols: $\Sigma = \{\top, \bot\} \cup P \cup \Gamma$.

Production rules: $R = R_{\mathcal{P}} \cup R_{final} \cup \{S \rightarrow \top\overline{p_{\$}}\overline{\$}\bot\}$; the last rule initializes the starting configuration of $\mathcal{P}$.

The production rules in $R_{\mathcal{P}}$ simulate the semantics of the UPDS as defined by its transition rules Δ. For each switch rule $\delta : (p, a) \rightarrow (p', b) \in \Delta_{switch}$, the following grammar rules belong to $R_{\mathcal{P}}$ in order to allow $pa \dashrightarrow^*_{\mathcal{G}} p'b$:

$$(r_0^\delta) \quad \overline{pa} \rightarrow \delta\overline{a} \quad (r_1^\delta) \quad \delta\overline{a} \rightarrow \delta\overline{b} \quad (r_f^\delta) \quad \delta\overline{b} \rightarrow \overline{p'b}$$

For each pop rule $\delta : (p, a) \rightarrow (p', \varepsilon) \in \Delta_{pop}$, the following grammar rules belong to $R_{\mathcal{P}}$ in order to allow $pa \dashrightarrow^*_{\mathcal{G}} ap'$:

$$(r_0^\delta) \quad \overline{pa} \rightarrow \overline{p}\delta \quad (r_1^\delta) \quad \overline{p}\delta \rightarrow \overline{a}\delta \quad (r_f^\delta) \quad \overline{a}\delta \rightarrow \overline{a}\overline{p'}$$

For each push rule $\delta : (p, a) \rightarrow (p', bc) \in \Delta_{push}$, the following grammar rules belong to $R_{\mathcal{P}}$ in order to allow $xpa \dashrightarrow^*_{\mathcal{G}} p'bc$ and $\top pa \dashrightarrow^*_{\mathcal{G}} \top p'bc$:

$$\begin{array}{cccc} (r_0^\delta) & \overline{pa} \rightarrow \delta_0\overline{a} & \forall x \in \Gamma, (r_1^{\delta,x}) & \overline{x}\delta_0 \rightarrow \delta_1\delta_0 \\ (r_1^{\delta,\top}) & \top\delta_0 \rightarrow \top\delta_1\delta_0 & (r_2^\delta) & \delta_1\delta_0\overline{a} \rightarrow \delta_1\delta_0\overline{c} \\ (r_3^\delta) & \delta_1\delta_0\overline{c} \rightarrow \delta_1\overline{bc} & (r_f^\delta) & \delta_1\overline{bc} \rightarrow \overline{p'}\overline{bc} \end{array}$$

It is worth noting that, once a production rule r_0^δ has been applied, there is no other derivation possible in $R_{\mathcal{P}}$ but to apply the other production rules $(r_i^\delta)_i$ in the order they've been defined until a state symbol in P

has been written again by r_f^δ. This sequence simulates a single transition rule of the UPDS $\mathcal{P}$.

Finally, the rules in R_{final} merely switch symbols in $\overline{\Gamma} \cup \overline{P}$ to their equivalent letters in Σ in order to generate a terminal word, starting with the state symbol to prevent any further use of $R_{\mathcal{P}}$:

$$\begin{array}{lll} \forall p \in P & (r_p^{final}) & \overline{p} \to p \\ \forall x, y \in \Gamma \cup P & (r_{\overline{x},y}^{final}) & \overline{x}y \to xy \\ \forall x, y \in \Gamma \cup P & (r_{y,\overline{x}}^{final}) & y\overline{x} \to yx \end{array}$$

Once a rule r_p^{final} has been applied, the only production rules available for further derivations are in R_{final}.

We prove that $\mathcal{L}(\mathcal{G})$ is in bijection with $post^*(\{c_\$\})$.

Lemma 10. *If $\langle p, w_u, w_l \rangle \in post^*(\{c_\$\})$, then from S we can derive the nonterminal word $\top\overline{w_u p w_l}\bot$ in $\mathcal{G}$, and $\top w_u p w_l \bot \in \mathcal{L}(\mathcal{G})$.*

Proof. By induction on n, we must prove that if $c_\$ \Rightarrow^n \langle p, w_u, w_l \rangle$, then we can derive in $\mathcal{G}$ the non-terminal word $\top\overline{w_u p w_l}\bot$.

Basis: we have $S \to \top\overline{p_\$}\overline{\$}\bot$.

Induction step: if $c \Rightarrow^n \langle p, w_u, w_l \rangle \overset{\delta}{\Rightarrow} \langle p', w'_u, w'_l \rangle$, then the non terminal word $\top\overline{w_u p w_l}\bot$ can be derived from S in $\mathcal{G}$ by the induction hypothesis. From this word, we can further derive $\top\overline{w'_u p' w'_l}\bot$ using the production rules $r_0^\delta, \ldots, r_f^\delta$ associated with the transition rule δ.

Finally, from any non terminal word of the form $\top\overline{w_u p w_l}\bot$, we can derive in $\mathcal{G}$ a terminal $\top w_u p w_l \bot$ using rules in R_{final}. □

Moreover, by design of the grammar $\mathcal{G}$, the following lemma holds:

Lemma 11. *If $S \dashrightarrow^* \top w_u p w_l \bot$, $w_u, w_l \in \Gamma^*$, $p \in P$, then $\langle p, w_u, w_l \rangle \in post^*(\{c_\$\})$.*

Hence, the following result holds:

Theorem 15. *Given a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$, we can compute a context-sensitive grammar $\mathcal{G}$ such that $\langle p, w_u, w_l \rangle \in post^*(\mathcal{P}, \mathcal{C})$ if and only if $\top w_u p w_l \bot \in \mathcal{L}(\mathcal{G})$*

Since the membership problem is decidable for context-sensitive grammars, the following theorem holds:

Theorem 16. *Given a UPDS $\mathcal{P}$, a regular set of configurations $\mathcal{C}$, and a configuration c of $\mathcal{P}$, we can decide whether $c \in post^*(\mathcal{P}, \mathcal{C})$ or not.*

Unfortunately, this method cannot be extended to pre^* due to a property of context-sensitive grammars: each time a context-sensitive rule is applied to a non-terminal word to produce a new word, the latter is of greater or equal length than the former. The forward reachability relation does comply with this monotony condition, as the combined size of the upper and lower stacks can only increase or stay the same during a computation, but the backward reachability relation does not.

4.3 Under-approximating *pre**

Under-approximations of reachability sets can be used to discover errors in programs: if $\mathcal{X}$ is a regular set of forbidden configurations of a UPDS $\mathcal{P}$, $\mathcal{C}$ a regular set of starting configurations, and $\mathcal{U} \subseteq pre^*(\mathcal{X})$ a regular under-approximation, then $\mathcal{U} \cap \mathcal{C} \neq \emptyset$ implies that a forbidden configuration can be reached from the starting set, as shown in Figure 4.3. The emptiness of the above intersection has to be decidable, hence, the need for a regular approximation.

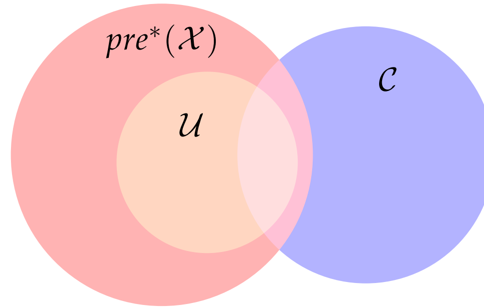

FIGURE 4.3: Using an under-approximation.

In Section 3.4, we used MPDSs with bounded phases in order to under-approximate the HyperLTL model-checking problem for PDSs. Here, we apply results on *multi-stack pushdown automata* to define an under-approximation of pre^* for UPDSs.

The notion of bounded-phase computations can be extended to UPDSs. A run r of $\mathcal{P}$ is said to be *k-phased* if it is of the form: $r = r_1 \cdot r_2 \ldots r_k$ where $\forall i \in \{1, \ldots, k\}, r_i \in (\Delta_{push} \cup \Delta_{switch})^* \cup (\Delta_{pop} \cup \Delta_{switch})^*$. During a phase, one can either push or pop, but can't do both. Such a run has therefore at most k alternations between push and pop rules. We can extend this notion to traces.

The *k-bounded* reachability relation $\Rightarrow_k$ is defined as follows: $c_0 \Rightarrow_k c_1$ if there exists a k-phased run r on $\mathcal{P}$ with a matching trace t such that $c_0 \overset{t}{\Rightarrow} c_1$. Using this new reachability relation, given a set of configurations $\mathcal{C}$, we can define $pre^*(\mathcal{P}, \mathcal{C}, k)$.

We can show that a UPDS $\mathcal{P}$ can be simulated by a MPDS $\mathcal{M}$ with two stacks, the second stack of $\mathcal{M}$ being equivalent to the lower stack, and the first one, to a mirrored upper stack followed by a symbol $\perp$ that can't be popped and is used to know when the end of the stack has been reached. Elements of $P \times \Gamma^* \times \Gamma^*$ can equally be considered as configurations of $\mathcal{P}$ or $\mathcal{M}$, assuming in the latter case that we consider the mirror of the first stack and add a $\perp$ symbol to its bottom. Thus:

Lemma 12. *For a given UPDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a regular set of configurations $\mathcal{C}$, there exists a MPDS $\mathcal{M}$, a regular set of configurations $\mathcal{C}'$, and $\perp \notin \Gamma$ such that $\langle p, w_u^R \perp, w_l \rangle \in pre^*_{MPDS}(\mathcal{M}, \mathcal{C}', k) \cap (P \times \Gamma^* \times \Gamma^*)$ if and only if $\langle p, w_u, w_l \rangle \in pre^*(\mathcal{P}, \mathcal{C}, k)$.*

Proof. Following the above intuition, we define a two-stack pushdown system $\mathcal{M} = (P \cup \Delta_{push} \cup \Delta_{pop}, \Gamma \cup \{\perp\}, 2, \Delta')$ where Δ' has the following rules:

Switch rules: if $\delta = (p, a) \rightarrow (q, b) \in \Delta_{switch}$, then $(p, a, 2) \rightarrow (q, b) \in \Delta'$.

Pop rules: if $\delta = (p, a) \rightarrow (q, \varepsilon) \in \Delta_{pop}$, then $(p, a, 2) \rightarrow (\delta, \varepsilon) \in \Delta'$ and $(\delta, x, 1) \rightarrow (q, ax) \in \Delta'$ for each $x \in \Gamma \cup \{\perp\}$.

Push rules: if $\delta = (p, a) \rightarrow (q, bc) \in \Delta_{push}$, then we define $(p, a, 2) \rightarrow (\delta, bc) \in \Delta'$, $(\delta, \perp, 1) \rightarrow (q, \perp) \in \Delta'$ and $(\delta, x, 1) \rightarrow (q, \varepsilon) \in \Delta'$ for each $x \in \Gamma$. If we reach $\perp$ on the second stack, the upper stack is considered to be empty and no symbol should be popped from it.

We then define $\mathcal{C}' = \{\langle p, w_u^R \perp, w_l \rangle \mid \langle p, w_u, w_l \rangle \in \mathcal{C}\}$. $\mathcal{M}$ simulates $\mathcal{P}$ and $\mathcal{C}'$ is equivalent to (in bijection with) $\mathcal{C}$. □

From Theorem 9, we get:

Theorem 17. *Given a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$, the set $pre^*(\mathcal{P}, \mathcal{C}, k)$ is regular and effectively computable.*

$pre^*(\mathcal{P}, \mathcal{C}, k)$ is then obviously an under-approximation of $pre^*(\mathcal{P}, \mathcal{C})$.

4.4 Over-approximating *post**

While under-approximations of reachability sets can be used to show that an error can occur, *over-approximations* can, on the other hand, prove that a program is safe from a particular error. If $\mathcal{X}$ is a regular set of forbidden configurations on a UPDS $\mathcal{P}$, $\mathcal{C}$ a regular set of starting configurations, and $O \supseteq post^*(\mathcal{C})$ a regular over-approximation, then $O \cap \mathcal{X} = \emptyset$ implies that no forbidden configuration can be reached

from the starting set and that the program is therefore safe, as shown in Figure 4.4.

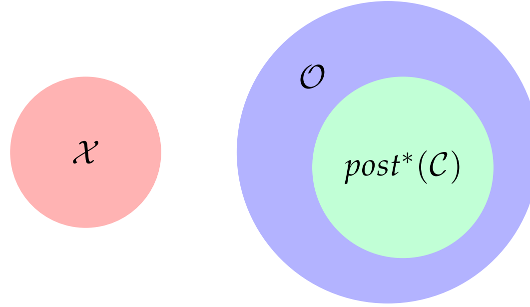

FIGURE 4.4: Using an over-approximation.

The emptiness of the above intersection has to be decidable, hence, the need for a regular approximation.

4.4.1 A relationship between runs and the upper stack

We prove here that from a regular set of traces of a given UPDS, a regular set of corresponding upper stacks can be computed. A subclass of programs whose UPDS model has a regular set of traces would be programs with finite recursion (hence, with a stack of finite height).

Given a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$ and a configuration $c = \langle p, w_u, w_l \rangle$ of $\mathcal{P}$, we match inductively to each sequence of transition $\tau \in \Delta^*$ an upper stack word $v(\tau, c) \in \Gamma^*$ according to the following rules:

— $v(\varepsilon, c) = w_u$;

— if $\delta = (p, \gamma) \rightarrow (p', \gamma') \in \Delta_{switch}$, then $v(\tau\delta, c) = v(\tau, c)$;

— if $\delta = (p, \gamma) \rightarrow (p', \varepsilon) \in \Delta_{pop}$, then $v(\tau\delta, c) = v(\tau) \cdot a$;

— if $\delta = (p, \gamma) \rightarrow (p', \gamma'\gamma'') \in \Delta_{push}$, then $v(\tau\delta, c) = \varepsilon$ if $v(\tau, c) = \varepsilon$, and $v(\tau\delta, c) = w$ if $v(\tau) = wx$, where $x \in \Gamma$;

Intuitively, $v(\tau, c)$ is the upper stack content after applying the sequence of transitions τ, starting from a configuration c. Note that τ may not be an actual trace of $\mathcal{P}$; $v(\tau)$ is merely the *virtual* upper stack built by pushing and popping values in a write-only manner, regardless of the lower stack, the control states, and the coherence of the sequence of transitions used. However, if t is indeed a trace of $\mathcal{P}$, then the upper stack configuration $v(t, c)$ is indeed reachable from c using the trace t.

A sequence of transitions is said to be *meaningful* if $\forall p' \in P$, any transition ending in state p' can only be followed by a transition starting in state p'. A trace of $\mathcal{P}$ is obviously a meaningful sequence. A set

of sequences of transitions T is said to be *prefix-closed* if, given $t \in T$, any prefix of t is in T as well. The set of all traces of a given system is obviously prefix-closed.

The following theorem holds:

Theorem 18. *For a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$, a regular set of configurations $\mathcal{C}$, and a regular, prefix-closed set of meaningful sequences of transitions $T \subseteq \Delta^*$ of $\mathcal{P}$ from $\mathcal{C}$, the set of upper stack configurations $\mathcal{U}(T) = \{\langle p', w'_u \rangle \mid \exists c = \langle p, w_u, w_l \rangle \in \mathcal{C}, \exists t \in T, t$ starts in state p and ends in state $p', v(t, c) = w'_u\}$ spawned by T from $\mathcal{C}$ is regular and effectively computable.*

Thanks to Theorem 13, we consider the single configuration case where $\mathcal{C} = \{c_\$\}$ without loss of generality. Let $\mathcal{A}_T = (\Delta, Q, E, I, F)$ be a finite state automaton such that $\mathcal{L}(\mathcal{A}_T) = T$. Since T is meaningful, we can assume that $Q = \bigcup_{p \in P} Q_p$ where Q_p is such that $\forall q \in Q_p$, if there is an edge $q' \xrightarrow{\delta}_E q$, then the pushdown rule δ is of the form $(p', a) \to (p, w)$. We can also assume that $F = Q$ since T is prefix-closed.

We introduce the automaton $\mathcal{A}_U = (\Gamma, Q, E', I, F)$ whose set of transitions E' is defined by applying the following rules until saturation, starting from $E' = \emptyset$:

(S_{pop}) if there is an edge $q_0 \xrightarrow{\delta}_E q_1$ in $\mathcal{A}_T$ and δ is of the form $(p, a) \to (p', \varepsilon)$, then we add the edge $q_0 \xrightarrow{a} q_1$ to E'.

(S_{switch}) if there is an edge $q_0 \xrightarrow{\delta}_E q_1$ in $\mathcal{A}_T$ and δ is of the form $(p, a) \to (p', b)$, then we add the edge $q_0 \xrightarrow{\varepsilon} q_1$ to E'.

(S_{push}) if there is an edge $q_0 \xrightarrow{\delta}_E q_1$ in $\mathcal{A}_T$, δ is of the form $(p, a) \to (p', bc)$, and there is a state q such that either **(1)** $q \in Q$ and $q \xrightarrow{x}_{E'} q_0$ for $x \in \Gamma$ or **(2)** $q \in I$ and $q \xrightarrow{\varepsilon}^*_{E'} q_0$, then we add an edge $q \xrightarrow{\varepsilon} q_1$ to E'.

We call $(E'_i)_i$ the finite, growing sequence of edges created during the saturation procedure.

Our intuition behind the above construction is to create a new automaton that uses the states of the sequence automaton but accepts upper stack words instead: an upper stack word w is accepted by $\mathcal{A}_U$ with the path $q_i \xrightarrow{w}^*_{E'} q$, where $q_i \in I$ if and only if $\mathcal{A}_T$ accepts a sequence t with the path $q_i \xrightarrow{t}^*_E q$ where t ends in state p and $v(t, c_\$) = w$. This property is preserved at every step of the saturation procedure.

Indeed, consider a sequence t and $w = v(t, c_\$)$. Suppose that t and w satisfy the property above: there is a path $q_i \xrightarrow{t}^*_E q_0$ in $\mathcal{A}_T$ and a path

$q_i \xrightarrow{w}{}^*_{E'} q_0$ in $\mathcal{A}_U$. Let $q_0 \xrightarrow{\delta}_E q_1$ be a transition of $\mathcal{A}_T$, $q_1 \in Q$ and $\delta \in \Delta$. $t\delta$ is a sequence of transitions in T with a labelled path $q_i \xrightarrow{t\delta}{}^*_E q_1$ in $\mathcal{A}_T$, and in order to satisfy the above property, a path $q_i \xrightarrow{w'}{}^*_{E'} q_1$ labelled by $w' = v(t\delta, c_\$)$ should exist in $\mathcal{A}_U$ as well.

If $\delta \in \Delta_{pop}$, then $w' = v(t\delta, c_\$) = wa$, where $a \in \Gamma$. Rule (S_{pop}) creates an edge $q_0 \xrightarrow{a} q_1$ in $\mathcal{A}_U$ such that there is a path $q_i \xrightarrow{w}{}^*_{E'} q_0 \xrightarrow{a}_{E'} q_1$ labelled by w'.

If $\delta \in \Delta_{switch}$, then $w' = v(t\delta, c_\$) = w$. Rule (S_{switch}) creates an edge $q_0 \xrightarrow{\varepsilon} q_1$ in $\mathcal{A}_U$ such that there is a path $q_i \xrightarrow{w}{}^*_{E'} q_0 \xrightarrow{\varepsilon}_{E'} q_1$ labelled by w'.

If $\delta \in \Delta_{push}$ and $w = w_0 x$, where $x \in \Gamma$, then $w' = v(t\delta, c_\$) = w_0$ and for every state $q \in Q$ such that $q_i \xrightarrow{w_0}{}^*_{E'} q \xrightarrow{x}_{E'} q_0$, (S_{push}) adds an edge $q \xrightarrow{\varepsilon} q_1$ to $\mathcal{A}_U$ such that there is a path $q_i \xrightarrow{w_0}{}^*_{E'} q \xrightarrow{\varepsilon}_{E'} q_1$ labelled by w'.

