Abstract lattices for the verification of systèmes with stacks and queues

Tristan Le Gall

N° d'ordre: 3755

THÈSE

Présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Tristan Le Gall

Équipe d'accueil : VerTeCS - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Abstract Lattices for the Verification of Systems with Queues and Stacks

Soutenue le 2 juillet 2008 devant la commission d'examen

M. :	Olivier	Ridoux	Président
MM. :	Ahmed	Bouajjani	Rapporteurs
	Jean-François	Raskin	
MM. :	Bertrand	Jeannet	Examinateurs
	Grégoire	Sutre	
	Thierry	Jeron	

Remerciements

Je remercie Ahmed Bouajjani et Jean-François Raskin pour avoir accepté de lire cette thèse et d'en être les rapporteurs. C'est un honneur d'avoir dans mon jury des chercheurs si connus dans le domaine de la vérification des systèmes infinis.

Je remercie Olivier Ridoux pour avoir accepté de présider mon jury de soutenance. Je le remercie également pour m'avoir encadré durant ma maîtrise d'informatique, pour un stage qui m'a donné un premier aperçu du monde de la recherche.

Je remercie Grégoire Sutre pour m'avoir accueilli brièvement à Bordeaux, et pour avoir accepté de faire partie de ce jury de thèse. J'espère que notre collaboration scientifique sera fructueuse.

Je remercie Bertrand Jeannet pour m'avoir encadré, aussi bien de près, à Rennes, que de loin, depuis Grenoble. Merci pour m'avoir soutenu dans mes moments de doute, et pour avoir travaillé avec moi pendant plus de quatre ans (et ce n'est pas fini, je l'espère). Je suis son premier thésard et, troublante coïncidance, il est mon premier directeur de thèse. Comme quoi le hasard fait bien les choses.

Je remercie Thierry Jéron pour son soutien, son écoute et sa patience aussi bien comme directeur de thèse que comme chef d'équipe. Il a été disponible autant que possible, et je sais que c'est une qualité assez rare parmis les chercheurs ayant comme lui autant de charges administratives et de charges d'encadrement.

Je remercie aussi les autres membres de l'équipe Vertecs pour m'avoir supporté pendant presque quatre ans: Camille, Nathalie, Patricia (si si tu es dans Vertecs), Christophe, Florimond, Gwenaël, Hatem, Hervé, Jérémy, Vlad, Wassim. Rares sont les équipes de recherche où règne une ambiance aussi bonne, aussi agréable pour travailler. Je remercie également Thierry Massart et Gabriel Kalyon, qui sont venus à Rennes si longtemps que l'on peut les déclarer membres de l'équipe Vertecs/antenne bruxelloise.

Contents

Résumé des travaux de thèse

Nota Bene: *This is an "extended summary" of the document, in french, which is mandatory since the thesis is written in english.*

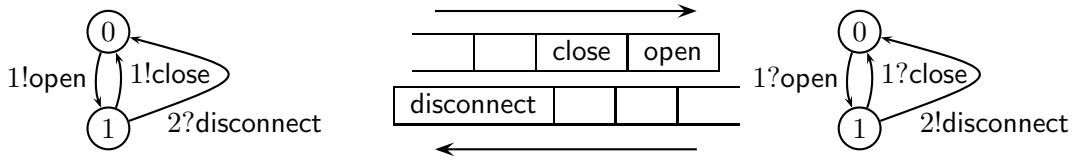
La vérification de systèmes informatiques complexes est actuellement un domaine de recherche très actif en informatique fondamentale. En témoigne la récente attribution du prix Turing à trois chercheurs de ce domaine : Edmund Clarke, Allen Emerson et Joseph Sifakis.

La vérification des systèmes dits infinis est un problème ardu, car il s'agit de déterminer quels sont les comportements possibles d'un système ayant un nombre très grand, voire infini, de configurations possibles. Pour déterminer ces comportements, on est amené à calculer, à partir d'un ensemble de configurations de départ C_0, l'ensemble des configurations accessibles en une étape, $\mathrm{Post}(C_0)$, puis en un nombre d'étapes quelconque, $\mathrm{Post}^*(C_0)$, appelé ensemble d'atteignabilité.

Plusieurs types de méthodes symboliques sont utilisées pour vérifier, ou tenter de vérifier, ces systèmes infinis. Nous en détaillerons deux. La première, ce sont les *techniques d'accélération*. À la fin des années 1990, elles ont été utilisées pour vérifier des systèmes avec des files de communication, avec un certain succès. Toutefois, elles ont le défaut de ne proposer que des semi-algorithmes, c'est-à-dire des algorithmes qui donnent le bon résultat, quand ils terminent, sachant que leur terminaison n'est pas garantie. Nous détaillerons plus loin ces techniques d'accélération pour les systèmes communiquant par files avec une politique FIFO (*first in - first out*).