Following this intuition, we can prove this lemma:

Lemma 13. *For every sequence t such that $\exists q_i \in I$, $\exists q \in Q_p$, $q_i \xrightarrow{t}{}^*_E q$, then there exists a path $q_i \xrightarrow{w}{}^*_{E'} q$ in $\mathcal{A}_U$ such that $v(t, c_\$) = w$.*

On the other hand, we must prove this lemma to get the full equivalence:

Lemma 14. *At any step i of the saturation procedure, if $q_i \xrightarrow{w}{}^*_{E'_i} q$ where $q_i \in I$, then there exists a sequence of transitions t in T such that $q_i \xrightarrow{t}{}^*_E q$ and $v(t, c_\$) = w$.*

Proof. We prove this lemma by induction on the saturation step i:

Basis: $E'_0 = \emptyset$ and the lemma holds.

Induction step: Let $q_1 \xrightarrow{x}{}^*_{E'_{i+1}} q_2$ be the $i+1$-th transition added to E'.

Let $w' = wx$ be such that there is a path $q_i \xrightarrow{w}{}^*_{E'_i} q_1 \xrightarrow{x}_{E'_{i+1}} q_2$. By induction hypothesis, there is a sequence $t \in T$ such that $q_i \xrightarrow{t}{}^*_E q_1$ and $v(t, c_\$) = w$.

If $x \in \Gamma$, then there is a rule $\delta \in \Delta_{pop}$ popping x from the stack such that $q_1 \xrightarrow{\delta}_E q_2$ by definition of the saturation rules. We have $v(t\delta, c_\$) = wx = w'$ and the lemma holds at the $i+1$-th step.

If $x = \varepsilon$, then the rule δ spawning $q_1 \xrightarrow{x}{}^*_{E'_{i+1}} q_2$ is either a switch or push rule. The switch case being similar to the pop case, we will consider that $\delta \in \Delta_{push}$.

In the first case of the push saturation rule, there exists by definition a state q_0 such that $q_0 \xrightarrow{\delta}_E q_2$ and $y \in \Gamma$ such that $q_1 \xrightarrow{y}_{E'_i} q_0$. Hence, there is $\delta' \in \Delta_{pop}$ popping y from the stack such that $q_1 \xrightarrow{\delta'}{}^*_E q_0$. If we consider the sequence $q_i \xrightarrow{t}{}^*_E q_1 \xrightarrow{\delta'}_E q_0 \xrightarrow{\delta}_E q_2$, then $v(t\delta'\delta, c_\$) = w = w'$ and the lemma holds at the $i+1$-th step. The second case of the push saturation rule is similar.

□

Let $\mathcal{L}_p(\mathcal{A}_U) = \{w \mid \exists q_i \in I, \exists f \in Q_p, i \xrightarrow{w}{}^*_{E'} f\}$ be the set of paths in $\mathcal{A}_U$ ending in a final node related to a state p of $\mathcal{P}$. By Lemmas 13 and 14, $\mathcal{U}(R) = \{\langle p, w_u \rangle \mid w_u \in \mathcal{L}_p(\mathcal{A}_U)\}$. Since the languages $\mathcal{L}_p$ are regular and there is a finite number of them, $\mathcal{U}(T)$ is regular as well and can be computed using $\mathcal{A}_U$.

4.4.2 Computing an over-approximation

The set of traces of a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$ from a regular set of configurations $\mathcal{C}$ is not always regular. By Lemma 4, traces of $\mathcal{P}$ are the same for the UPDS and PDS semantics. Thus, we can apply methods originally designed for PDSs to over-approximate traces of a UPDS in a regular fashion, as shown in Chapter 3.

With one of these methods, we can therefore compute a regular over-approximation $\mathcal{T}(\mathcal{P}, \mathcal{C})$ of the set of traces of $\mathcal{P}$ from $\mathcal{C}$. Using the saturation procedure underlying Theorem 18, we can then compute the set $\mathcal{U}(\mathcal{T}(\mathcal{P}, \mathcal{C}))$ of upper stack configurations reachable using over-approximated traces of $\mathcal{P}$, hence, an over-approximation of the actual set of reachable upper stack configurations.

However, we still lack the lower stack component of the reachability set. As shown in Chapter 2, $post^*_{\text{PDS}}(\mathcal{P}, \mathcal{C})$ is regular and computable, and we can determine the exact set of reachable lower stack configurations.

We define the set $O = \{\langle p, w_u, w_t \rangle \mid \langle p, w_u \rangle \in \mathcal{T}(\mathcal{R}(\mathcal{P}, \mathcal{C})), \langle p, w_t \rangle \in post^*_{\text{PDS}}(\mathcal{P}, \mathcal{C})\}$. O is a regular over-approximation of $post^*(\mathcal{P}, \mathcal{C})$.

4.5 Applications

The UPDS model can be used to detect stack behaviours that cannot be found using a simple pushdown system. In this section, we present three such examples.

4.5.1 Stack overflow detection

A stack overflow is a programming malfunction occurring when the call stack pointer exceeds the stack bound. In order to analyze a program's vulnerability to stack overflow errors, we compute its representation as a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$, using the control flow model outlined in Chapter 2.

Let $\mathcal{C} = P \times \top\#^m \times L$ be the set of starting configurations, where $\top \in \Gamma$ is a top stack symbol that does not appear in any rule in Δ, $\# \in \Gamma$ a filler symbol, m an integer depending on the maximal size of the stack, and L a regular language of lower stack initial words. Overwriting the top symbol would represent a stack overflow misfunction. Since there is no such thing as an upper stack in a simple pushdown automaton, we need a UPDS to detect this error, as shown in Figure 4.5.

FIGURE 4.5: Using $\top$ to bound the upper stack.

Let $\mathcal{X} = P \times (\Gamma \backslash \{\top\})^*\Gamma^* \times \Gamma^*$ be the set of forbidden configurations where the top stack symbol has been overwritten. If the intersection of the under-approximation $\mathcal{U}$ of $pre^*(\mathcal{X})$ with $\mathcal{C}$ is not empty, then a stack overflow does happen in the program. On the other hand, if the intersection of the over-approximation $\mathcal{O}$ of $post^*(\mathcal{C})$ with the set $\mathcal{X}$ of forbidden configurations is empty, then we are sure that a stack overflow will not happen in the program

4.5.2 Reading the upper stack

Let us consider the piece of code 4.1. In line 1, the bottom symbol of the upper stack sp − 4, just above the stack pointer, is copied into the register eax. In line 2, the content of eax is compared to a given value a. In line 3, if the two values are not equal, the program jumps to an error state err.

LISTING 4.1: Reading the upper stack

```
1 mov eax, [sp - 8]
2 cmp eax, a
3 je err
```

Using a simple PDS model, it is not possible to know what is being read. However, our UPDS model and the previous algorithms provide us with reasonable approximations which can be used to examine possible values stored in eax, as shown in Figure 4.6.

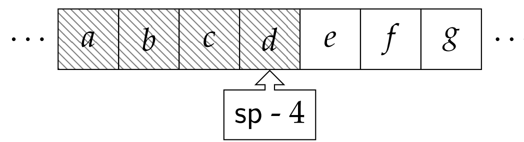

FIGURE 4.6: The stack being read.

To check whether this program reaches the error state *err* or not, we define the regular set $\mathcal{X} = P \times \Gamma^* a \times \Gamma^*$ of forbidden configurations where a is present on the upper stack just above the stack pointer. If the intersection of the under-approximation of $pre^*(\mathcal{X})$ with the set of starting configurations $\mathcal{C}$ of the program is not empty, then eax can contain a critical value, and the program is unsafe. On the other hand, if the intersection of the over-approximation of $post^*(\mathcal{C})$ with the set $\mathcal{X}$ is empty, then the program can be considered safe.

4.5.3 Changing the stack pointer

Another malicious use of the stack pointer sp would be to change the starting point of the stack. As an example, the instruction mov sp, sp - 12 changes the stack pointer in such a manner that, from the configuration of Figure 4.7, the top three elements above it now belong to the stack, as shown in Figure 4.8.

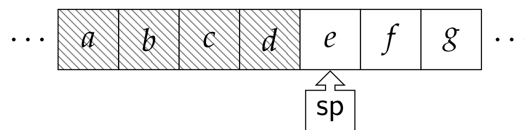

FIGURE 4.7: The original stack.

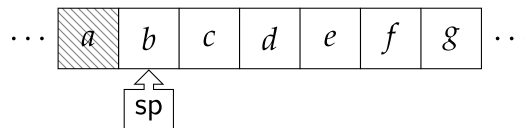

FIGURE 4.8: After changing sp.

If we model a program as a UPDS, then using our previous algorithms to compute approximations of the reachability set would allow us to have an approximation of the content of the new stack after the stack pointer change.

4.6 Related work

One way to improve the expressiveness of pushdown automata is to change the way transition rules interact with the stack. Ginsburg et al. introduced in [GGH67] *stack automata* that can read the inside of their own stack using a moving stack pointer but can only modify the top. As shown in [HU68], stack automata are equivalent to linear bounded automata (LBA). A LBA is a non-deterministic Turing machine whose tape is bounded between two end markers that cannot be overwritten. This model cannot simulate a UPDS whose lower stack is of unbounded height.

Uezato et al. defined in [UM13] *pushdown systems with transductions*: in such a model, a finite transducer is applied to the whole stack after each transition. However, this model is Turing powerful unless the transducers used have a finite closure, in which case it is equivalent to a simple pushdown system. When the set of transducers has a finite closure, this class cannot be used to simulate UPDSs.

Multi-stack automata have two or more stacks that can be read and modified, but are unfortunately Turing powerful. Following the work of Qadeer et al. in [QR05], La Torre et al. introduced in [TMP07] *multi-stack pushdown systems with bounded phases*: in each phase of a run, there is at most one stack that is popped from. Anil Seth later proved in [Set10] that the pre^* of a regular set of configurations of a multi-pushdown system with bounded phases is regular; we use this result to perform a bounded-phase analysis of our model.

2-visibly pushdown automata (2-VPDA) were defined by Carotenuto et all. in [CMP07] as a variant of two-stack automata where the stack operations are driven by the input word. However, an ordering constraint on the stacks that prevent a 2-VPDA from simulating a UPDS has to be introduced for this class of automata in order to solve the emptiness problem or the model-checking problem.

4.7 Conclusion

The first contribution of this chapter is a more precise pushdown model of the stack of a program as defined in Section 1. We then investigate the sets of predecessors and successors of a regular set of configurations of an UPDS. Unfortunately, we prove that neither of them are regular. However, we show that the set of successors is context-sensitive. As a consequence, we can decide whether a single configuration is forward reachable or not in an UPDS.

We then prove that the set of predecessors of an UPDS is regular given a limit of k phases, where a phase is a part of a run during which either pop or push rules are forbidden. Bounded-phase reachability is an under-approximation of the actual reachability relation on UPDSs that we can use to detect some incorrect behaviours.

We also give an algorithm to compute an over-approximation of the set of successors. Our idea is to first over-approximate the runs of the UPDS, then compute an over-approximation of the reachable upper stack configurations from this abstraction of runs and consider its product with the regular, accurate and computable set of lower stack configurations.

Finally, we use these approximations on programs to detect stack overflow errors as well as malicious attacks that rely on stack pointer manipulations.

Chapter 5

Static Analysis of Multi-threaded Recursive Programs Communicating via Rendez-vous

We present in this chapter a generic framework for the analysis of multi-threaded programs with recursive procedure calls, synchronization by rendez-vous between parallel threads, and dynamic creation of new threads. To this end, we consider a model called *synchronized dynamic pushdown networks* (SDPNs) that can be seen as a network of pushdown processes executing synchronized transitions, spawning new pushdown processes, and performing internal pushdown actions. The reachability problem for this model is obviously undecidable, as proven by Ramalingam in [Ram00].

However, we can tackle this problem by introducing an abstraction framework based on Kleene algebras in order to compute an abstraction of the set of execution paths between two regular sets of configurations. We combine constraint solving in a finite domain with an automata-theoretic saturation procedure. We then apply this framework to a iterative abstraction refinement scheme, using multiple abstractions of increasing complexity and precision.

Chapter outline. In Section 1 of this chapter, we define *synchronized dynamic pushdown networks* (SDPNs). We study in Section 2 the reachability problem for this class of automata. We introduce in Section 3 an automata-theoretic representation of sets of paths, and prove in Section 4 that the set of execution paths between two sets of configurations C and C' of a SDPN is the least solution of a set constraints. Since we can't solve these constraints, we present in Section 5 an abstraction framework for paths based on Kleene algebras. In Section 6, we apply this framework in order to over-approximate the reachability problem.

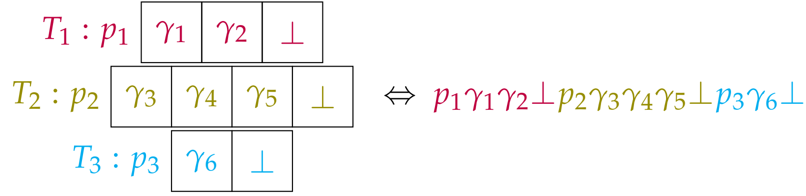

FIGURE 5.1: Representing configurations of a DPN.

In Section 7, we present a *iterative abstraction refinement* scheme that relies on our abstraction framework and apply it to a model of an actual program in section 8. Finally, we describe the related work in Section 9 and present our conclusion in Section 10.

These results were published in [PT17].

5.1 Synchronized dynamic pushdown networks

5.1.1 Dynamic pushdown networks

Definition 16 (Bouajjani et al. [BMOT05]). *A* dynamic pushdown network *(DPN) is a triplet $M = (P, \Gamma, \Delta)$ where P is a finite set of control states, Γ a finite stack alphabet disjoint from P, and $\Delta \subseteq (P\Gamma \times P\Gamma^*) \cup (P\Gamma \times P\Gamma^* P\Gamma^*)$ a finite set of transition rules featuring:*

- *simple pushdown operations in $(P\Gamma \times P\Gamma^*)$ of the form $p\gamma \to p'w$;*
- *thread spawns in $(P\Gamma \times P\Gamma^* P\Gamma^*)$ of the form $p\gamma \to p_1 w_1 p_2 w_2$.*

Let $Conf_M = (P\Gamma^*)^*$ be the set of configurations of a DPN M. A configuration $p_1 w_1 \ldots p_n w_n$ represents a network of n pushdown processes, the i-th process being in control point p_i with stack content w_i, as shown in Figure 5.1 where a single word in $Conf_M$ is used to represent the state of three PDSs in a network.

We define an immediate successor relation $\to_M$ on $Conf_M$ according to the following semantics:

- if $p\gamma \to p'w$ in Δ, then $\forall u, v \in Conf_M$, $\forall w' \in \Gamma^*$, $up\gamma w'v \to_M up'ww'v$; a thread applies a pushdown operation on its own stack, as shown in Figure 5.2;
- if $p\gamma \to p_1 w_1 p_2 w_2$ in Δ, then $\forall u, v \in Conf_M$, $\forall w' \in \Gamma^*$, $up\gamma w'v \to_M up_1 w_1 p_2 w_2 w'v$; a thread spawns a new son with its own stack and control state, as shown in Figure 5.3.

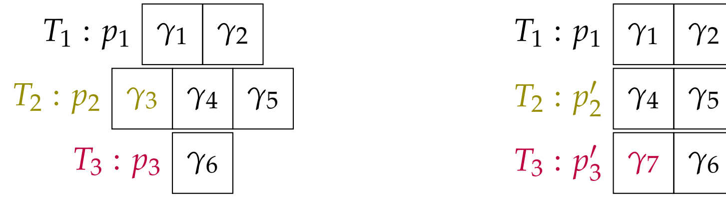

FIGURE 5.2: A DPN with 3 threads after a pop from T_2 and a push on T_3.

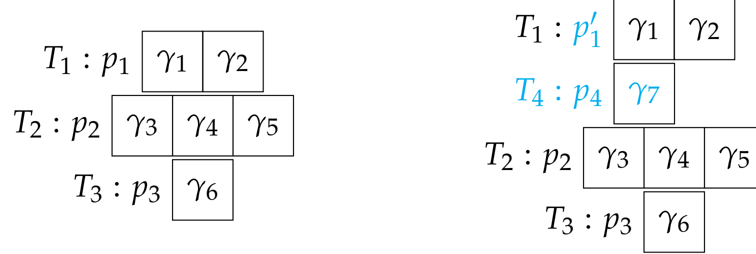

FIGURE 5.3: A DPN with 3 threads after thread T_1 spawns a new thread T_4.

Let $\rightarrow_M^*$ be the transitive and reflexive closure of this relation. Given a set $\mathcal{C} \subseteq Conf_M$ of configurations, we introduce its set of *predecessors* $pre^*(M, \mathcal{C}) = \{c \in Conf_M \mid \exists c' \in \mathcal{C}, c \Rightarrow_M c'\}$. If $\mathcal{C}$ is regular, this set can be effectively computed:

Theorem 19 (Bouajjani et al. [BMOT05]). *Given a DPN M and a regular set of configurations $\mathcal{C} \subseteq Conf_M$, the set $pre^*(M, \mathcal{C})$ of predecessors is regular.*

The saturation algorithm used to compute $pre^*(M, \mathcal{C})$ is detailled in Section 5.4.1.

5.1.2 The model and its semantics

We introduce a new model:

Definition 17. *A* synchronized dynamic pushdown Network *(SDPN) is a quadruplet $M = (Act, P, \Gamma, \Delta)$ where Act is a finite set of actions, P a finite set of control states, Γ a finite stack alphabet disjoint from P, and $\Delta \subseteq (P\Gamma \times Act \times P\Gamma^*) \cup (P\Gamma \times Act \times P\Gamma^* P\Gamma^*)$ a finite set of transition rules.*

If $(p\gamma, a, w) \in \Delta$, $p \in P$, $\gamma \in \Gamma$, $a \in Act$, and $w \in P\Gamma^* \cup P\Gamma^* P\Gamma^*$, we write that $p\gamma \xrightarrow{a} w \in \Delta$. There are two types of transition rules in a SDPN:

— rules of the form $p\gamma \xrightarrow{a} p'w$ in $P\Gamma \times Act \times P$ allow a pushdown process in the network to pop a symbol γ from its stack, push

$$\begin{array}{ccccc} u_1 & p\gamma w & u_2 & p'\gamma' w' & u_3 \\ & \downarrow a & & \downarrow \overline{a} & \\ u_1 & w_1 w & u_2 & w_1' w' & u_3 \end{array}$$

FIGURE 5.4: Semantics of synchronized actions.

a word w, then move from state p to p'; these rules are standard pushdown rules and model a thread calling or ending procedures while moving through its control flow;

— rules of the form $p\gamma \xrightarrow{a} p''w'p'w$ in $P\Gamma \times Act \times P\Gamma^* P\Gamma^*$ allow a pushdown process in the network to pop a symbol γ from its stack, push a word w, move from state p to p', then spawn a new pushdown process in state p'' and with initial stack w'; these rules model dynamic creation of new threads.

We assume that the set Act contains a letter τ that represents internal or synchronized actions, and that other letters in $Lab = Act \setminus \{\tau\}$ model synchronization signals. Moreover, to each synchronization signal a in Lab, we can match an unique co-action $\overline{a} \in Lab$, such that $\overline{\overline{a}} = a$.

We introduce the set $Conf_M = (P\Gamma^*)^*$ of configurations of a SDPN M. In a manner similar to DPNs, a configuration $p_1 w_1 \ldots p_n w_n$ represents a network of n processes where the i-th process is in control point p_i and has stack content w_i.

The strict semantics.

We will model synchronization between threads as a form of *communication by rendez-vous*: two pushdown processes can synchronize if one performs a transition labelled with a and the other, a transition labelled with $\overline{a}$. Intuitively, one thread sends a signal over a channel and the other thread waits for a signal to be received along the same channel.

To this end, we define a strict transition relation $\dashrightarrow_M$ on configurations of M according to the following *strict semantics*:

(1) given a symbol $a \in Act$, two rules $p\gamma \xrightarrow{a} w_1$ and $p'\gamma' \xrightarrow{\overline{a}} w_1'$ in Δ, and two configurations $u = u_1 p\gamma u_2 p'\gamma' u_3$ and $v = u_1 w_1 u_2 w_1' u_3$ of M, we have $u \dashrightarrow_M v$; two synchronized processes perform a simultaneous action, as shown in Figure 5.4;

(2) given a rule $p\gamma \xrightarrow{\tau} w_1$ in Δ and two configurations $u = u_1 p\gamma u_2$ and $v = u_1 w_1 u_2$ of M, we have $u \dashrightarrow_M v$; a process performs an internal action, as shown in Figure 5.5.

$$\begin{array}{ccc} u_1 & p\gamma w & u_2 \\ & \downarrow \tau & \\ u_1 & w_1 w & u_2 \end{array}$$

FIGURE 5.5: Semantics of internal actions.

$$\begin{array}{ccc} u_1 & p\gamma w & u_2 \\ & \downarrow a & \\ u_1 & w_1 w & u_2 \end{array}$$

FIGURE 5.6: Semantics of unsynchronized actions.

We say that v is reachable from u with regards to the strict semantics if $u \dashrightarrow_M^* v$, where $\dashrightarrow_M^*$ stands for the transitive closure of $\dashrightarrow_M$.

The strict semantics accurately model communication by rendez-vous. However, for technical reasons, we also need to consider a relaxed semantics for SDPNs.

The relaxed semantics.

The *relaxed semantics* on SDPNs allow partially synchronized executions on a SDPN: a process can perform a transition labelled with $a \in Lab$ even if doesn't synchronize with a matching process executing a transition labelled with $\overline{a}$.

We therefore introduce a relaxed transition relation $\rightarrow_M$ labelled in Act on configurations of M:

(1) & (2) given two configurations u and v of M, if $u \dashrightarrow_M v$, then $u \xrightarrow{\tau}_M v$; $\rightarrow_M$ features rules **(1)** and **(2)** of $\dashrightarrow_M$;

(3) given a rule $p\gamma \xrightarrow{a} w_1$ in Δ, a word $w_1 \in (P\Gamma^*) \cup (P\Gamma^*)^2$, and two configurations $u = u_1 p\gamma u_2$ and $v = u_1 w_1 u_2$ of M, we have $u \xrightarrow{a}_M v$; a process performs an action but does not synchronize, as shown in Figure 5.6.

The restriction of the relaxed semantics to rules **(2)** and **(3)** yields the DPN semantics if we ignore the labels, as defined by Bouajjani et al. in [BMOT05].

For a given word $\sigma = a_1 \dots a_n \in Act^*$ and two configurations c, c' of M, we write that $c \xrightarrow{\sigma}_M^* c'$ if there are n configurations $c_1, \dots, c_n$ of M such that $c \xrightarrow{a_1}_M c_1 \xrightarrow{a_2}_M c_2 \dots \xrightarrow{a_n}_M c_n$ and $c_n = c'$. We then say that c' is reachable from c with regards to the relaxed semantics. For a given set of configurations C, we introduce $pre^*(M, C) = \{c' \mid \exists c \in C, \exists w \in \Gamma^*, c' \xrightarrow{w}_M^* c\}$.

Given two subsets C and C' of $Conf_M$, we define the set of all execution paths from C to C' $Paths_M(C, C') = \{\sigma \in Act^* \mid \exists c \in C, \exists c' \in C', c \xrightarrow{\sigma}{}^*_M c'\}$, including paths with non-synchronized actions labelled in Lab.

5.1.3 From a program to a SDPN model

We can assume that the program is given by a *control flow graph*, whose nodes represent control points of threads or procedures and whose edges are labelled by statements. These statements can be variable assignments, procedure calls or returns, spawns of new threads, or communications between threads through unidirectional point-to point channels, where a thread sends a value x through a channel c and another thread waits for this value then assigns it to a variable y.

Without loss of generality, we assume that threads share no global variables and instead can only synchronize through channels. We distinguish local variables that belong to a single procedure from thread-local variables that can be accessed by any procedure called by a given instance of a thread. We also consider that both local and global variables may only take a finite number of values.

Given a control flow graph, we define a corresponding SDPN. The set of states P is the set of all possible valuations of thread-local variables. The stack alphabet Γ is the set of all pairs (n, l) where n is a node of the flow graph and l is a valuation of the local variables of the current procedure.

Channels can be used to send and receive values. For each channel c and value x that can be sent through c, a label $(c!, x)$ and its co-action $(c?, x) = \overline{(c!, x)}$ belong to Act. The internal action τ belongs to Act as well.

For each statement s labelling an edge of the flow graph between nodes n_1 and n_2, we introduce the following transition rules in the corresponding SDPN, where g_1 and g_2 (resp. l_1 and l_2) are the valuations of thread-local (resp. procedure-local) variables before and after the execution of the statement:

— if s is an assignment, rules of the form $g_1(n_1, l_1) \xrightarrow{\tau} g_2(n_2, l_2)$ represent s; assigning new values to variables in g_1 and l_1 results in new valuations g_2 and l_2;

— if s is a procedure call, rules of the form $g_1(n_1, l_1) \xrightarrow{\tau} g_2(f_0, l_0)$ (n_2, l_2) represent s, where f_0 is the starting node of the called procedure and l_0 the initial valuation of its local variables;

— if s is a procedure return, it is represented by rules of the form $g_1(n_1, l_1) \xrightarrow{\tau} g_2\varepsilon$; we simulate returns of values by introducing an additional thread-local variable and assigning the return value to it in the valuation g_2;

— if s is a thread spawn, it is represented by rules of the form $g_1(n_1, l_1) \xrightarrow{\tau} g_0(n_0, l_0)g_2(n_2, l_2)$, where g_0 and l_0 are respectively the initial valuations of the thread-local and procedure-local variables of the new thread, and n_0 its starting node;

— if s is an assignment of a value x carried through a channel c to a variable y, it is represented by rules of the form $g_1(n_1, l_1) \xrightarrow{(c?,x)} g_2(n_2, l_2)$ where g_1 and g_2 (resp. l_1 and l_2) are such that assigning the value x to the variable y in g_1 (resp. l_1) results in the new valuations g_2 (resp. l_2);

— if s is an output through a channel c of the value x of a variable y, it is represented by rules of the form $g_1(n_1, l_1) \xrightarrow{(c!,x)} g_2(n_2, l_2)$ such that the variable y has value x in either g_1 or l_1.

Finally, we consider the starting configuration $g_{init}(n_{init}, l_{init})$ where g_{init} and l_{init} are respectively the initial valuations of the thread-local and procedure-local variables of the main thread, and n_{init} its starting node.

5.2 The reachability problem

As described previously in section 5.1.3, we can model the behaviour of a real multi-threaded program with a SDPN. Many static analysis techniques rely on being able to determine whether a given critical state is reachable or not from the starting configuration of a program.

Since checking reachability in a real program amounts to checking reachability in its corresponding SDPN w.r.t to the strict semantics, we want to solve the following *reachability problem*: given a SDPN M and two sets of configuration C and C', is there a configuration in C' that is reachable from C with regards to the *strict* semantics?

It has unfortunately been proven by Ramalingam in [Ram00] that, even if C and C' are regular, this problem is undecidable for synchronization sensitive pushdown systems, hence, SDPNs. Therefore, we reduce this problem to an analysis of the execution paths of SDPNs with *relaxed* semantics.

5.2.1 From the strict to the relaxed semantics

It is easy to see that the following theorem holds:

Theorem 20. *Let M be a SDPN and c, c' two configurations of M; $c \dashrightarrow^*_M c'$ if and only if $\exists n \geq 0$ such that $c \xrightarrow{\tau^n}{}^*_M c'$.*

Intuitively, an execution path with regards to the relaxed semantics of the form τ^n only uses internal actions or synchronized actions between two threads: a synchronization signal a is always paired with its co-action $\overline{a}$. Any configuration reachable using this path can be reached with regards to the strict semantics as well. Such a path is said to be *perfectly synchronized*.

Therefore, the reachability problem amounts to determining whether:

$$Paths_M(C, C') \cap \tau^* = \emptyset$$

that is, if there is an execution path from C to C' with regards to the relaxed semantics of the form τ^n. Obviously, we can't always compute $Paths_M(C, C')$. Our idea is therefore to compute an abstraction (over approximation) of $Paths_M(C, C')$ and check the emptiness of its intersection with τ^*: if it is indeed empty, then C' can't be reached from C with regards to the strict semantics.

It is worth noting that a configuration $p'_1 w'_1 p'_2 w'_2$ reachable from $p_1 w_1$ $p_2 w_2$ with regards to the strict semantics by synchronizing two rules $p_1 w_1 \xrightarrow{a} p'_1 w'_1$ and $p_2 w_2 \xrightarrow{\overline{a}} p'_2 w'_2$ using the synchronization rule **(1)** can obviously be reached with regards to the relaxed semantics by applying these two rules sequentially, using rule **(3)** twice, although the resulting path would obviously not be perfectly synchronized. Hence, the following theorem holds:

Theorem 21. *Let M be a SDPN and c, c' two configurations of M; c' is reachable from c w. r. t. the relaxed SDPN semantics if and only if it is reachable w. r. t. the DPN semantics.*

It implies that, since we can compute $pre^*(M, C)$ with regards to the DPN semantics thanks to Theorem 19, we can compute it with regards to the relaxed SDPN semantics as well.

5.2.2 Representing infinite sets of configurations

In order to compute an abstraction of $Paths_M(C, C')$ we need to be able to finitely represent infinite sets of configurations of a SDPN M. To do so, we introduce a class of finite automata called M-automata:

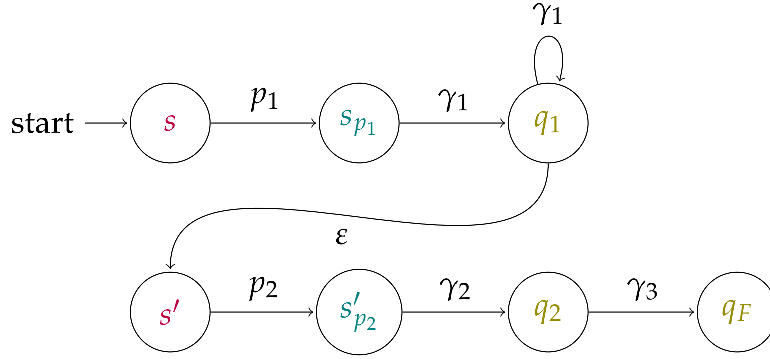

FIGURE 5.7: Accepting a regular set $p_1\gamma_1^+p_2\gamma_2\gamma_3$ with an M-automaton.

Definition 18 (Bouajjani et al. [BMOT05]). *Given a SDPN $M = (Act, P, \Gamma, \Delta)$, an M-automaton is a finite automaton $A = (\Sigma, S, \delta, s_{init}, F)$ such that:*

- *$\Sigma = P \cup \Gamma$ is the input alphabet;*
- *the set of states $S = S_C \cup S_S$ can be partitioned in two disjoint sets S_C and S_S;*
- *$\delta \subseteq S \times \Sigma \times S$ is the set of transitions;*
- *$\forall s \in S_C$ and $\forall p \in P$, there is at most a single state s_p such that $(s, p, s_p) \in \delta$; moreover, $s_p \in S_S$ and s is the only predecessor of s_p; transitions from states in S_C are always labelled with state symbols in P and go to dedicated states in S_S;*
- *states in S_C do not have exiting transitions labelled with letters in Γ;*
- *states in S_S do not have exiting transitions labelled in P; transitions labelled with letters in Γ always go to states in S_S;*
- *transitions from S_S to S_C are always labelled with ε; these are the only allowed ε-transitions in the M-automaton;*
- *$s_{init} \in S_C$ is the initial state;*
- *$F \subseteq S_C$ is the set of final states.*

An M-automaton A is designed in such a manner that every path accepting a configuration $p_1w_1 \dots p_nw_n$ is a sequence of sub-paths $s_i \xrightarrow{p_i}_A s_p \xrightarrow{w_i}{}^*_A q \xrightarrow{\varepsilon}_A s_{i+1}$ where $s_i \in S_C$, $s_{i+1} \in S_C$ and every state in the path $s_p \xrightarrow{w_i}{}^*_A q$ is in S_S. Being a finite state automaton, an M-automaton accepts a regular language that is a subset of $Conf_M$. Any regular language in $(P\Gamma^*)^*$ can be accepted by an M-automaton, as shown in Figure 5.7.

M-automata were introduced so that one could compute the set of predecessors of a DPN, hence, of a SDPN as well, by applying a saturation

procedure to an M-automaton accepting the set of starting configurations, as proven by Bouajjani et al. in [BMOT05].

5.3 Representing the set of paths

In this section, we introduce an automata-theoretic representation of sets of synchronized paths by adding extra labels to M-automata.

5.3.1 Π-configurations

Let $\Pi = 2^{Act^*}$ be the set of all possible languages on Act. We define a Π-configuration of M as a pair $(c, \pi) \in Conf_M \times Act^*$. We can extend the transition relation $\longrightarrow_M$ to Π-configurations with the following semantics:

$$\forall a \in Act, \text{ if } c \xrightarrow{a}_M c', \text{ then } \forall \pi \in Act^*, (c, a \cdot \pi) \longrightarrow_{M,\Pi} (c', \pi)$$

The configuration $(c, a \cdot \pi)$ is said to be an immediate Π-predecessor of (c', π). The reachability relation $\rightarrow^*_{M,\Pi}$ is the reflexive transitive closure of the relation $\longrightarrow_{M,\Pi}$.

Given a set of configurations C, we introduce the set of Π-predecessors $pre^*_\Pi(M, C)$ of all Π-configurations $(c', \pi) \in Conf_M \times Act^*$ such that $(c', \pi) \rightarrow^*_{M,\Pi} (c, \varepsilon)$ for $c \in C$. Obviously, we have:

$$pre^*_\Pi(M, C) = \left\{(c', \pi) \mid c' \in pre^*(M, C), \pi \in Paths_M(\{c'\}, C)\right\}$$

Intuitively, (c', π) is in $pre^*_\Pi(M, C)$ if one can reach a configuration $c \in C$ from c' by following a path π.

5.3.2 The shuffle product

Assuming we know the path languages of two different threads, we want to compute the path language of these two threads running in parallel.

Intuitively, this new language will be an interleaving of the two aforementioned sets, but can feature synchronized actions between the two threads as well.

To this end, we define inductively a shuffle operation $\sqcup\!\sqcup : Act^* \times Act^* \rightarrow \Pi$ such that, given two paths, their shuffle product is the set of all possible interleaving (with synchronization) of these paths.

Let $w = a_1 \dots a_n$ and $w' = b_1 \dots b_m$ be two such paths:

- $w \sqcup\!\sqcup \varepsilon = \varepsilon \sqcup\!\sqcup w = \{w\}$;
- if $b_1 \neq \overline{a_1}$, then: $w \sqcup\!\sqcup w' = a_1 \cdot [(a_2 \dots a_n) \sqcup\!\sqcup (b_1 \dots b_m)] \cup b_1 \cdot [(a_1 \dots a_n) \sqcup\!\sqcup (b_2 \dots b_m)]$;
- if $b_1 = \overline{a_1}$, then: $w \sqcup\!\sqcup w' = a_1 \cdot [(a_2 \dots a_n) \sqcup\!\sqcup (b_1 \dots b_m)] \cup b_1 \cdot [(a_1 \dots a_n) \sqcup\!\sqcup (b_2 \dots b_m)] \cup \tau \cdot [(a_2 \dots a_n) \sqcup\!\sqcup (b_2 \dots b_m)]$; two synchronized actions a_1 and $\overline{a_1}$ result in an internal action τ, hence, there is a component $\tau \cdot (w_1 \sqcup\!\sqcup w_2)$ of the shuffle product where the two paths synchronize.

The shuffle operation is obviously *commutative* and *associative*, as outlined in [lot97]. We can extend naturally the operation $\sqcup\!\sqcup$ to sets of paths: $\kappa_1 \sqcup\!\sqcup \kappa 2 = \bigcup_{\pi_1 \in \kappa_1, \pi_2 \in \kappa_2} (\pi_1 \sqcup\!\sqcup \pi_2)$. It is still commutative and associative.

5.3.3 Π-automata

We represent sets of Π-configurations of a SDPN M with a class of labelled M-automata, called Π-automata.

Definition 19. *Let $M = (Act, P, \Gamma, \Delta)$ be a SDPN, a Π-automaton is a finite automaton $A = (\Sigma, S, \delta, s_{init}, F)$ where $\Sigma = P \cup \Gamma$ is the input alphabet, $S = S_C \cup S_S$ is a finite set of control states with $S_C \cap S_S = \emptyset$, $\delta \subseteq (S_C \times P \times S_S) \cup (S_S \times \Gamma \times \Pi^\Pi \times S_S) \cup (S_S \times \{\varepsilon\} \times S_C)$ a finite set of transition rules (where Π^Π is the set of functions from Π to Π), s_{init} an initial state, and F a set of final states.*

Moreover, A is such that, if we consider the projection δ_Σ of δ on $S \times \Sigma^* \times S$, ignoring labels in Π^Π, then $(\Sigma, S, \delta_\Sigma, s_{init}, F)$ is a M-automaton.

Intuitively, a Π-automaton can be seen as an M-automaton whose transitions labelled by stack symbols in Γ have been given an additional label in Π^Π. We can consider a simple M-automaton as a Π-automaton if we label each transition in $S_S \times \Gamma \times S_S$ with the identity function.

While it would be simpler to label transitions of a M-automaton with subsets of Π, this representation would be flawed for the purpose of the algorithms outlined in Section 5.4.2. The intuition behind the use of functions in Π^Π as labels is detailed there.

The transition relation.

Let A be a Π-automaton. We define a simple transition relation $\longrightarrow_A$ according to the following semantics:

— if $(s, p, s') \in \delta \cap (S \times (P \cup \{\varepsilon\}) \times S)$, then $s \xrightarrow{p}_A s'$;

— if $(s, \gamma, e, s') \in \delta \cap (S_S \times \Gamma \times \Pi^\Pi \times S_S)$, then $s \xrightarrow{(\gamma,e)}_A s'$;

— if $s \xrightarrow{(w_1,e_1)}_A s_1$ and $s_1 \xrightarrow{(w_2,e_2)}_A s'$, then $s \xrightarrow{(w_1 w_2, e_1 \circ e_2)}_A s'$, where $\circ$ is the composition operation on functions.

We then extend inductively this transition relation to a full path relation $\Longrightarrow_A \subseteq S \times \Sigma^* \times \Pi^\Pi \times S$:

— for each $s \in S_S$, $s \xRightarrow{(\varepsilon,Id)}_A s$, where Id stands for the identity function;

— if there is a sequence $s_0 \xrightarrow{(\gamma_1,e_1)}_A s_1 \ldots s_{n-1} \xrightarrow{(\gamma_n,e_n)}_A s_n$ with $s_0, \ldots, s_n \in S_S$, then $s_0 \xRightarrow{(w,e)}_A s_n$, where $w = \gamma_1 \ldots \gamma_n$ and $e = e_1 \circ \ldots \circ e_n$; this is a simple sequence of actions along a single thread;

— if there is a sequence $s \xrightarrow{p_1}_A s_{p_1} \xRightarrow{(w_1,e_1)}_A q \xrightarrow{\varepsilon}_A s' \xrightarrow{p_2}_A s'_{p_2} \xRightarrow{(w_2,e_2)}_A q'$ such that $q', q \in S_S$ and $s, s' \in S_C$, then $s \xRightarrow{(w,e)}_A q'$, where $w = p_1 w_1 p_2 w_2$ and $e : y \longrightarrow e_1(\{\varepsilon\}) \sqcup\!\sqcup\, e_2(y)$; the automaton represents two parallel processes $p_1 w_1$ and $p_2 w_2$ whose abstract execution paths must be shuffled; moreover, since the first process will no longer be extended by further transitions of the automaton, we get rid of the variable of e_1 by considering $e_1(\{\varepsilon\})$ instead.