La seconde famille de techniques est appelée *l'interprétation abstraite*. On part du constat que l'ensemble $\mathrm{Post}^*(C_0)$ s'exprime comme étant le plus petit point-fixe de la fonction $X \mapsto C_0 \cup \mathrm{Post}(X)$. L'idée est alors de calculer une approximation de cet ensemble non pas dans l'ensemble des ensembles d'états (le treillis concret), mais dans un ensemble plus simple, appelé treillis abstrait. Il est parfois nécessaire de disposer, dans ce treillis abstrait, d'un *opérateur d'élargissement* ∇.

Le principal sujet de recherche de cette thèse est de vérifier des systèmes infinis avec des files FIFO ou des piles par interprétation abstraite. Les systèmes avec file FIFO seront modélisés par des automates communicants (CFSM en anglais), comme par exemple celui représenté sur la figure 1.



(a) Client (b) Files (c) Serveur

Figure 1: Exemple d'automates communicants (CFSM).

Vérification des systèmes à files FIFO avec les techniques d'accélération

Principe des techniques d'accélération

Les techniques d'accélération partent du constat que, lorsque l'on veut faire une exploration de l'espace des états d'un CFSM[1], la principale difficulté est de prendre en compte l'effet d'une boucle de la structure de contrôle. En effet, la partie "linéaire" de la structure de contrôle ne peut engendrer qu'un nombre fini de configurations, borné par la taille de la structure. Par contre, une boucle peut générer un ensemble de plus en plus grand de configurations. Le raisonnement qui sous-tend les techniques d'accélération est le suivant :

- On part d'une représentation exacte de l'ensemble initial de configurations. Cette représentation est choisie dans une certaine classe, par exemple c'est un langage régulier L_0.
- On prouve que, pour tout ensemble L de configurations appartenant à cette classe, l'ensemble des configurations pouvant être atteintes en itérant la boucle θ un nombre arbitraire de fois, $\mathrm{Post}^*_\theta(L)$, appartient à cette même classe. En général, cette preuve est constructive et fournit un moyen de calculer, pour tout langage L et toute boucle θ, le langage $\mathrm{Post}^*_\theta(L)$.
- On explore l'ensemble des états en remplaçant, lorsque l'on rencontre une boucle θ, l'ensemble L des configurations déjà atteignables par $\mathrm{Post}^*_\theta(L)$.

Le nom d'accélération se justifie car, en ajoutant en une seule opération tous les états de $\mathrm{Post}^*_\theta(L)$, on accélère considérablement l'exploration de l'ensemble des états. Cette exploration peut toutefois ne pas terminer. En effet, si on a un nombre infini de boucles, même en sachant accélérer chaque boucle, on peut avoir toujours avoir de nouvelles boucles à accélérer.

L'exemple de la figure 2 illustre ce problème. On représente le contenu de la file par un langage régulier. Une boucle est une série d'émissions de a ou de b. Aussi,

[1] Les techniques d'accélération s'appliquent également à d'autres types de systèmes infinis, par exemple les automates à compteurs. On ne détaillera ici que les techniques d'accélération conçues pour l'analyse des systèmes FIFO.

pour tout langage régulier et toute boucle θ, on a $\mathrm{Post}^*_\theta(L) = L.\theta^*$. On peut imaginer que l'on commence par accélérer selon la boucle élémentaire $!a$; on obtient un ensemble de configurations $L_1 = a^*$. Puis on accélère selon la boucle élémentaire $!b$; on obtient $L_2 = a^*.b^*$. Ensuite, on accélère selon la boucle $!a!b$, on obtient $L_3 = a^*.b^*.(a.b)^*$. On peut ainsi continuer indéfiniment sans pour autant obtenir l'ensemble des configurations accessibles, $(a + b)^*$, sauf à procéder plus astucieusement, en remarquant par exemple que l'on peut permuter les deux boucles $!a$ et $!b$.

Figure 2: Automate avec des boucles imbriquées.

Cette limitation est commune à toutes les techniques d'accélération. En outre, chaque technique est limitée par sa représentation de l'ensemble des configurations. Nous allons maintenant détailler les différentes représentations utilisées pour faire de l'accélération pour des systèmes FIFO, qui sont :

- les *Queue-Contents Decision Diagrams (QDD)*,
- les *Constrained Queue-contents Decision Diagrams (CQDD)*,
- les *Simply Regular Expressions (SRE)*,
- les *Semi-Linear Regular Expressions (SLRE)*.

Queue-Contents Decision Diagrams (QDD)

Les diagrammes de décision pour les contenus de file, ou QDD en anglais, sont des automates finis représentant le contenu de plusieurs files à la fois. Ils ont été introduits par [BGWW97]. Un QDD reconnaît des mots qui sont la concaténation du contenu des N files : $w_1 \sharp w_2 \sharp \ldots \sharp w_N$. L'ordre des files est arbitraire, et n'a aucune influence sur la puissance expressive du QDD.

L'intérêt d'utiliser les automates finis pour représenter les contenus de file est que l'envoi ou la réception d'un message est une opération assez simple sur un automate fini. Nous reprendrons ces opérations, que nous détaillons sur la figure 3, pour définir la sémantique abstraite des CFSM.

Les QDD permettent d'accélérer une boucle dont les opérations concernent une seule file ; si θ est une suite d'émissions et de réceptions affectant une seule file, et L un langage représentable par un QDD, alors $\mathrm{Post}^*_\theta(L)$ est représentable par un QDD.

En revanche, quand une boucle affecte le contenu de plusieurs files, il n'est pas toujours possible de représenter l'ensemble des états accessibles après une itération d'un nombre arbitraire de fois de la boucle. La raison est qu'une telle itération peut générer un langage non-régulier, comme par exemple le langage $\{a^n \sharp b^n | n \geq 0\}$. Par contre, on peut accélérer une boucle qui ne fait pas office de compteur.

Avant l'envoi de a
Langage: $a + b + b.c$

Après l'envoi de a
Langage: $a.a + b.a + b.c.a$

Avant la réception de a
Langage: $a + b + b.c$

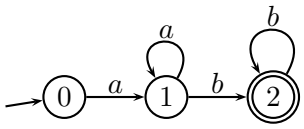
Après la réception de a
Langage: ε

Figure 3: Opérations $!a$ et $?a$ sur un QDD avec une seule file.

On ne peut guère espérer de meilleurs résultats, car on est limité aux ensembles réguliers. Pour pouvoir accélérer n'importe quelle boucle, il faut utiliser des automates capables de représenter des ensembles non-réguliers, par exemple les CQDD.

Constrained Queue-Contents Decision Diagrams (CQDD)

Un diagramme de décision des contenus de file contraint, CQDD en anglais [BH99], associe des automates "simples" et des formules de l'arithmétique de Presburger. Les variables de ces formules comptent le nombre de fois où on passe dans une boucle de l'automate. Par exemple, la figure 4 représente un CQDD reconnaissant l'ensemble non-régulier $\{a^n.b^n | n \geq 1\}$.



$$\mathcal{F} : x_{(1,a,1)} = x_{(2,b,2)}$$

Figure 4: Exemple de CQDD.

Un CQDD est en fait un ensemble fini de composantes, chaque composante étant:

- N automates simples (un par file), et
- une formule de Presburger, dont les variables sont associées aux transitions de tous ces automates.

Une configuration $(c, L_1, \ldots, L_N)$ est acceptée par un CQDD si une de ses composantes reconnaît les langages $L_1, \ldots, L_N$ en prenant les transitions des automates un nombre de fois compatible avec la formule de Presburger de cette composante.

L'avantage des CQDD est qu'ils permettent d'accélérer n'importe quelle boucle. Si un ensemble de configurations L est accepté par un CQDD, alors pour n'importe quelle boucle θ dans la structure de contrôle d'un CFSM, $\mathrm{Post}^*_\theta(L)$ est aussi accepté par un CQDD, et on peut le calculer (la preuve de ce théorème donne la construction de $\mathrm{Post}^*_\theta(L)$).

Simply Regular Expressions

Contrairement aux deux représentations précédentes, qui étaient basées sur des automates, les expressions simplement régulières [ABJ98], ou SRE, sont des expressions régulières ainsi formées :

- un atome est une expression $(a + \varepsilon)$ avec $a \in \Sigma$, ou $(a_1 + a_2 + \cdots + a_m)^*$ avec $m > 1$ et chaque a_i étant un lettre de l'alphabet ;
- une SRE est une somme de produits d'atomes.

Les SRE sont utilisées pour représenter des contenus de files d'un *lossy channel system*, c'est-à-dire d'un CFSM dans lequel les message en transit peuvent être perdus. Cette propriété justifie que l'on change de représentation, les QDD et CQDD n'étant pas adaptés à ce type de modèle. Cette représentation permet d'accélérer n'importe quelle boucle [ABJ98].

Semi-Linear Regular Expressions

Enfin, notons qu'il existe également une représentation basée sur les expressions régulières pour les CFSM classiques, sans perte de message : les expressions régulières semi-linéaires [FIS03] :

- une expression régulière linéaire est une expression régulière de la forme $x_0.y_0^*.x_1.y_1^* \ldots x_{n-1}.y_{n-1}^*.x_n$, avec x_i des mots sur l'alphabet des messages Σ et y_i des mots non vides;
- une expression régulière semi-linéaire est une somme finie d'expressions régulières linéaires.