Note that this path relation is well-defined because $\sqcup\!\sqcup$ is associative.

A path $s_0 \xRightarrow{(c,e)}_A s_n$ is said to be an *execution* of A if $s_0 = s_{init}$. It is then said to be *accepting* if $s_n \in F$. We then say that A *accepts* (c, π) for all $\pi \in \Pi$ such that $\pi \in e(\{\varepsilon\})$. This way, accepting execution paths in Π-automata can be used to represent whole sets of paths. We define the set $L_\Pi(A)$ of all Π-configurations of M accepted by A.

5.4 Characterizing the set of paths

Let C be a regular set of configurations of a SDPN $M = (Act, P, \Gamma, \Delta)$. We want to define a Π-automaton $A_{pre^*_\Pi}$ accepting $pre^*_\Pi(M, C)$. Our intuition is to add extra labels in Π^Π to the M-automaton accepting $pre^*(M, C)$.

5.4.1 Computing $pre^*(M,C)$

Given a SDPN M and a regular set C of configurations of M accepted by an M-automaton A, we want to compute an M-automaton A_{pre^*} accepting $pre^*(M,C)$. Thanks to Theorem 21, we can apply the saturation procedure defined in [BMOT05] to A. Let us remind this procedure. Initially, $A_{pre^*} = A$, then we apply the following rules until saturation to A_{pre^*}:

(R_1) if $p\gamma \xrightarrow{a} p'w \in \Delta$ and $s \xrightarrow{p'w}{}^*_{A_{pre^*}} s'$ for $s \in S_S$, $s' \in S$, then add $s_p \xrightarrow{\gamma}{}^*_{A_{pre^*}} s'$;

(R_2) if $p\gamma \xrightarrow{a} p_1\gamma_1 p_2\gamma_2 \in \Delta$ and $s \xrightarrow{p_1\gamma_1 p_2\gamma_2}{}^*_{A_{pre^*}} s'$ for $s \in S_S$, $s' \in S$, then add $s_p \xrightarrow{\gamma}{}^*_{A_{pre^*}} s'$.

$\longrightarrow^*_{A_{pre^*}}$ stands for the transitive closure of the transition relation on the finite state automaton A_{pre^*}. The initial and final states remain the same.

Let us remind the intuition of these rules. We consider a sub-path $s \xrightarrow{p'w}{}^*_{A_{pre^*}} s'$. By design of an M-automaton, s should be in S_C and there should be a path $s \xrightarrow{p'} s_{p'} \xrightarrow{w}{}^* s'$ in the automaton A_{pre^*}. If we apply the saturation rule (R_1), we add an edge $s_p \xrightarrow{p} s'$ to A_{pre^*} and create a sub-path $s \xrightarrow{p\gamma}{}^* s'$ in the automaton. Therefore, if A_{pre^*} accepts a configuration $u_1 p'w'u_2$ with a path $s_{init} \xrightarrow{u_1}{}^* s_{p'} \xrightarrow{p'w}{}^* s' \xrightarrow{u_2}{}^* q_F$, $q_F \in F$, then it will accept its predecessor $u_1 p\gamma u_2$ as well with a path $s_{init} \xrightarrow{u_1}{}^* s_p \xrightarrow{p\gamma}{}^* s' \xrightarrow{u_2}{}^* q_F$. The role of (R_2) is similar.

Thus, when this saturation procedure ends, the M-automaton A_{pre^*} accepts the regular set $pre^*(M,C)$.

5.4.2 From $pre^*(M,C)$ to $pre^*_\Pi(M,C)$

Given a SDPN M and a regular set C of configurations of M accepted by an M-automaton A, we want to compute a Π-automaton $A_{pre^*_\Pi}$ accepting $pre^*_\Pi(M,C)$. To this end, we will add new labels to the M-automaton A_{pre^*}. Our intuition is the following: $A_{pre^*_\Pi}$ should be such that if we have $(c',\pi) \rightarrow^*_{M,\Pi} (c,\varepsilon)$, $c \in C$, then c can be reached from c' by a path π and $A_{pre^*_\Pi}$ should accept (c',π).

In order to compute $A_{pre^*_\Pi}$, we proceed as follows: we first accept configurations in C with the path ε, then, from there, set constraints on the labelling functions of transitions of A_{pre^*} depending on the relationship between edges introduced by the previous saturation procedure. This way, we build iteratively a set of constraints whose least solution is the set of execution paths from $pre^*(M, C)$ to C.

To this end, each transition t in A_{pre^*} labelled in Γ is given a second label $\lambda(t) \in \Pi^\Pi$. We compute a set of constraints whose smallest solution (according to the order $\subseteq$ of the language lattice) will be the labels $\lambda(t)$. If $t = q_1 \xrightarrow{\gamma} q_2$, then we write $\lambda(t) = \lambda(q_1, \gamma, q_2)$.

The need for functions.

We will explain intuitively here why we label the automaton with functions and not with sets of paths in Π.

Let us consider a M-automaton labelled by sets of paths as shown in Figure 5.8. To a thread in configuration $p_1\gamma_1$, we match a set $\{a\}$ and to a thread in $p_2\gamma_2\gamma_3$, we match $\{b, c\}$. We assume there is a rule $p\gamma \xrightarrow{d} p_1\gamma_1 p_2\gamma_2$ in M. By applying a saturation rule of the algorithm outlined in the previous section, we add a new dotted transition $s_{1_p} \xrightarrow{\gamma} q_2$. Intuitively, we label it with $d \cdot (\{a\} \sqcup\!\sqcup \{b\}) = \{dab, dba\}$: the label of the spawn action, followed by the synchronization of the paths matched to the two resulting threads.

However, assuming semantics similar to Π-automata, the automaton in Figure 5.8 would accept the configuration $(p\gamma\gamma_3, \{dabc, dbac\})$ by going through states s_1, p_1, q_2, and q_3. But intuitively, we want to accept the set $\{dabc, dbac, dbca\} = d \odot (\{a\} \sqcup\!\sqcup \{bc\})$ instead, as the action c can appear before a in the interleaving of the execution paths matched to the two threads.

We can consider instead the Π-automaton shown in Figure 5.9. The new dotted transition is labelled by the function $x \to d \cdot [a \sqcup\!\sqcup (b \cdot x)] = \{dabx, dbax, dbxa\}$. The variable x stands for the end of the paths matched to the second thread (in this case, the action c) that are not examined by the saturation rule but would nonetheless have to be shuffled with the path a of the first thread.

The automaton in Figure 5.9 has an execution path labelled by $(p\gamma\gamma_3,$ $f : x \to \{dabcx, dbacx, dbcxa\})$ when it goes through states s_1, p_1, q_2, and q_3. Therefore, it accepts all the paths $f(\{\varepsilon\}) = \{dabc, dbac, dbca\}$.

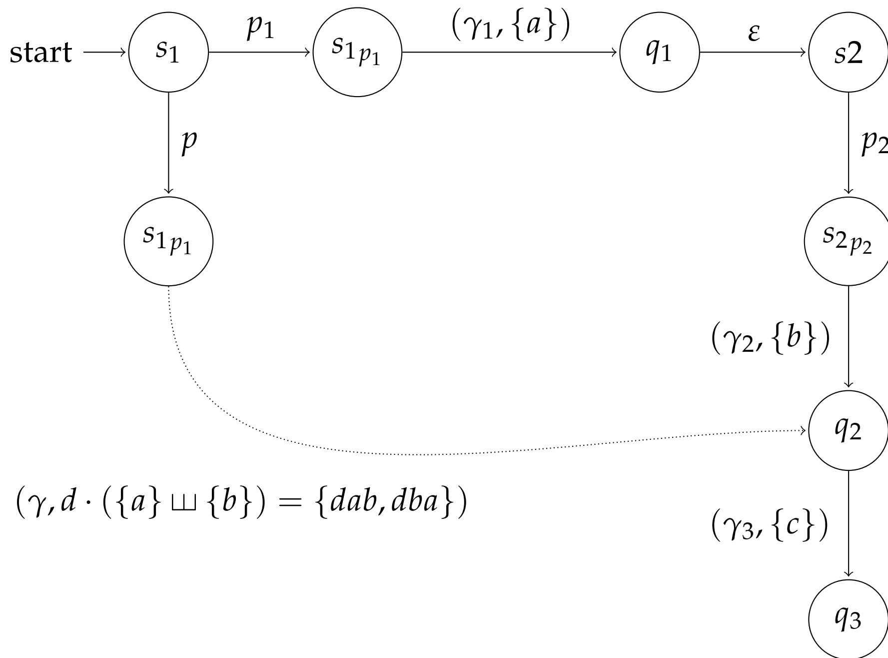

FIGURE 5.8: Using labels in Π.

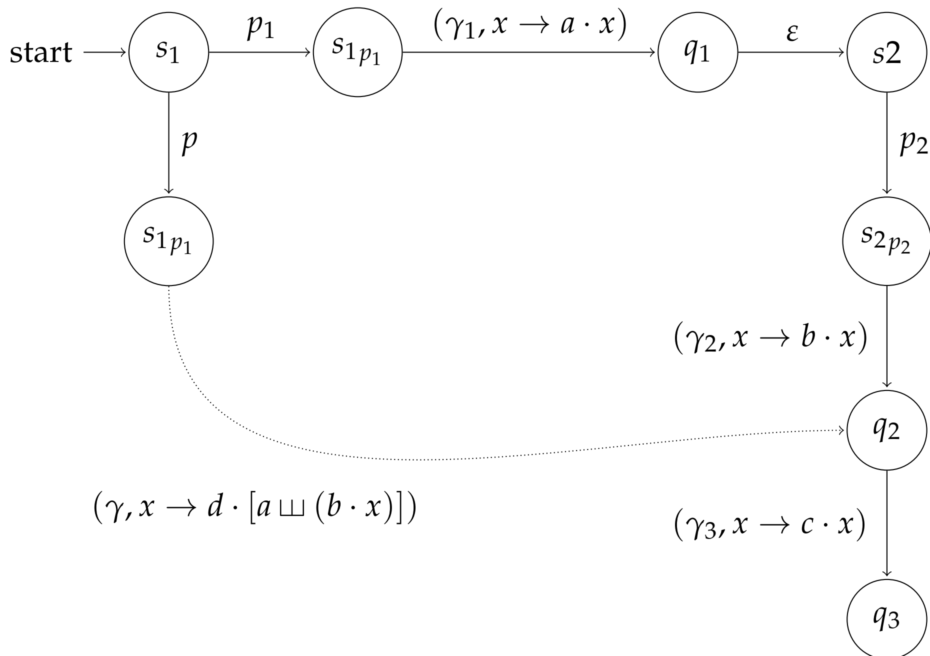

FIGURE 5.9: Using labels in Π^{Π}.

The constraints.

For two functions in Π^Π, we write that $f \subseteq g$ if $\forall x \in \Pi$, $f(x) \subseteq g(x)$. We now consider the following set of constraints on the labels of transitions of A_{pre^*} in $S_S \times \Gamma \times S_S$, where Q is the set of states of A:

(Z_1) if t belongs to A, then:
$$Id \subseteq \lambda(t)$$

(Z_2) for each rule $p\gamma \xrightarrow{a} p'\gamma' \in \Delta$, for each $q \in Q$, for each $s \in S_c$:
$$a \cdot \lambda(s_{p'}, \gamma', q) \subseteq \lambda(s_p, \gamma, q)$$

(Z_3) for each rule $p\gamma \xrightarrow{a} p'\varepsilon \in \Delta$, for each $s \in S_c$:
$$a \cdot Id \subseteq \lambda(s_p, \gamma, s_{p'})$$

(Z_4) for each rule $p\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \in \Delta$, for each $q \in Q$, for each $s \in S_c$:
$$\bigcup_{q' \in Q} a \cdot (\lambda(s_{p'}, \gamma_1, q') \circ \lambda(q', \gamma_2, q)) \subseteq \lambda(s_p, \gamma, q)$$

(Z_5) for each rule $p\gamma \xrightarrow{a} p_2\gamma_2 p_1\gamma_1 \in \Delta$, for each $q \in Q$, for each $s \in S_c$:
$$\bigcup_{s'' \xrightarrow{\varepsilon}_{A_{pre^*}} s'} a \cdot (\lambda(s_{p_2}, \gamma_2, s'')(\{\varepsilon\}) \sqcup \lambda(s'_{p_1}, \gamma_1, q)) \subseteq \lambda(s_p, \gamma, q)$$

In a manner similar to [BET05], we eventually define the labels of $A_{pre^*_\Pi}$ as the least solution of the set of constraints outlined above in the complete lattice of functions in Π^Π. By Tarski Theorem, this solution exists. The following theorem holds:

Theorem 22. *Let M be a SDPN and A an M-automaton accepting a regular set of configurations C. Then the Π-automaton $A_{pre^*_\Pi}$ accepts the set $pre^*_\Pi(M, C)$.*

Note that it doesn't mean we can compute the labels: an iterative computation of the least solution may not terminate. We now explain intuitively the meaning of these constraints.

The intuition.

If c is a configuration of C, then $A_{pre^*_\Pi}$ should accept (c, ε). This is expressed by constraint (Z_1).

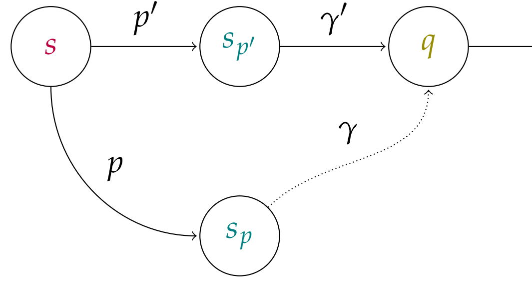

FIGURE 5.10: Case of a switch rule.

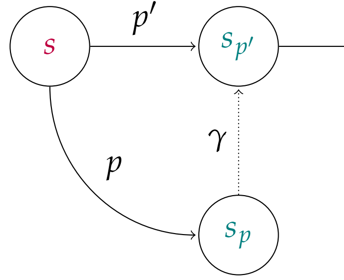

FIGURE 5.11: Case of a pop rule.

Let $c' = p'\gamma' w \in pre^*(M,C)$. If $p\gamma \xrightarrow{a} p'\gamma' \in \Delta$ and $(c',\pi) \in pre^*_\Pi(M,C)$, then $c = p\gamma w \in pre^*(M,C)$, $(c, a\cdot\pi) \rightarrow^*_{M,\Pi} (c',\pi)$, and $(c, a\cdot\pi) \in pre^*_\Pi(M,C)$. Hence, if $A_{pre^*_\Pi}$ accepts (c',π) and uses a transition $s_{p'} \xrightarrow{\gamma'} q$ while doing so, then it should accept $(c, a\cdot\pi)$ as well using a transition $s_p \xrightarrow{\gamma} q$, as shown in Figure 5.10. This is expressed by constraint (Z_2).

Let $c' = p'w \in pre^*(M,C)$. If $p\gamma \xrightarrow{a} p'\varepsilon \in \Delta$ and $(c',\pi) \in pre^*_\Pi(M,C)$, then $c = p\gamma w \in pre^*(M,C)$, $(c, a\cdot\pi) \rightarrow^*_{M,\Pi} (c',\pi)$, and $(c, a\cdot\pi) \in pre^*_\Pi(M,C)$. Hence, if $A_{pre^*_\Pi}$ accepts (c',π), then it should accept $(c, a\cdot\pi)$ as well using a transition $s_p \xrightarrow{\gamma} s'_p$, as shown in Figure 5.11. This is expressed by constraint (Z_3).

Let $c' = p'\gamma_1\gamma_2 w \in pre^*(M,C)$. If $p\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \in \Delta$ and also $(c',\pi) \in pre^*_\Pi(M,C)$, then $c = p\gamma w \in pre^*(M,C)$, $(c, a\cdot\pi) \rightarrow^*_{M,\Pi} (c',\pi)$, and $(c, a\cdot\pi) \in pre^*_\Pi(M,C)$. Hence, if $A_{pre^*_\Pi}$ accepts (c',π) and uses two transition $s_{p'} \xrightarrow{\gamma_1} q'$ and $q' \xrightarrow{\gamma_2} q$ while doing so, then it should accept $(c, a\cdot\pi)$ as well using a transition $s_p \xrightarrow{\gamma} q$, as shown in Figure 5.12. Moreover, there can be many possible intermediate states q' between s'_p and q such that $s_{p'} \xrightarrow{\gamma_1} q'$ and $q' \xrightarrow{\gamma_2} q$. In the automaton $A_{pre^*_\Pi}$, the path π should therefore be represented by the union for all possible intermediate state q' of the concatenation of the two labelling functions $\lambda(s_{p'}, \gamma_1, q')$ and $\lambda(q', \gamma_2, q)$. This is expressed by constraint (Z_4).

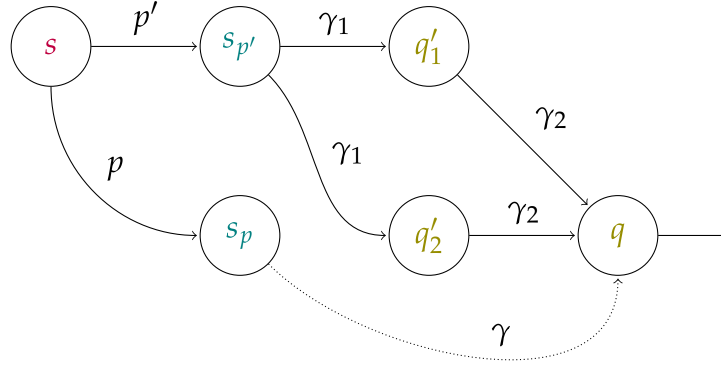

FIGURE 5.12: Case of a push rule.

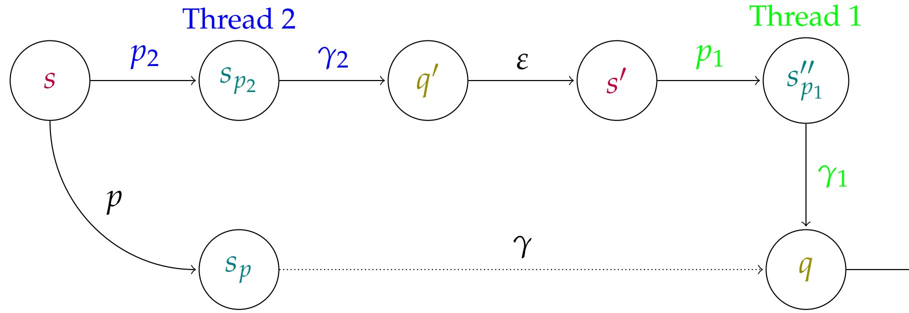

FIGURE 5.13: Case of a spawn rule.

Let $c' = p_2\gamma_2p_1\gamma_1w \in pre^*(M,C)$. If $p\gamma \xrightarrow{a} p_2\gamma_2p_1\gamma_1 \in \Delta$ and $(c',\pi) \in pre^*_\Pi(M,C)$, then $c = p\gamma w \in pre^*(M,C)$, $(c, a \cdot \pi) \rightarrow^*_{M,\Pi} (c',\pi)$, and $(c, a \cdot \pi) \in pre^*_\Pi(M,C)$, as shown in Figure 5.13. The two processes $p_2\gamma_2$ (thread 2 in 5.13) and $p_1\gamma_1$ (thread 1 in 5.13) are interleaved, hence, their execution paths must be shuffled: if π_1 is an execution path associated to $p_1\gamma_1$, and π_2, to $p_2\gamma_2$, then an path $\pi' = \pi_2 \sqcup \pi_1$ should be associated to $p_2\gamma_2p_1\gamma_1$. Moreover, if we consider a path $s_{p_2} \xrightarrow{\gamma_2} s'' \xrightarrow{\varepsilon} s' \xrightarrow{p_1} s'_{p_1} \xrightarrow{\gamma_1} q$ in the automaton $A_{pre^*_\Pi}$, then no path π_2 associated to $p_2\gamma_2$ can be extended further, and should therefore be represented by $(\lambda(s_{p_2},\gamma_2,s')(\{\varepsilon\}))$. Again, we must also consider each possible intermediate state s'' in the previous path, hence, an union of functions. This is expressed by constraint (Z_5).

5.4.3 Proof of Theorem 22

In order to prove Theorem 22, we first show that $A_{pre^*_\Pi}$ accepts every Π-predecessor of C.

Lemma 15. *We consider $c = p_1v_1 \dots p_nv_n \in C$, s_{init} the initial state of A, and F its set of final states. If $(c' = (p'_1w_1 \dots p'_lw_l), \pi) \rightarrow^*_{M,\Pi} (c, \varepsilon)$ for $\pi \in \Pi_\Pi$, then $\exists e \in \Pi^\Pi$ and $q_F \in F$ such that $\pi \in e(\varepsilon)$ and $s_{init} \xRightarrow{(c',e)}_{A_{pre^*_\Pi}} q_F$.*

We then prove that every configuration accepted by $A_{pre^*_\Pi}$ is a Π predecessor of C.

Lemma 16. *$\forall \pi \in Act^*$, if there is an accepting execution $s_{init} \xRightarrow{(c',e)}_{A_{pre^*_\Pi}} q_F$ such that $c' = (p'_1w'_1 \dots p'_nw'_n)$ and $\pi \in e(\{\varepsilon\})$, there is a configuration c such that $s_{init} \xrightarrow{c}_A q_F \in F$ and $(c', \pi) \rightarrow^*_{M,\Pi} (c, \varepsilon)$.*

Proof of Lemma 15

We can expand the labelling of functions to paths by composing labels along the paths, in a manner similar to the semantics of Π-automata. Hence, we can define $\lambda(s, w, q)$ such that $s \xRightarrow{(w,\lambda(s,w,q))}_A q$.

We prove by induction on k that if $c' = ((p'_1w'_1 \dots p'_lw'_l), \pi) \rightarrow^k_{M,\Pi} (c, \varepsilon)$, then $\exists e \in \Pi^\Pi$ and $q_F \in F$ such that $\pi \in e(\{\varepsilon\})$ and $s_{init} \xRightarrow{(c,e)}_{A_{pre^*_\Pi}} q_F$.

Basis: if $k = 0$, then $c' = c$, and $\pi = \varepsilon$. Because of constraint (Z_1), each transition t that exists both in A and $A_{pre^*_\Pi}$ is such that $Id \subseteq \lambda(t)$. Hence, if we follow an accepting execution $s_{init} \xRightarrow{(c,e)}_{A_{pre^*_\Pi}} q_F$ only using transitions in A, it must be such that $\varepsilon \in e(\{\varepsilon\})$. Hence, $A_{pre^*_\Pi}$ accepts (c, ε).

Induction step: we consider $(c_1 = (p''_1u_1 \dots p''_ju_j), \pi')$ such that $(c', \pi) \rightarrow^*_{M,\Pi} (c_1, \pi')$ and $(c_1, \pi') \rightarrow^k_{M,\Pi} (c, \varepsilon)$. We have the two following cases:

— if $j = l + 1$, a new process has been created from c' to c_1 by a rule of the form $r = p'_i\gamma \xrightarrow{a} p''_iu_ip''_{i+1}u'_{i+1}$. Then, there are t_1, t_2 in $(P\Gamma^*)^*$ such that $c_1 = t_1p''_iu_ip''_{i+1}u_{i+1}t_2$ and $c' = t_1p'_iw'_it_2$, and there is $u \in \Gamma^*$ such that $w'_i = \gamma u$ and $u_{i+1} = u'_{i+1}u$.

By induction, as shown in figure 5.14, there is a path in $A_{pre^*_\Pi}$ such that it accepts (c_1, π') and $\pi' \in \lambda_1(\{\varepsilon\}) \sqcup \lambda_2(\{\varepsilon\}) \sqcup \lambda_3 \circ \lambda_4(\{\varepsilon\}) \sqcup \lambda_5(\{\varepsilon\})$, where $\lambda_1 = \lambda(s_{init}, t_1, s)$, $\lambda_2 = \lambda(s_1, p''_iu_i, s_2)$, $\lambda_3 = \lambda(s_3, p''_{i+1}u'_{i+1}, s_4)$, $\lambda_4 = \lambda(s_4, u, s_5)$ and $\lambda_5 = \lambda(s_6, t_2, q_F)$. The saturation procedure creates a transition $(s_{1p'_i}, \gamma, s_4) = a \cdot (\lambda_2(\{\varepsilon\}) \sqcup \lambda_3)$. Hence:

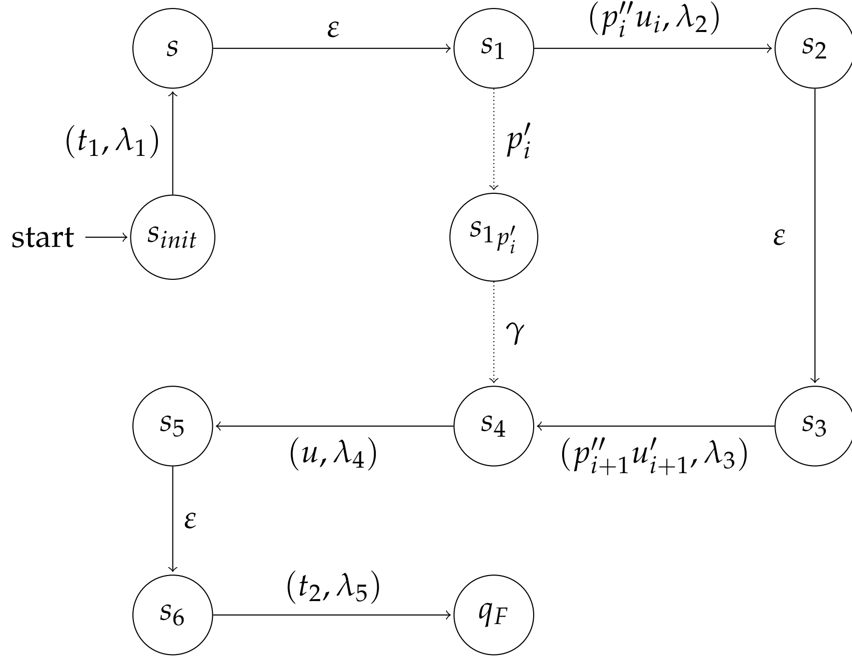

FIGURE 5.14: Adding an edge if a new process is spawned.

$$\pi' \in \lambda_1(\{\varepsilon\}) \sqcup \lambda_2(\{\varepsilon\}) \sqcup \lambda_3 \circ \lambda_4(\{\varepsilon\}) \sqcup \lambda_5(\{\varepsilon\})$$

$$\pi = a \cdot \pi' \in a \cdot (\lambda_1(\{\varepsilon\}) \sqcup \lambda_2(\{\varepsilon\}) \sqcup \lambda_3 \circ \lambda_4(\{\varepsilon\}) \sqcup \lambda_5(\{\varepsilon\}))$$

$$\pi \in \lambda_1(\{\varepsilon\}) \sqcup (a \cdot (\lambda_2(\{\varepsilon\}) \sqcup \lambda_3) \circ \lambda_4(\{\varepsilon\})) \sqcup \lambda_5(\{\varepsilon\})$$

Moreover, A_{pre^*} accepts c' with the path $s_{init} \xrightarrow{t_1 p'_i \gamma u t_2}_{A_{pre^*}} q_F$. But we have $\lambda(s_{init}, t_1 p'_i \gamma u t_2, q_F) \supseteq \lambda_1 \sqcup (a \cdot (\lambda_2(\{\varepsilon\}) \sqcup \lambda_3) \circ \lambda_4) \sqcup \lambda_5$ because of constraint (Z_5). $A_{pre^*_\Pi}$ therefore accepts π.

— if $j = l$, no new process has been created while moving from the configuration c' to the configuration c_1. Let $p'_i \gamma \xrightarrow{a}_M p''_i u'$ be the transition used to move from c' to c_1.

Let t_1 and t_2 be two words in $(P\Gamma^*)^*$ such that $c_1 = (t_1 p''_i u_i t_2)$ and $c' = (t_1 p'_i w'_i t_2)$. There is $u \in \Gamma^*$ such that $w'_i = \gamma u$ and $u_i = u'u$.

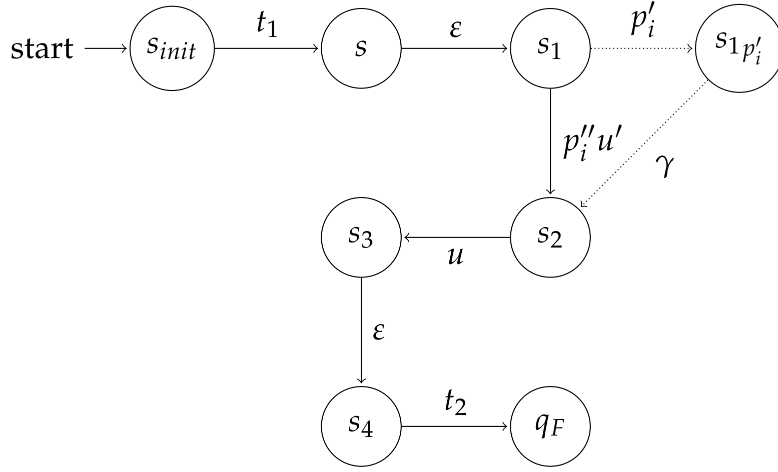

FIGURE 5.15: Adding an edge if no new process is spawned.

By induction, as shown in figure 5.15, there is a path π' such that $A_{pre^*_\Pi}$ accepts (c_1, π') and $\pi' \in \lambda_1(\{\varepsilon\}) \sqcup \lambda_2 \circ \lambda_3(\{\varepsilon\}) \sqcup \lambda_4(\{\varepsilon\})$, where $\lambda_1 = \lambda(s_{init}, t_1, s)$, $\lambda_2 = \lambda(s_1, p''_i u', s_2)$, $\lambda_3 = \lambda(s_2, u, s_3)$ and $\lambda_4 = \lambda(s_4, t_2, q_F)$.

The saturation procedures creates a transition $(s_{1p'_i}, \gamma, s_2)$ such that we have $\lambda(s_{1p'_i}, \gamma, s_2) \subseteq a \cdot (\lambda_2)$. Moreover:

$$\pi' \in \lambda_1(\{\varepsilon\}) \sqcup \lambda_2 \circ \lambda_3(\{\varepsilon\}) \sqcup \lambda_4(\{\varepsilon\})$$

$$\pi = a \cdot \pi' \in a \cdot (\lambda_1(\{\varepsilon\}) \sqcup \lambda_2 \circ \lambda_3(\{\varepsilon\}) \sqcup \lambda_4(\{\varepsilon\}))$$

$$\pi \in \lambda_1(\{\varepsilon\}) \sqcup a \cdot \lambda_2 \circ \lambda_3(\{\varepsilon\}) \sqcup \lambda_4(\{\varepsilon\})$$

The automaton A_{pre^*} accepts c' with a path $s_{init} \xrightarrow{t_1 p'_i \gamma u t_2}_{A_{pre^*}} q_F$ such that $\lambda(s_{init}, t_1 p'_i \gamma u t_2, q_F) \supseteq \lambda_1 \sqcup a \cdot (\lambda_2 \circ \lambda_3) \sqcup \lambda_4$ because of constraints (Z_2, Z_3, Z_4). The automaton $A_{pre^*_\Pi}$ therefore accepts π. □

Proof of Lemma 16

We prove this lemma by induction the length $|\pi|$ of π.

Basis: if $|\pi| = 0$, then $\pi = \varepsilon$; because of constraints (Z_2, Z_3, Z_4, Z_5), paths that use new transitions introduced by the saturation procedure are of length at least 1 and (c', π) can only be accepted by

a sequence of transitions following constraint (Z_1), hence, transitions in A. Therefore, $c' \in C$ and the property holds.

Induction step: if $|\pi| = k > 0$, let $\pi = a_1 \cdot a_2 \cdot \ldots \cdot a_k$. If $A_{pre^*_\Pi}$ accepts (c', π), there is an accepting execution $s_{init} = s_1 \xrightarrow{(p'_1 w'_1, e_1)}_{A_{pre^*_\Pi}} s_2 \ldots \xrightarrow{(p'_n w'_n, e_1)}_{A_{pre^*_\Pi}} s_{n+1} \in F$ such that $\pi \in e_1(\{\varepsilon\}) \sqcup e_2(\{\varepsilon\}) \ldots \sqcup e_n(\{\varepsilon\})$, with $e_i = \lambda(s_i, p'_i w'_i, s_{i+1})$.

We have the two following cases:

— if $a_1 \neq \tau$, for each $i \in \{1, \ldots, n\}$, we split $e_i(\{\varepsilon\})$ into two parts: $a_1 \cdot S_i$, which represents the path expressions in e_i starting with a_1, and S'_i, which stands for the path expressions starting with another symbol than a_1. We have $e_j(\{\varepsilon\}) = (a_1 \cdot S_j) \cup S'_j$ and $\pi = a_1 a_2 \ldots a_k \in ((a_1 \cdot S_1) \cup S'_1) \sqcup \ldots ((a_1 \cdot S_n) \cup S'_n)$.

Let i be such that $a_2 a_3 \ldots a_k \in ((a_1 \cdot S_1) \cup S'_1) \sqcup \ldots \sqcup (S_i) \sqcup \ldots ((a_1 \cdot S_n) \cup S'_n)$. The first symbol a_1 of π must appear in one (let's say the i-th) of the n sets $e_1(\{\varepsilon\}), \ldots, e_n(\{\varepsilon\})$.

We define $w'_i = \gamma_1 \gamma_2 \ldots \gamma_j$. We consider the sub-sequence $s_i = s_{i_1} \xrightarrow{(p'_i \gamma_1, \lambda_1)}_{A_{pre^*_\Pi}} s_{i_2} \ldots \xrightarrow{(\gamma_j, \lambda_j)} s_{i_{j+1}}$ of the accepting execution outlined earlier. We have $e_i = \lambda_1 \circ \lambda_2 \ldots \circ \lambda_j$.

Since there are paths starting with a_1 in e_i, then there are words starting with a_1 in λ_1 as well. Therefore, there is a rule $r = p'_i \gamma_1 \xrightarrow{a_1}_M q_i u$ in M and a transition $(s_{i q_i}, u, s_{i_2})$ in A_{pre^*}, from which a transition $(s_{i p'_i}, \gamma_1, s_{i_2})$ labelled by λ_1 can be added by the saturation rules in such a manner that the inequality $a_1 \cdot \lambda(s_{i q_i}, u, s_{i_2}) \subseteq \lambda_1$ holds.

The automaton $A_{pre^*_\Pi}$ then has an accepting execution labelled by:

$$c_1 = (p'_1 w'_1 \ldots p'_{i-1} w'_{i-1} q_i u \gamma_2 \ldots \gamma_j p'_{i+1} w'_{i+1} \ldots p'_n w'_n)$$

and:

$$(e_1 \sqcup \ldots e_{i-1} \sqcup (\lambda' \circ \lambda_2 \ldots \circ \lambda_j) \sqcup e_{i+1} \ldots \sqcup e_n)$$

where $\lambda' = \lambda(s_{i q_i}, u, s_{i_2})$.

We have $a_1 \cdot \lambda'(s_{i q_i}, u, s_{i_2}) \subseteq \lambda_1$ and $a_1 \cdot \lambda' \circ \lambda_2 \ldots \circ \lambda_j(\{\varepsilon\}) = a_1 \cdot S_i$, hence $\lambda' \circ \lambda_2 \ldots \circ \lambda_j(\{\varepsilon\}) = S_i$.

If we apply rule r to (c, π), we move to a configuration (c_1, π'), with $\pi' = a_2 \dots a_k$ and $\pi' \in ((a_1 \cdot S_1) \cup S'_1) \sqcup \dots \sqcup (S_i) \sqcup \dots ((a_1 \cdot S_n) \cup S'_n) = e_1(\{\varepsilon\}) \sqcup \dots e_{i-1}(\{\varepsilon\}) \sqcup (\lambda' \circ \lambda_2 \dots \circ \lambda_j)(\{\varepsilon\}) \sqcup e_{i+1}(\{\varepsilon\}) \dots \sqcup e_n(\{\varepsilon\})$. Moreover, $|\pi'| < |\pi|$.

If we apply the induction hypothesis, there is c such that $s_{init} \xrightarrow{c}_A q_F$ and $(c_1, \pi') \rightarrow^*_{M,\Pi} (c, \varepsilon)$. Since $(c', \pi) \rightarrow^*_{M,\Pi} (c_1, \pi')$, $(c', \pi) \rightarrow^*_{M,\Pi} (c', \varepsilon)$.

— if $a_1 = \tau$, from c, the automaton M can either move to another configuration if two of its processes synchronize with an action a or apply an internal action; we focus on the first case, the second case being similar to the previous unsynchronized action in terms of pushdown operations.

For $i \in \{1, \dots, n\}$, we split $e_i(\{\varepsilon\})$ in three parts: the set $a \cdot S_i$ of path expressions starting by a, the set $\overline{a} \cdot S'_i$ of path expressions starting by $\overline{a}$, and the set S''_i of path expressions starting with neither a nor $\overline{a}$. We have $e_i(\{\varepsilon\}) = (a \cdot S_i) \cup (\overline{a} \cdot S'_i) \cup S''_i$ and:

$$\pi = \tau \cdot a_2 \cdot \dots a_k \in ((a \cdot S_1) \cup (\overline{a} \cdot S'_1) \cup S''_1) \sqcup \dots$$

Let i and j be two integers such that $a_2 \cdot \dots a_k \in ((a \cdot S_1) \cup (\overline{a} \cdot S'_1) \cup S''_1) \sqcup \dots \sqcup \dots (S_i) \sqcup \dots (S'_j) \sqcup \dots ((a \cdot S_n) \cup (\overline{a} \cdot S'_n) \cup S''_n))$. The symbol τ of π comes from a shuffle of a word in $a \cdot S_i$, hence in $e_i(\{\varepsilon\})$, and a word in $\overline{a} \cdot S'_j$, hence in $e_j(\{\varepsilon\})$.

We define $w'_i = \gamma_1 \gamma_2 \dots \gamma_l$. We consider the sub-sequence $s_i = s_{i_1} \xRightarrow{(p'_i \gamma_1, \lambda_1)}_{A_{pre^*_\Pi}} s_{i_2} \dots \xRightarrow{(\gamma_l, \lambda_l)} s_{i_{l+1}}$ of the accepting execution outlined earlier.

We also define $w'_j = \alpha_1 \alpha_2 \dots \alpha_m$ and the sub-sequence $s_j = s_{j_1} \xRightarrow{(p'_j \alpha_1, \lambda'_1)}_{A_{pre^*_\Pi}} s_{j_2} \dots \xRightarrow{(\alpha_m, \lambda'_m)} s_{j_{m+1}}$ of the accepting execution outlined earlier.

We have $e_i = \lambda_1 \circ \lambda_2 \dots \circ \lambda_l$. There are paths starting by a in e_i, hence in λ_1 as well, and there is therefore a rule:

$$r_1 = p'_i \gamma_1 \xrightarrow{a}_M q_i u$$

By the saturation rules, from the transition $(s_{i q_i}, u, s_{i_2})$, we add a new transition $(s_{i p'_i}, \gamma_1, s_{i_2})$ labelled by λ_1 such that $a \cdot \lambda(s_{i q_i}, u, s_{i_2}) \subseteq \lambda_1$.

We have $e_j = \lambda'_1 \circ \lambda'_2 \ldots \circ \lambda'_m$. There are paths starting by $\bar{a}$ in e_j, hence in λ'_1 as well, and there is a rule:

$$r_2 = p'_j \alpha_1 \xrightarrow{\bar{a}} q_j u'$$

By the saturation rules, from the transition $(s_{j_{q_j}}, u', s_{j_2})$, we add a new transition $(s_{j_{p'_j}}, \alpha_1, s_{j_2})$ labelled by λ'_1 such that $\bar{a} \cdot \lambda(s_{j_{q_j}}, u', s_{j_2}) \subseteq \lambda'_1$.

We define $v_1 = \gamma_2 \ldots \gamma_l$ and $v_2 = \alpha_2 \ldots \alpha_m$.

The automaton $A_{pre^*_\Pi}$ then has an accepting execution labelled by $c_1 = (p'_1 w'_1 \ldots p'_{i-1} w'_{i-1} q_i u v_1 \ldots q_j u' v_2 \ldots p'_n w'_n)$ and $(e_1 \sqcup \ldots e_{i-1} \sqcup (\lambda \circ \lambda_2 \ldots \circ \lambda_l) \sqcup \ldots (\lambda' \circ \lambda'_2 \ldots \circ \lambda'_m) \ldots \sqcup e_n)$, where $\lambda = \lambda(s_{i_{q_i}}, u, s_{i_2})$ and $\lambda' = \lambda(s_{j_{q_j}}, u', s_{j_2})$.

We have $a \cdot \lambda \subseteq \lambda_1$ and $a \cdot \lambda \circ \lambda_2 \ldots \circ \lambda_l(\{\varepsilon\}) = a \cdot S_i$, hence $\lambda \circ \lambda_2 \ldots \circ \lambda_l(\{\varepsilon\}) = S_i$. Moreover, $\bar{a} \cdot \lambda' \subseteq \lambda'_1$ and $\bar{a} \cdot \lambda' \circ \lambda'_2 \ldots \circ \lambda'_m((\{\varepsilon\})) = \bar{a} \cdot S'_j$, hence $\lambda' \circ \lambda'_2 \ldots \circ \lambda'_m(\{\varepsilon\}) = S'_j$.

If we apply r_1 and r_2 in a synchronized manner to the configuration (c, π), we move to another configuration (c_1, π'), where $\pi' = a_2 \ldots a_k$ and $\pi' \in ((a \cdot S_1) \cup (\bar{a} \cdot S'_1) \cup S''_1) \sqcup \ldots (S_i) \sqcup \ldots (S'_j) \sqcup \ldots ((a \cdot S_n) \cup (\bar{a} \cdot S'_n) \cup S''_n) = e_1(\{\varepsilon\}) \sqcup \ldots e_{i-1}(\{\varepsilon\}) \sqcup (\lambda \circ \ldots \circ \lambda_l)(\{\varepsilon\}) \sqcup \ldots (\lambda' \circ \lambda'_2 \ldots \circ \lambda'_m)(\{\varepsilon\}) \ldots \sqcup e_n(\{\varepsilon\})$.

Since $|\pi'| < |\pi|$, we can apply the induction hypothesis. There is a configuration c such that $s_{init} \xrightarrow{c}_A q_F$ and $(c_1, \pi') \rightarrow^*_{M,\Pi} (c, (\varepsilon))$. Since $(c', \pi) \rightarrow^*_{M,\Pi} (c_1, \pi')$, it follows that $(c', \pi) \rightarrow^*_{M,\Pi} (c, \varepsilon)$. □

5.5 An abstraction framework for paths

We can't compute the exact set $Paths_M(C, C')$, we will therefore over-approximate it. To do so, we use the following mathematical framework, basing our technique on the approach presented by Bouajjani et al. in [BET03].

5.5.1 Abstractions and Galois connections

Let $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$ be the complete lattice of languages on Act.

Our abstraction of $\mathcal{L}$ requires a lattice $E = (D, \leq, \sqcup, \sqcap, \bot, \top)$, from now on called the *abstract lattice*, where D is a set called the abstract domain, as well as a pair of mappings (α, β) called a *Galois connection*, where $\alpha : 2^{Act^*} \to D$ and $\beta : D \to 2^{Act^*}$ are such that $\forall x \in 2^{Act^*}$, $\forall y \in D$, $\alpha(x) \leq y \Leftrightarrow x \subseteq \beta(y)$.

$\forall L \in \mathcal{L}$, given a Galois connection (α, β), we have $L \subseteq \beta(\alpha(L))$. Hence, the Galois connection can be used to over-approximate a language, such as the set of execution paths of a SDPN.

Moreover, it is easy to see that $\forall L_1, \forall L_2 \in \mathcal{L}$, $\alpha(L_1) \sqcap \alpha(L_2) = \bot$ if and only if $\beta(\alpha(L)) \cap \beta(\alpha(L)) = \emptyset$. We therefore only need to check if $\alpha(Paths_M(C, C')) \sqcap \alpha(\tau^*) = \bot$. From then on, $\alpha(Paths_M(C, C'))$ will be called the *abstraction* of $Paths_M(C, C')$, although technically the set $\beta(\alpha(Paths_M(C, C'))$ is the actual over-approximation.

5.5.2 Kleene algebras

We want to define abstractions of $\mathcal{L}$ such that we can compute the abstract path language $\alpha(Paths_M(C', C))$, assuming the sets C' and C are regular. In order to do so, we consider a special class of abstractions, called *Kleene abstractions*.

An idempotent semiring is a structure $K = (A, \oplus, \odot, \overline{0}, \overline{1})$, where $\oplus$ is an associative, commutative, and idempotent ($a \oplus a = a$) operation, and $\odot$ is an associative operation. $\overline{0}$ and $\overline{1}$ are neutral elements for $\oplus$ and $\odot$ respectively, $\overline{0}$ is an annihilator for $\odot$ ($a \odot \overline{0} = \overline{0} \odot a = \overline{0}$) and $\odot$ distributes over $\oplus$.

K is an Act-semiring if it can be generated by $\overline{0}$, $\overline{1}$, and elements of the form $v_a \in A$, $\forall a \in Act$. A semiring is said to be closed if $\oplus$ can be extended to an operator over countably infinite sets while keeping the same properties as $\oplus$.

We define $a^0 = \overline{1}$, $a^{n+1} = a \odot a^n$ and $a^* = \bigoplus_{n \geq 0} a^n$. Adding the $*$ operation to an idempotent closed Act-semiring K transforms it into a *Kleene algebra*.

5.5.3 Kleene abstractions

An abstract lattice $E = (D, \leq, \sqcup, \sqcap, \bot, \top)$ is said to be compatible with a Kleene algebra $K = (A, \oplus, \odot, \overline{0}, \overline{1})$ if $D = A$, $x \leq y \Leftrightarrow x \oplus y = y$, $\bot = \overline{0}$ and $\sqcup = \oplus$.

A *Kleene abstraction* is an abstraction such that the abstract lattice E is compatible with the Kleene algebra and the Galois connection $\alpha : 2^{Act^*} \to D$ and $\beta : D \to 2^{Act^*}$ is defined by:

$$\alpha(L) = \bigoplus_{a_1 \ldots a_n \in L} v_{a_1} \odot \ldots \odot v_{a_n}$$

$$\beta(x) = \left\{ a_1 \ldots a_n \in 2^{Act^*} \mid v_{a_1} \odot \ldots \odot v_{a_n} \leq x \right\}$$

Intuitively, a Kleene abstraction is such that the abstract operations $\oplus$, $\odot$, and $*$ can be matched to the union, the concatenation, and the Kleene closure of the languages of the lattice $\mathcal{L}$, $\overline{0}$ and $\overline{1}$ to the empty language and $\{\varepsilon\}$, v_a to the language $\{a\}$, the upper bound $\top \in K$ to Act^*, and the operation $\sqcap$ to the intersection of languages in the lattice $\mathcal{L}$.

In order to compute $\alpha(L)$ for a given language L, each word $a_1 \ldots a_n$ in L is matched to its abstraction $v_{a_1} \odot \ldots \odot v_{a_n}$, and we consider the sum of these abstractions.

We can check if $\alpha(Paths_M(C, C')) \sqcap \bigoplus_{n \geq 0} v_\tau^n = \bot$; if it is indeed the case, then $\beta(\alpha(Paths_M(C, C'))) \cap \tau^* = \emptyset$, and since $\beta(\alpha(Paths_M(C, C')))$ is an over-approximation $Paths_M(C, C')$, it follows that $Paths_M(C, C') \cap \tau^* = \emptyset$.

A *finite-chain* abstraction is an abstraction such that the lattice $(K, \oplus)$ has no infinite ascending chains. In this chapter, we rely on a particular class of finite-chain abstractions, called *finite-domain* abstractions, whose abstract domain K is finite, such as the following examples:

Prefix abstractions.

Let n be an integer and $W(n) = \{w \in Act^* \mid |w| \leq n\}$ be the set of words of length smaller than n. We define the n-th order *prefix* abstraction α_n^{pref} as follows: the abstract lattice $A = 2^W$ is generated by the elements $v_a = \{a\}, a \in Act$; $\oplus = \cup$; $U \odot V = \{\text{pref}_n(uv) \mid u \in U, v \in V\}$ where $\text{pref}_n(w)$ stands for the prefix of w of length n (or lower if w is of length smaller than n); $\overline{0} = \emptyset$; and $\overline{1} = \{\varepsilon\}$. From there, we build an abstract lattice where $\top = W$, $\sqcap = \cap$, and $\leq = \subseteq$. This abstraction

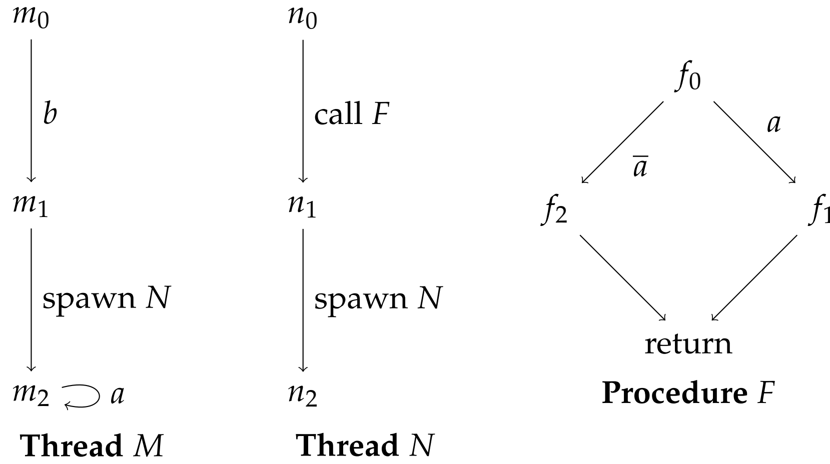

FIGURE 5.16: Applying a second order prefix abstraction to an example.

is accurate for the n-th first steps of a run, then approximates the other steps by Act^*.

We can apply a prefix abstraction of order 2 to the example shown in Figure 5.16. For ease of representation, we show a control flow graph, although we could use the procedure outlined in section 5.1.3 to compute an equivalent SDPN. We also consider without loss of generality that spawns, calls, and returns are silent ε-transitions.

We check that, starting from an initial set of configurations C with a single thread M in state m_0, the set C' where M is in state m_2 can't be reached with regards to the strict semantics.

We have $\alpha_2^{pref}(Paths_M(C,C')) = \{b, b\tau, ba, b\overline{a}\}$, $\alpha_2^{pref}(\tau^*) = \{\varepsilon, \tau, \tau\tau\}$, and $\alpha_2^{pref}(Paths_M(C,C')) \cap \alpha_2^{pref}(\tau^*) = \emptyset$, hence, C' can't be reached from C with regards to the strict semantics. Intuitively, the transition labelled with b in thread M can't synchronize as there isn't any transition labelled with $\overline{b}$ in the whole program.

Suffix abstractions.

Let W be the set of words of length smaller than n. We define the n-th order *suffix* abstraction α_n^{suff} as follows: the abstract lattice $A = 2^W$ is generated by the elements $v_a = \{a\}$, $a \in Act$; $\oplus = \cup$; $U \odot V = \{\text{suff}_n(uv) \mid u \in U, v \in V\}$ where $\text{suff}_n(w)$ stands for the suffix of w of length n (or lower if w is of length smaller than n); $\overline{0} = \emptyset$; and $\overline{1} = \{\varepsilon\}$. From there, we build an abstract lattice where $\top = W$, $\sqcap = \cap$, and $\leq = \subseteq$. This abstraction is accurate for the n-th last steps of a run, then approximates the other steps by Act^*.

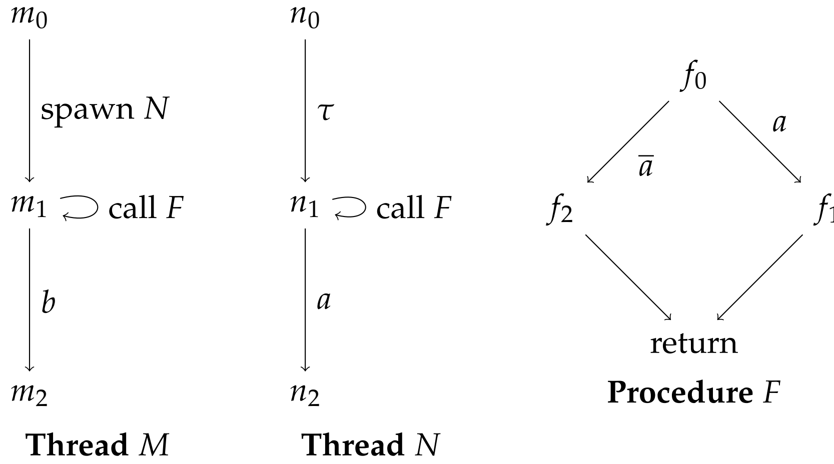

FIGURE 5.17: Applying a second order suffix abstraction to an example.

We apply a suffix abstraction of order 2 to the example shown in Figure 5.17. We check that, starting from an initial set of configurations C with a single thread M in state m_0, the set C' where M is in state m_2 can't be reached with regards to the strict semantics.

We have $\alpha_2^{suff}(Paths_M(C,C')) = \{b, ab, \bar{a}b, \tau b\}$, $\alpha_2^{suff}(\tau^*) = \{\varepsilon, \tau, \tau\tau\}$, and $\alpha_2^{suff}(Paths_M(C,C')) \cap \alpha_2^{suff}(\tau^*) = \emptyset$, hence, C' can't be reached from C with regards to the strict semantics. Intuitively, the transition labelled with b in thread M can't synchronize as there isn't any transition labelled with $\bar{b}$ in the whole program.

It is worth noting that the reachability problem in Example 5.17 can't be solved by a prefix abstraction, no matter its order. The reason is that $\forall n \geq 0$, there is an execution path $\tau^n b \in Paths_M(C,C')$, hence, $\tau^n \in \alpha_n^{pref}(Paths_M(C,C'))$. Intuitively, the two self-pointing loops of nodes m_1 and n_1 can synchronize.

Conversely, we can't use a suffix abstraction to solve the reachability problem in Example 5.16. The reason is that $\forall n \geq 0$, there is an execution path $b\tau^n \in Paths_M(C,C')$, hence, $\tau^n \in \alpha_n^{suff}(Paths_M(C,C'))$. Intuitively, the self-pointing loop of node m_2 can synchronize with the self-spawning loop in thread N.

Thus, these two abstractions (prefix and suffix) complement each other.

5.6 Abstracting the set of paths

Since we can't compute the solution of the constraints outlined in 5.4.2 on paths, our intuition now is to solve them in an abstract *finite* domain

defined by a Kleene abstraction where we can compute the least pre-fixpoint, in a manner similar to [BET05].

5.6.1 From the language of paths to the Kleene abstraction

We abstract the the complete lattice of languages $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$ by a finite domain Kleene abstraction on an abstract lattice $E = (D, \leq, \sqcup, \sqcap, \bot, \top)$ and a Kleene algebra $K = (A, \oplus, \odot, \overline{0}, \overline{1})$, as defined in Section 5.5.3.

Intuitively:

- the set Π is abstracted by K;
- the operator $\cdot$ is abstracted by $\odot$;
- the operator $\cup$ is abstracted by $\sqcup = \oplus$;
- the operator $\subseteq$ is abstracted by $\leq$;
- the operator $\cap$ is abstracted by $\sqcap$;
- $\emptyset$ is abstracted by $\bot = \overline{0}$;
- $\{\varepsilon\}$ is abstracted by $\overline{1}$;
- Act^* is abstracted by the set of path expressions Π_K, that is, the smallest set such that:
  - $\overline{1} \in \Pi_K$;
  - if $\pi \in \Pi_K$, then $\forall a \in Act$, $v_a \odot \pi \in \Pi_K$.

We can define K-configurations in $Conf_M \times \Pi_K$ and K-automata in a similar manner to Π-configurations and Π-automata. For a given set of configurations C, we introduce the set $pre^*_K(M, C)$ of K-configurations (c, π) such that $(c, \pi) \rightarrow^*_{M,K} (c', \overline{1})$ for $c' \in C$. The following property obviously holds:

$$pre^*_K(M, C) = \{(c', \pi) \mid c' \in pre^*(M, C), \pi \leq \alpha(Paths_M(\{c'\}, C))\}$$

The abstract path expression π is meant to be the abstraction of an actual execution path from c to c'.

To do so, we need to define the shuffle operation to paths expressions. However, it has to be *well-defined*: given two representations $v_{a_1} \odot \ldots \odot v_{a_n} = v_{b_1} \odot \ldots \odot v_{b_m}$ of a same path expression, $\forall w \in \{v_a | a \in Act\}^*$, we must have $(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup\, w = (v_{b_1}, \ldots, v_{b_m}) \sqcup\!\sqcup\, w$.

To this end, we first inductively define a shuffle operation $\sqcup\!\sqcup : (\{v_a | a \in Act\}^*)^2 \to K$ such that, given two sequences representing path expressions, their shuffle product is the set of all possible interleaving (with synchronization) of these sequences.

Let $w = (v_{a_1}, \ldots, v_{a_n})$ and $w' = (v_{b_1}, \ldots, v_{b_m})$ be two such sequences:

— $(v_{a_1}, \ldots v_{a_n}) \sqcup\!\sqcup (\varepsilon) = (\varepsilon) \sqcup\!\sqcup (v_{a_1}, \ldots v_{a_n}) = \{v_{a_1} \odot \ldots \odot v_{a_n}\}$;

— if $b_1 \neq \overline{a_1}$, then $((v_{a_1}, \ldots, v_{a_n})) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m}) = v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})) \oplus v_{b_1} \odot ((v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m}))$;

— if $b_1 = \overline{a_1}$, then $((v_{a_1}, \ldots, v_{a_n})) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m}) = v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})) \oplus v_{b_1} \odot ((v_{a_1}, \ldots v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m})) \oplus v_\tau \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m}))$; two synchronized actions a_1 and $\overline{a_1}$ result in an internal action τ, hence, there is a component $v_\tau \odot (w_1 \sqcup\!\sqcup w_2)$ of the shuffle product where the two paths synchronize.

We now that prove that the shuffle product is well-defined on path expressions for the prefix and suffix abstractions. We define the length $|\pi|$ of a path expression π as the length n of the smallest sequence $(v_{a_1}, \ldots, v_{a_n})$ such that $\pi = v_{a_1} \odot \ldots \odot v_{a_n}$, length 0 meaning that $\pi = \overline{1}$.

Note that this sequence is unique for the prefix and suffix abstractions; we can therefore define a function $\theta(\pi) = (v_{a_1}, \ldots, v_{a_n})$ that matches to a path expression its smallest representation.

Lemma 17. *The shuffle product is well-defined for the prefix abstraction.*

Proof. We will show by induction on $m + n$ that, given two sequences $(v_{a_1}, \ldots, v_{a_n})$ and $(v_{b_1}, \ldots, v_{b_m})$, we have $(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m}) = \theta(v_{a_1} \odot \ldots \odot v_{a_n}) \sqcup\!\sqcup \theta(v_{b_1} \odot \ldots \odot v_{b_m})$, i.e., that the shuffle of two path sequences is equal to the shuffle of their smallest representations.

— if $n \leq l$ and $m \leq l$, $\theta(v_{a_1} \odot \ldots \odot v_{a_n}) = (v_{a_1}, \ldots, v_{a_n})$ and $\theta(v_{b_1} \odot \ldots \odot v_{b_m}) = (v_{b_1}, \ldots, v_{b_m})$, by definition of the l-th prefix abstraction: indeed, the smallest representation in this abstraction of a word of length smaller than l is itself.

— if $n > l$ and $m > l$, note that $\theta(v_{a_1} \odot \ldots \odot v_{a_n}) = (v_{a_1}, \ldots, v_{a_l})$ and $\theta(v_{b_1} \odot \ldots \odot v_{b_m}) = (v_{b_1}, \ldots, v_{b_l})$ by definition of the l-th prefix. If we suppose that $b_1 = \overline{a_1}$:

$$
\begin{aligned}
& (v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m}) \\
= {} & v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})) \oplus v_{\overline{a_1}} \odot ((v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup \\
& (v_{b_2}, \ldots, v_{b_m})) \oplus v_\tau \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m})) \\
= {} & v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_{l+1}}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_l})) \oplus v_{\overline{a_1}} \odot ((v_{a_1}, \ldots, v_{a_l}) \sqcup\!\sqcup
\end{aligned}
$$

$$(v_{b_2}, \ldots, v_{b_{l+1}})) \oplus v_\tau \odot ((v_{a_2}, \ldots, v_{a_{l+1}}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_{l+1}}))$$

if we apply the induction hypothesis. However, by definition of the prefix abstraction, given two sequences $(v_{x_1}, \ldots, v_{x_l})$ and w and $\forall x_0 \in Act$, $v_{x_0} \odot ((v_{x_1}, \ldots, v_{x_l}) \sqcup\!\sqcup w) = v_{x_0} \odot ((v_{x_1}, \ldots, v_{x_{l-1}}) \sqcup\!\sqcup w)$. Intuitively, the symbols $v_{x_1}, \ldots, v_{x_{l-1}}$ have to be inserted after v_{x_0} but before v_{x_l}, and v_{x_l} will therefore be cut out of the prefix when we concatenate v_{x_0}. Hence:

$$(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})$$

$$= v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_l}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_l})) \oplus v_{\overline{a_1}} \odot ((v_{a_1}, \ldots, v_{a_l}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_l})) \oplus v_\tau \odot ((v_{a_2}, \ldots, v_{a_l}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_l}))$$

$$= (v_{a_1}, \ldots, v_{a_l}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_l})$$

The case $b_1 \neq \overline{a_1}$ is similar. Hence, the induction holds.

As a consequence, the shuffle product is well-defined: if $v_{a_1} \odot \ldots \odot v_{a_n} = v_{b_1} \odot \ldots \odot v_{b_m} = \pi$, then for all sequences w, $(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup w = \theta(\pi) \sqcup\!\sqcup w = (v_{b_1}, \ldots, v_{b_m}) \sqcup\!\sqcup w$. □

Lemma 18. *The shuffle product is well-defined for the suffix abstraction.*

Proof. We will show again by induction on $m + n$ that, given two sequences $(v_{a_1}, \ldots, v_{a_n})$ and $(v_{b_1}, \ldots, v_{b_m})$, we have $(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m}) = \theta(v_{a_1} \odot \ldots \odot v_{a_n}) \sqcup\!\sqcup \theta(v_{b_1} \odot \ldots \odot v_{b_m})$, i.e., that the shuffle of two path sequences is equal to the shuffle of their smallest representations.

— if $n \leq l$ and $m \leq l$, $\theta(v_{a_1} \odot \ldots \odot v_{a_n}) = (v_{a_1}, \ldots, v_{a_n})$ and $\theta(v_{b_1} \odot \ldots \odot v_{b_m}) = (v_{b_1}, \ldots, v_{b_m})$, by definition of the l-th suffix abstraction: indeed, the smallest representation in this abstraction of a word of length smaller than l is itself.

— if $n > l$ and $m > l$, note that $\theta(v_{a_1} \odot \ldots \odot v_{a_n}) = (v_{a_{n-l+1}}, \ldots, v_{a_l})$ and $\theta(v_{b_1} \odot \ldots \odot v_{b_m}) = (v_{b_{m-l+1}}, \ldots, v_{b_l})$ by definition of the l-th suffix. If we suppose that $b_1 = \overline{a_1}$:

$$(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})$$

$$= v_{a_1} \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})) \oplus v_{\overline{a_1}} \odot ((v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m})) \oplus v_\tau \odot ((v_{a_2}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_2}, \ldots, v_{b_m}))$$

$$= v_{a_1} \odot ((v_{a_{n-l+1}}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_{m-l+1}}, \ldots, v_{b_m})) \oplus v_{\overline{a_1}} \odot ((v_{a_{n-l+1}}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_{m-l+1}}, \ldots, v_{b_m})) \oplus v_\tau \odot ((v_{a_{n-l+1}}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_{m-l+1}}, \ldots, v_{b_m}))$$

if we apply the induction hypothesis. However, by definition of the suffix abstraction, given two sequences $(v_{x_1}, \ldots, v_{x_l})$ and w and $\forall x_0 \in Act$, $v_{x_0} \odot ((v_{x_1}, \ldots, v_{x_l}) \sqcup\!\sqcup w) = ((v_{x_1}, \ldots, v_{x_l}) \sqcup\!\sqcup w)$. Intuitively, the symbol v_{x_1} is concatenated to shuffled paths that are already of length greater than l, hence, will be cut out of the suffix. Therefore:

$$(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_1}, \ldots, v_{b_m})$$

$$= (v_{a_{n-l+1}}, \ldots, v_{a_n}) \sqcup\!\sqcup (v_{b_{m-l+1}}, \ldots, v_{b_m})$$

The case $b_1 \neq \overline{a_1}$ is similar. Hence, the induction holds.

As a consequence, the shuffle product is well-defined: if $v_{a_1} \odot \ldots \odot v_{a_n} = v_{b_1} \odot \ldots \odot v_{b_m} = \pi$, then for all sequences w, $(v_{a_1}, \ldots, v_{a_n}) \sqcup\!\sqcup w = \theta(\pi) \sqcup\!\sqcup w = (v_{b_1}, \ldots, v_{b_m}) \sqcup\!\sqcup w$. □

From now on, we consider that α is either the prefix or suffix abstraction of rank l.

5.6.2 Computing $pre_K^*(M, C)$

Given a SDPN M and a regular set C of configurations of M accepted by an M-automaton A, we want to compute a K-automaton $A_{pre_K^*}$ accepting $pre_K^*(M, C)$. To this end, we will add new labels in K^K to the M-automaton A_{pre^*}.

Let Q be the set of states of A, hence, of A_{pre^*} as well. We now consider the following set of constraints in the abstract domain on the labels of transitions of A_{pre^*} in $S_S \times \Gamma \times S_S$:

(Y_1) if t belongs to A, then:

$$Id \leq \lambda(t)$$

(Y_2) for each rule $p\gamma \xrightarrow{a} p'\gamma' \in \Delta$, for each $q \in Q$, for each $s \in S_c$:

$$v_a \odot \lambda(s_{p'}, \gamma', q) \leq \lambda(s_p, \gamma, q)$$

(Y_3) for each rule $p\gamma \xrightarrow{a} p'\varepsilon \in \Delta$, for each $s \in S_c$:

$$v_a \odot Id \leq \lambda(s_p, \gamma, s_{p'})$$

(Y_4) for each rule $p\gamma \xrightarrow{a} p'\gamma_1\gamma_2 \in \Delta$, for each $q \in Q$, for each $s \in S_c$:

$$\bigoplus_{q' \in Q} v_a \odot (\lambda(s_{p'}, \gamma_1, q') \circ \lambda(q', \gamma_2, q)) \leq \lambda(s_p, \gamma, q)$$

(Y_5) for each rule $p\gamma \xrightarrow{a} p_2\gamma_2 p_1\gamma_1 \in \Delta$, for each $q \in Q$, for each $s \in S_c$:

$$\bigoplus_{s'' \xrightarrow{\varepsilon}_{A_{pre^*}} s'} v_a \odot (\lambda(s_{p_2}, \gamma_2, s'')(\bar{1}) \sqcup \lambda(s'_{p_1}, \gamma_1, q)) \leq \lambda(s_p, \gamma, q)$$

Since α is a finite-domain abstraction, the set K^K of functions in K is finite as well. Let $t_1, \ldots, t_m$ be an arbitrary numbering of the transitions of $A_{pre^*_K}$ labelled with functions in the abstract domain and let $k_1, \ldots, k_n$ be an enumeration of the elements of the finite domain K ($n = |K|$). The labelling constraints of section 5.4.2 define a system of inequalities on $m * n$ variables $x_1, \ldots, x_{mn}$ such that its smallest solution is $t_1(k_1), \ldots, t_1(k_n)$, $t_2(k_1), \ldots, t_m(k_n)$. It is worth noting that we can replace two different inequalities $e_1(x) \leq t_i(x)$ and $e_2(x) \leq t_i(x)$ by a single inequality $e_1(x) \oplus e_2(x) \leq t_i(x)$. We therefore end up with a system of the form:

$$f_i(x_1, \ldots, x_{mn}) \leq x_i, \text{ for } i = 1, \ldots, mn$$

where the functions f_i are monomials in $K[x_1, \ldots, x_{mn}]$. Finding the least solution of this system of inequalities amounts to finding the least pre-fixpoint of the monotonic and continuous function:

$$F(x_1, \ldots, x_{mn}) = (f_1(x_1, \ldots, x_{mn}), \ldots, f_{mn}(x_1, \ldots, x_{mn}))$$

By Tarski's theorem, this fixpoint exists and is equal to $\bigoplus_{i \geq 0} F^i(\bar{0})$.

In a finite-domain, this iterative computation always terminates in a number of steps bounded by the length of the longest ascending chain in K, hence, l for a prefix or suffix abstraction of order l. There are mn functions f_i, each with a number of $\oplus$, $\odot$, and $\sqcup$ operations in $O(|\Delta| \cdot |Q|)$. Moreover, according to [BMOT05], the size of the automaton A_{pre^*} is $m = O(|Q|^2 \cdot |\Delta|)$. Each iteration step therefore features $O(n \cdot |\Delta|^2 \cdot |Q|^3)$ operations, and the whole procedure, $O(l \cdot n \cdot |\Delta|^2 \cdot |Q|^3)$ operations. For a prefix or suffix abstraction of order l, $n = 2^{|Act|^l}$, hence, a total of $O(l \cdot 2^{|Act|^l} \cdot |\Delta|^2 \cdot |Q|^3)$ operations.

5.6.3 Finding the abstraction

We can compute an automaton $A_{pre^*_K}$ that accepts the set $pre^*_K(M, C')$. We then want to find a K-automaton A' that accepts $pre^*_K(M, C') \cap C \times \Pi_K$.

To do so, we define the intersection $A' = (\Sigma, S', \delta', s'_{init}, F')$ of the automaton $A_{pre^*} = (\Sigma, S, \delta, s_{init}, F)$ with an M-automaton accepting C

called $A_1 = (\Sigma, S_1, \delta_1, s_{1,init}, F_1)$ accepting C, where $S' = S \times S_1$, $s'_{init} = (s_{init}, s_{1,init})$, $F = F \times F_1$, and $\delta = \{(q, q_1) \xrightarrow{a} (q', q'_1) \mid q \xrightarrow{a} q' \in \delta, q \xrightarrow{a} q' \in \delta_1\}$. Moreover, we label A' with abstract functions in such a manner that $\lambda((q, q_1), a, (q', q'_1)) = \lambda(q, a, q')$.

The K-automaton A' then obviously accepts $pre^*_K(M, C') \cap C \times \Pi_K$. The abstraction considered is therefore $\alpha(Paths_M(C, C')) = \bigoplus\{\pi \mid (c, \pi) \in L_K(A')\}$.

5.7 Using our framework in a iterative abstraction refinement scheme

Following the work of Chaki et al. in [CCK+06], we propose a semi-decision procedure that, in case of termination, allows us to answer exactly whether $Paths_M(C, C') \cap \tau^* = \emptyset$.

We first model a program as a SDPN M, as shown in section 5.1.3, its starting configurations as a regular set C, and a set of critical configurations whose reachability we need to study as another regular set C'.

We then introduce an iterative abstraction refinement scheme based on the finite-domain abstraction framework detailed previously, starting from $n = 1$.

Abstraction: we compute abstractions $\alpha(Paths_M(C, C'))$ of the set of executions paths for $\alpha = \alpha_n^{pref}$ and $\alpha = \alpha_n^{suff}$;

Verification: for $\alpha = \alpha_n^{pref}$ and $\alpha = \alpha_n^{suff}$, we check if $\alpha(Paths_M(C, C')) \sqcap \alpha(\tau^*) = \bot$; if it is indeed true, then we conclude that C' can't be reached from C using only internal or synchronized actions;

Counter-example validation: if there is such a path, we then check if our abstraction introduced a spurious counter-example; this can be done in a finite number of steps by checking if this counter-example can be reached within the n-th first or last execution steps of the program, depending on which abstraction (prefix or suffix) provided us with a counter-example; if the counter-example is not spurious, then we conclude that C' is reachable from C w.r.t. the strict semantics;

Refinement: if the counter-example was spurious, we go back to the first step, but use this time finite-domain abstractions of order $n + 1$.

If this procedure ends, we can decide the reachability problem.

5.8 A case study

We use an iterative abstraction refinement scheme to find an error in a Bluetooth driver for Windows NT. We consider here a simplified version of a driver found in [QW04] that nonetheless keeps the erroneous trace, in a manner similar to [CCK+06] and [PST07].

We model the network of processes in the driver as a SDPN. New requests for the driver are represented by thread spawns, and the driver's counter of active requests, by a counter on the stack, hence, a recursive process, making full use of our model's features.

We were able to discover the bug by applying our finite-domain abstraction in an iterative abstraction refinement scheme: we start from abstractions of order 1 and increment the order until we deduce that the erroneous configuration is reachable using a prefix abstraction of size 12. We then correct one of the program's subroutines accordingly and apply our iterative abstraction refinement scheme to prove it's now error-free.

Note that this bug was also discovered in [CCK+06, QW04, PST07]. However, our approach is more complete and precise than these works: [QW04] can only discover errors, whereas our scheme can also prove that a patched version of the driver is correct; [CCK+06] does not handle dynamic thread creation, and thus had to guess the number of threads for which the error arises; and [PST07] models thread creation as parallel calls (not as spawns), where the father process waits for its children to terminate in order to resume its execution.

5.8.1 The program

The driver consists of a certain number of processes running in parallel. Amongst these processes is an arbitrary number of requests meant to be handled by the driver. An internal counter records the number of requests currently handled by the driver: it is incremented when a request starts handling a task, and decremented once the request terminates. At any time, a special process in the driver may send a 'stop' signal; if it does, the driver switches an internal 'stopping' flag to true. The driver, however, can't stop yet and must wait until all requests have been processed.

Once the 'stopping' flag is switched on, requests may no longer perform their tasks and must instead end while decrementing the counter of active requests. Once the counter reaches zero, an internal 'stopping event' is switched to true, and the driver frees its allocated resources. If

a request tries to perform a task after the resources have been released, it must abort and send an error.

Our intuition is that the 'stop' signal can interrupt a request while the latter has already started handling its task after being told the driver was still running; the driver will then free its allocated resources and let the request resume its execution, leading to an error state where the request must abort because the resources it needs are now missing.

5.8.2 From the driver to the SDPN model

We model this network of processes as a SDPN. To do so, we consider that each thread has no internal variables and a single control state, as we switch the handling of the control flow to the stack by storing the current control point of each thread on its stack. The threads can communicate and handle flags and counters by using synchronized actions: as an example, two threads can synchronize over an action $stop$ in order to switch the 'stopping' flag (represented by a control point) to true; a function *Increment* can also synchronize with a counter over an action *incr* in order to increase this counter.

The driver uses the following processes:

COUNTER: this process counts the number of requests the driver receives plus the thread STOP-D; this number is set to 1 initially, is incremented when the driver receives a new request, and is decremented when a request ends;

STOP-D: this process may issue a request to stop the driver at any time; it has then to wait until all the other requests have finished their work, then, when it receives a signal sent by the function *Decrement*, can stop the driver and free its allocated resources;

STOPPING-FLAG: this process is either in state FALSE-STOP-FLAG (from then on FSF) or state TRUE-STOP-FLAG (TSF), depending on whether STOP-D is trying to stop the driver or not; it is initially in state FSF, and moves to state TSF if it receives a message from STOP-D; no new thread can enter the driver if this process is in TSF;

STOPPING-EVENT: it is either in state TRUE-STOP-EVENT (TSE) or FALSE-STOP-EVENT (FSE); this process enters state TSE if the driver stops, i.e. when the number of running REQUESTs reaches 0;

GEN-REQ: this process can spawn new requests as long as the driver is not trying to stop, that is, if STOPPING FLAG isn't in state TSF;

REQUEST: when a new REQUEST enters the driver, it has to increment the number stored in COUNTER, perform a task, then decrement this number before exiting the driver; it uses two functions *Increment* and *Decrement* to do so.

If a REQUEST tries to perform its task but the allocated resources of the driver have been released, the program reaches an error state. We will check the reachability of this state.

Each process can be modelled by a SDPN as follows:

The process COUNTER.

Let p_0 be its unique state. The number of threads is represented by a stack. Its stack alphabet is $\{0,1\}$. Initially, the stack contains the word 10, meaning that the number of request is zero and only STOP-D is running. It can then contain any word in 1^*0. The number of 1's in the stack corresponds to the number of running requests minus 1. The incrementation and decrementation procedures are done by receiving *incr* and *decr* actions from the functions *Increment* and *Decrement*.

COUNTER is represented by the following SDPN rules:

(r_{1a}) $p_0 1 \xrightarrow{\overline{incr}} p_0 11$ and (r_{1b}) $p_0 0 \xrightarrow{\overline{incr}} p_0 10$; these rules increment the counter when the process is asked to do so;

(r_2) $p_0 1 \xrightarrow{\overline{decr}} p_0 \varepsilon$; this rule decrements the counter when the process is asked to do so;

$(r_3 a)$ $p_0 1 \xrightarrow{not-zero} p_0 1$ and $(r_3 b)$ $p_0 0 \xrightarrow{is-zero} p_0 0$; these rules test whether the counter is 0 or not and send this information to other threads.

The process STOPPING-FLAG.

Let p_1 be its unique state. The process has two control points FSF and TSF. STOPPING-FLAG is represented by the following SDPN rules:

(r_4) p_1FSF $\xrightarrow{\overline{stop}}$ p_1TSF; the process receives a 'stop' request from STOP-D and changes its flag;

(r_5) p_1TSF $\xrightarrow{stopR}$ p_1TSF; the process sends a 'stop' message to the incoming REQUESTs;

(r_6) p_1FSF $\xrightarrow{not-stopR}$ p_1FSF; the process sends a 'non-stop' request to the incoming REQUESTs.

The process STOPPING-EVENT.

Let p_2 be its unique state. The process has two control points FSE and TSE. STOPPING-EVENT is represented by the following SDPN rules:

(r_7) $p_2\text{FSE} \xrightarrow{\overline{has-stopped}} p_2\text{TSE}$; the process receives an 'has-stopped' message and knows that the driver has stopped;

(r_8) $p_2\text{TSE} \xrightarrow{has-stopped} p_2\text{TSE}$; once the driver has stopped, it keeps sending the 'has-stopped' message;

(r_9) $p_2\text{FSE} \xrightarrow{non-stopped} p_2\text{FSE}$; the process sends a 'not-stopped' message if the driver is still running.

The process STOP-D.

Let p_3 be its unique state. It has three control points s_0, s_1, and R, the last one standing for 'release resources'. STOP-D is represented by the following SDPN rules:

(r_{10}) $p_3 s_0 \xrightarrow{stop} p_3 f_{Decrement} s_1$; STOP-D sends a 'stop' request to the process STOPPING-FLAG, and calls the function *Decrement*;

(r_{11}) $p_3 s_1 \xrightarrow{\overline{has-stopped}} p_3 R$; if the driver has stopped, the allocated resources are released.

The process REQUEST.

The process REQUEST executes the following instructions:

— it starts by calling a function *Increment*; this function returns -1 (stack symbol a_{-1}) if the STOPPING-FLAG is set to TRUE, otherwise, it increments the counter, and returns 0 (stack symbol a_0);

— if *Increment* returns 0, then REQUEST performs its task if it can assert that STOPPING-EVENT is in state FSE (i.e., that the driver is still running);

— it calls afterwards a function *Decrement* that decrements the counter; if this counter has reached 0, it sends a message to inform STOPPING-EVENT that the driver has stopped since there are no more requests running.

The process REQUEST has three control points r_0, r_{Work}, $r_{End-Work}$, and A, the last one standing for 'abort', and an unique state p_4. It can be modelled by the following SDPN rules:

(r_{12}) $p_4 r_0 \xrightarrow{\tau} p_4 f_{Increment}$; first, the function *Increment* is called;

(r_{13a}) $p_4 a_0 \xrightarrow{\tau} p_4 r_{Work} r_{End-Work}$ and (r_{13b}) $p_4 r_{Work} \xrightarrow{\tau} \varepsilon$; if the function *Increment* returns 0, then REQUEST can perform its (abstracted) work;

(r_{14}) $p_4 r_{End-Work} \xrightarrow{\overline{non-stopped}} p_4 f_{Decrement}$; once the work is finished, the process checks if the driver is still running, i.e. that process STOPPING-EVENT is in FSE;

(r_{15}) $p_4 r_{End-Work} \xrightarrow{\overline{has-stopped}} p_4 A$; if it is not the case, the program has reached an erroneous configuration and aborts.

The process GEN-REQ.

Let p_5 be its unique state, and g_0 its unique control point. GEN-REQ is represented by the following SDPN rule:

(r_{16}) $p_5 g_0 \xrightarrow{\overline{non-stopped}} p_4 r_0 p_5 g_0$; the process can spawn new requests as long as the driver is running.

The function Increment.

It has three control points i_0, a_0, and a_{-1}. The function *Increment* is represented by the following SDPN rules, as only REQUEST calls this function:

(r_{17}) $p_4 f_{Increment} \xrightarrow{\overline{stopR}} p_4 a_{-1}$; if STOPPING-FLAG is in TSF, the function returns -1;

(r_{18a}) $p_4 f_{Increment} \xrightarrow{\overline{not-stopR}} p_4 i_0$ and (r_{18b}) $p_4 i_0 \xrightarrow{incr} p_4 a_0$; otherwise, it returns 0 and increments the counter.

The function Decrement.

It has two control points d_0 and d_1. The function *Decrement* is represented by the following SDPN rules, where p stands for either p_3 or p_4, as only REQUEST and STOP-D call this function:

(r_{19}) $p f_{Decrement} \xrightarrow{decr} p d_0$; the counter is decremented;

(r_{20}) $p d_0 \xrightarrow{\overline{not-zero}} p\varepsilon$; then, if it has not reached 0, the function terminates;

(r_{21a}) $pd_0 \xrightarrow{\overline{is-zero}} pd_1$ and (r_{21b}) $pd_1 \xrightarrow{has-stopped} \varepsilon$; otherwise, a message 'has-stopped' is sent to STOPPING-EVENT.

We therefore model the program as a SDPN $M = (Act, P, \Gamma, \Delta)$, with:

— a set of control states $P = \{p_0, p_1, p_2, p_3, p_4, p_5\}$;

— a set of stack symbols $\Gamma = \{0, 1, \text{FSF}, \text{TSF}, \text{FSE}, \text{TSE}, s_0, s_1, R, r_0, r_{Work}, r_{End-Work}, A, g_0, f_{Increment}, i_0, a_0, a_{-1}, f_{Decrement}, d_0, d_1\}$;

— a set of actions $Act = \{\tau\} \cup L \cup \overline{L}$, where $L = \{incr, decr, is - zero, not - zero, stopR, not - stopR, has - stopped, non - stopped, stop\}$;

— a set of transitions $\Delta = \{r_{1a}, r_{1b}, \ldots, r_{21a}, r_{21b}\}$.

In the initial configuration, the counter is set to one, the flags in the processes STOPPING-FLAG and STOPPING-EVENT to FALSE, and all processes but REQUESTs (that will later be spawned by GEN-REQ) are running. We then need to check if the SDPN model of the program can reach with perfect synchronization a configuration where the process COUNTER has released its ressources and reached a control point R, while a process REQUEST has aborted its task and reached a control point A.

Our goal is therefore to check if, from the initial configuration $c_0 = p_0 10 \cdot p_1 FSF \cdot p_2 FSE \cdot p_3 s_0 \cdot p_5 g_0$, a configuration in the forbidden set of configurations:

$$C' = (P\Gamma^*)^* p_3 R (P\Gamma^*)^* p_4 A \Gamma^* (P\Gamma^*)^*$$

is reachable.

5.8.3 An erroneous execution path

We write $(r_i) \leftrightarrow (r_j)$ if we apply two rules that synchronize. The erroneous execution path is the following, starting from configuration c_0

$$p_0 10 \cdot p_1 FSF \cdot p_2 FSE \cdot p_3 s_0 \cdot p_5 g_0$$

(r_{16}) GEN-REQ spawns a REQUEST;

$$p_0 10 \cdot p_1 FSF \cdot p_2 FSE \cdot p_3 s_0 \cdot p_4 r_0 \cdot p_5 g_0$$

(r_{12}) REQUEST calls *Increment*;

$$p_0 10 \cdot p_1 FSF \cdot p_2 FSE \cdot p_3 s_0 \cdot p_4 f_{Increment} \cdot p_5 g_0$$

$(r_6) \leftrightarrow (r_{18a})$ STOPPING-FLAG sends $not - stopR$ to *Increment*;

$$p_0 10 \cdot p_1 FSF \cdot p_2 FSE \cdot p_3 s_0 \cdot p_4 i_0 \cdot p_5 g_0$$

$(r_{10}) \leftrightarrow (r_4)$ STOP-D sends $stop$ to STOPPING-FLAG;

$$p_0 10 \cdot p_1 TSF \cdot p_2 FSE \cdot p_3 f_{Decrement} s_1 \cdot p_4 i_0 \cdot p_5 g_0$$

$(r_{19}) \leftrightarrow (r_2)$ *Decrement* called by STOP-D sends *decr* to COUNTER;

$$p_0 0 \cdot p_1 TSF \cdot p_2 FSE \cdot p_3 d_0 s_1 \cdot p_4 i_0 \cdot p_5 g_0$$

$(r_{3b}) \leftrightarrow (r_{21a})$ COUNTER sends $is - zero$ to *Decrement* called by STOP-D;

$$p_0 0 \cdot p_1 TSF \cdot p_2 FSE \cdot p_3 d_1 s_1 \cdot p_4 i_0 \cdot p_5 g_0$$

$(r_{18b}) \leftrightarrow (r_{1b})$ *Increment* called by REQUEST resume its execution, returns 0, and sends *incr* to COUNTER;

$$p_0 10 \cdot p_1 TSF \cdot p_2 FSE \cdot p_3 d_1 s_1 \cdot p_4 a_0 \cdot p_5 g_0$$

$(r_{21a}) \leftrightarrow (r_8)$ *Decrement* called by STOP-D sends a signal $has - stopped$ to procedure STOPPING-EVENT;

$$p_0 10 \cdot p_1 TSF \cdot p_2 TSE \cdot p_3 s_1 \cdot p_4 a_0 \cdot p_5 g_0$$

$(r_8) \leftrightarrow (r_{11})$ *Decrement* STOPPING-EVENT sends $has - stopped$ to STOP-D that releases resources;

$$p_0 10 \cdot p_1 TSF \cdot p_2 TSE \cdot p_3 R \cdot p_4 a_0 \cdot p_5 g_0$$

(r_{13a}) REQUEST starts its task;

$$p_0 10 \cdot p_1 TSF \cdot p_2 TSE \cdot p_3 R \cdot p_4 r_{Work} r_{End-Work} \cdot p_5 g_0$$

(r_{13b}) REQUEST performs its work;

$$p_0 10 \cdot p_1 TSF \cdot p_2 TSE \cdot p_3 R \cdot p_4 r_{End-Work} \cdot p_5 g_0$$

$(r_8) \leftrightarrow (r_{15})$ STOPPING-EVENT sends $has - stopped$ to REQUEST that aborts;

$$p_0 10 \cdot p_1 TSF \cdot p_2 TSE \cdot p_3 R \cdot p_4 A \cdot p_5 g_0$$

This is an erroneous configuration reachable in 12 steps. We can find it using a prefix abstraction of order 12.

5.9 Related work

Wenner introduced in [Wen10] a model of *weighted dynamic pushdown networks* (WDPNs), extending the work of Reps et al. on *weighted pushdown systems* in [RSJM05] to DPNs. WDPNs share some similarities with our abstraction framework on SDPNs: each transition is labelled by a weight in a bounded idempotent semiring, these weights can be composed along execution paths, and the sum of the weights of all execution paths between two sets of configurations can be computed, provided that a simple extension of the original semiring to an abstract set of execution hedges can be found. WDPNs, however, do not feature simultaneous, synchronized actions between pushdown processes. Moreover, in order to be efficient, the extensions of the abstract domain have to be chosen on a case-by-case basis in order to label tree automata, whereas our framework works for every finite-domain abstraction and only uses finite state automata.

Multi-stack pushdown systems (MPDSs) are pushdown systems with two or more stacks, and can be used to model synchronized parallel programs. Qadeer et al. introduced in [QR05] the notion of context, that is, a part of an execution path during which only one stack of the automaton can be modified. The reachability problem within a bounded number of context switches is decidable for MPDSs. However, MPDSs have a bounded number of stacks and, unlike SDPNs, cannot therefore handle the dynamic creation of new threads.

Bouajjani et al. introduced in [BESS05] *asynchronous dynamic pushdown networks*, or ADPNs. This model extends DPNs by adding a global control state to the whole network as a mean of communication between processes; each pushdown process can then apply rules either by reading its own local state or the global state of the network. The reachability problem within a bounded number of context switches is decidable for ADPNs, where a context here stands for a part of an execution path during which transitions altering global variables are all executed by the same process. This is an under-approximation of the actual reachability problem for synchronized parallel programs, whereas we compute in this chapter an over-approximation of the same problem. The former can be used to find errors in a program but, unlike the latter, does not allow one to check that a program is free from errors.

Concurrent program with recursive procedures can be modeled as a network of synchronized pushdown systems. This model, called *communicating pushdown systems* (CPDSs), was introduced by Bouajjani et al. in [BET03]. The reachability problem being undecidable for this class of automata, a Kleene algebra framework was designed in order to find an over-approximation of the answer. Extensions of the abstraction framework of [BET03] were defined in [BET05, Tou05] to compute abstractions of execution paths of multi-threaded recursive programs communicating via rendez-vous. However, unlike SDPNs, the models considered in these articles cannot describe thread spawns, where the father of a new thread can resume its execution independently of its children.

5.10 Conclusion

Our first contribution in this chapter is a new pushdown system model that can handle both synchronization by rendez-vous between parallel threads and thread spawns. The reachability problem being undecidable for this class of automata, we seek to approximate it by abstracting the set of paths between two regular sets of configurations C and C'.

To this end, we introduce relaxed semantics with weaker synchronization constraints and extend the Kleene algebra abstraction framework shown in [BET03]. We label an automaton accepting the set of predecessors of C' with functions on a finite Kleene abstraction. These functions depend on a set of constraints computed according to the pushdown rules used during the saturation procedure.

This over-approximation allows us to define an iterative abstraction refinement scheme. We then apply it to find an erroneous execution trace in a Windows Bluetooth driver.

Chapter 6

Conclusion and Future Work

6.1 A brief summary of this thesis

In Chapter 3, we showed that model-checking hyper-properties expressed by the logic *HyperLTL* against PDSs is an undecidable problem. The restricted class of *visibly pushdown systems* did not help us to regain decidability. We therefore designed methods in order to approximate answers to the model-checking problem. We then applied these techniques to check security policies on toy examples.

In Chapter 4, we introduced *pushdown systems with an upper stack* (UPDSs), a model that allows us to keep track of the content of the memory region just above the stack. We proved that their reachability sets are not regular but nonetheless context-sensitive, and introduced techniques to under-approximate and over-approximate these sets. We then showed how this model could be used to look for safety or security flaws.

In Chapter 5, we introduced *synchronized dynamic pushdown systems* (SDPNs) in order to model multi-threaded programs with recursive procedure calls, synchronization, and dynamic creation of new threads. We designed an abstraction based on finite Kleene algebras in order to over-approximate the undecidable reachability problem. We then used this abstraction in an iterative abstraction refinement scheme that we applied to find an error in a Windows Bluetooth driver.

6.2 Future work

6.2.1 Tools for the model-checking of hyperproperties

In Chapter 3, we introduced algorithms to approximate the model-checking problem of HyperLTL against PDSs. An implementation of

these algorithms would be a valuable addition to existing model checking software.

We plan to design a new tool that would take as an input either a binary program or a Java program. We could perform in the former case a static analysis of the binary code with the tool Jakstab [KV08] that allows us to model the program as a control flow graph (CFG). By parsing this CFG, we could design a PDS model of the original code. In the latter case, we could use the PDS generated by the tool JimpleToPDSolver [HO10]. We could also handle C and C++ programs if we translate them into boolean programs with the tool SATABS [CKSY05].

If we could compute a PDS model of a program, we could then implement and apply one of the model-checking techniques described in Chapter 3 depending on the hyperproperty we want to verify.

6.2.2 Further reachability analysis of UPDSs

An obvious problem is to determine whether the reachability sets of the UPDS model introduced in Chapter 4 are context-free. However, the pumping lemma being significantly more complex for context-free languages, we did not manage to find a potential context-free counterexample.

If we managed to prove that at least one of the two reachability sets (predecessors and successors) of UPDSs is always context free, we then could determine whether a regular set C' of configurations is reachable from another regular set C. Indeed, since the emptiness of the intersection of a regular and a context-free language is decidable, we would only have to check whether $pre^*(C') \cap C = \emptyset$ or $C' \cap post^*(C) = \emptyset$.

6.2.3 Tools for the model-checking of concurrent programs

We plan to program a tool that would implement the abstraction framework for SDPNs designed in Chapter 5 in order to approximate the reachability problem.

We could model each thread of a program as a PDS that can spawn new PDSs and synchronize with other threads, using the tools SATABS [CKSY05] and JimpleToPDSolver [HO10] for C / C++ and Java programs respectively. We could then fully automate the iterative abstraction refinement scheme outlined in Section 5.7.

6.2.4 Abstract model-checking for SDPNs

A LTL model-checking framework for DPNs has been introduced by Song et al. in [ST15]. The reachability problem for SDPNs is unfortunately undecidable, and so is the model-checking problem as well.

We could nonetheless look for approximations using our abstraction framework based on finite Kleene algebras. Methods for abstracting the answer to the LTL and CTL model-checking problems for SDPNs would be a worthy addition to existing verification techniques on concurrent programs.

Bibliography

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory*, pages 135–150, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[BESS05] Ahmed Bouajjani, Javier Esparza, Stefan Schwoon, and Jan Strejček. Reachability analysis of multithreaded software with asynchronous communication. In Sundar Sarukkai and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 348–359, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BET03] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 62–73, New York, NY, USA, 2003. ACM.

[BET05] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. Reachability analysis of synchronized pa systems. *Electronic Notes in Theoretical Computer Science*, 138(3):153 – 178, 2005. Proceedings of the 6th International Workshop on Verification of Infinite-State Systems (INFINITY 2004).

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In Martín Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, pages 473–487, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BS90] Manuel E. Bermudez and Karl M. Schimpf. Practical arbitrary lookahead lr parsing. *Journal of Computer and System Sciences*, 41(2):230 – 250, 1990.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61 – 86, 1992.

[CCK+06] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing c programs with recursive calls. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, ETAPS '06, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 265–284, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CMP07] Dario Carotenuto, Aniello Murano, and Adriano Peron. 2-visibly pushdown automata. In Tero Harju, Juhani Karhumäki, and Arto Lepistö, editors, *Developments in Language Theory*, DLT '07, pages 132–144, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 232–247, London, UK, UK, 2000. Springer-Verlag.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 30–48, Cham, 2015. Springer International Publishing.

[GGH67] Seymour Ginsburg, Sheila A. Greibach, and Michael A. Harrison. Stack automata and compiling. *J. ACM*, 14(1):172–201, January 1967.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer-Verlag, Berlin, Heidelberg, 2008.

[HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[HO10] Matthew Hague and C.-H. Luke Ong. Analysing mu-calculus properties of pushdown systems. In *Proceedings of the 17th International SPIN Conference on Model Checking Software*, SPIN'10, pages 187–192, Berlin, Heidelberg, 2010. Springer-Verlag.

[HU68] J.E. Hopcroft and J.D. Ullman. Sets accepted by one-way stack automata are context sensitive. *Information and Control*, 13(2):114 – 133, 1968.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In Costas Courcoubetis, editor, *Computer Aided Verification*, pages 97–109, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[KV08] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2008.

[KYV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001.

[lot97] Combinatorics on words. Cambridge Mathematical Library, page 126. Cambridge University Press, 1997.

[PDT17] Adrien Pommellet, Marcio Diaz, and Tayssir Touili. Reachability analysis of pushdown systems with an upper stack. In *LATA*, volume 10168 of *Lecture Notes in Computer Science*, pages 447–459, 2017.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PST07] Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 254–257, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[PT17] Adrien Pommellet and Tayssir Touili. Static analysis of multithreaded recursive programs communicating via rendez-vous. In *APLAS*, volume 10695 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2017.

[PT18] Adrien Pommellet and Tayssir Touili. Model-checking hyperltl for pushdown systems. In *SPIN*, 2018.

[PW91] Fernando C. N. Pereira and Rebecca N. Wright. Finite-state approximation of phrase structure grammars. In *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics*, ACL '91, pages 246–255, Stroudsburg, PA, USA, 1991. Association for Computational Linguistics.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, March 2000.

[RSJM05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1):206 – 263, 2005. Special Issue on

the Static Analysis Symposium 2003.

[Set10] Anil Seth. Global reachability in bounded phase multi-stack pushdown systems. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 615–628, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[ST11] Fu Song and Tayssir Touili. Efficient ctl model-checking for pushdown systems. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 – Concurrency Theory*, pages 434–449, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[ST15] Fu Song and Tayssir Touili. Model-checking dynamic pushdown networks. *Formal Aspects of Computing*, 27(2):397–421, Mar 2015.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2):217 – 237, 1987.

[TMP07] S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 161–170, July 2007.

[Tou05] Tayssir Touili. Dealing with communication for dynamic multithreaded recursive programs. In *Verification of Infinite-State Systems with Applications to Security, Proceedings of the NATO Advanced Research Workshop "Verification of Infinite State Systems with Applications to Security VISSAS 2005", Timisoara, Romania, March 17-22, 2005*, pages 213–227, 2005.

[UM13] Yuya Uezato and Yasuhiko Minamide. Pushdown systems with stack manipulation. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis*, pages 412–426, Cham, 2013. Springer International Publishing.

[Var96] Moshe Y. Vardi. *An automata-theoretic approach to linear temporal logic*, pages 238–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[Wal00] Igor Walukiewicz. Model checking ctl properties of pushdown systems. In Sanjiv Kapoor and Sanjiva Prasad, editors, *FST TCS 2000: Foundations of Software Technology and*

Theoretical Computer Science, pages 127–138, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[Wal01] Igor Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234 – 263, 2001.

[Wen10] Alexander Wenner. Weighted dynamic pushdown networks. In Andrew D. Gordon, editor, *Programming Languages and Systems*, ESOP '10, pages 590–609, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.