Ces expressions permettent de représenter des langages qui sont représentables à la fois par un QDD et par un CQDD. Il s'agit donc de la représentation la moins expressive des trois, et elle ne permet pas d'accélérer n'importe quelle boucle.

Bilan des techniques d'accélération pour les systèmes FIFO

Les représentations proposées permettent donc d'accélérer n'importe quelle boucle de la structure de contrôle, que ce soit pour les CFSM ou les *lossy channel systems*. C'est un

résultat satisfaisant, mais qui ne permet toutefois pas de dépasser la limite commune à toutes les techniques d'accélérations, à savoir l'impossibilité d'obtenir un algorithme qui termine à coup sûr. Même si, dans certains cas, on peut garantir la terminaison de ces semi-algorithmes, il serait intéressant d'avoir, en plus de ces techniques d'accélération, une méthode qui termine dans tous les cas, quitte à devoir faire des approximations.

Un treillis abstrait basé sur les langages réguliers

Le treillis abstrait des langages réguliers

Nous proposons un treillis abstrait, basé sur les automates finis, pour abstraire des contenus de files. Ce treillis permet d'utiliser les analyses classiques, définies dans le cadre de l'interprétation abstraite, pour l'analyse des CFSM. On obtiendra alors un algorithme qui termine toujours, qui permet de vérifier des propriétés de sûreté mais qui procède à des sur-approximations au cours des calculs ; aussi on pourra, au mieux, déterminer qu'une propriété est vérifiée, ou répondre qu'on ne sait pas si elle est vérifiée ou non.

L'intérêt d'utiliser les automates finis est que les algorithmes codant les opérations ensemblistes (union, intersection, test d'inclusion, test du vide) sont bien connus, et que l'émission et la réception d'un message se traduisent par des opérations assez simples sur les automates, comme pour les QDD.

Cependant, ce treillis est de hauteur infinie, c'est-à-dire qu'il contient des suites infiniment croissantes. Il est alors nécessaire de définir un opérateur d'élargissement.

Opérateur d'élargissement

Le principe de cet opérateur d'élargissement est de fusionner certains états de l'automate $\mathcal{A} = (Q, \Sigma, Q_0, Q_f, \delta)$ équivalents pour la relation $\approx_k^{\text{col}}$ définie récursivement par :

- $q_1 \approx_0^{\text{col}} q_2$ si $\text{col}(q_1) = \text{col}(q_2)$, où $\text{col} : Q \to [1..N_{\text{col}}]$ est une fonction de coloriage.
- $q_1 \approx_{k+1}^{\text{col}} q_2$ si $q_1 \approx_k^{\text{col}} q_2$ et :

$$\begin{array}{l} \forall a \in \Sigma, \forall q_1' \in Q,\ q_1 \xrightarrow{a} q_1' \implies \exists q_2' \in Q :\ q_2 \xrightarrow{a} q_2' \wedge q_1' \approx_k^{\text{col}} q_2' \\ \forall a \in \Sigma, \forall q_2' \in Q,\ q_2 \xrightarrow{a} q_2' \implies \exists q_1' \in Q :\ q_1 \xrightarrow{a} q_1' \wedge q_1' \approx_k^{\text{col}} q_2' \end{array}$$

L'opération de quotient $\mathcal{A}/\approx_k^{\text{col}}$ a pour résultat l'automate $\mathcal{A}/\approx_k^{\text{col}} = \langle \Sigma, \overline{Q}, \overline{Q_0}, \overline{Q_f}, \overline{\delta} \rangle$:

- $\overline{Q} \subseteq \wp(Q)$ est l'ensemble des classes d'équivalence, notées $\bar{q} \subseteq Q$,
- $\overline{Q_0} \triangleq \{\bar{q} \mid \bar{q} \cap Q_0 \neq \emptyset\}$,
- $\overline{Q_f} \triangleq \{\bar{q} \mid \bar{q} \cap Q_f \neq \emptyset\}$,
- $\overline{\delta} = \{(\bar{q}, a, \overline{q'}) \mid (q, a, q') \in \delta\}$.

La fonction de coloriage permet de distinguer certains états. Par défaut, nous prendrons une fonction de coloriage standard qui différencie les états initiaux et finals du reste des états. Ainsi, nous avons l'assurance de conserver, lors de l'élargissement, le même ensemble de préfixes (de longueur 1) et les suffixes (de longueur k).

Cette opération sur les automates définit une opération sur les langages réguliers, en identifiant L et sa représentation canonique sous forme d'automate minimal $\mathcal{A}_L$. On définit $\rho_k^{\text{col}}(L) \triangleq \mathcal{A}_L/\approx_k^{\text{col}}$ et $L_1 \nabla_k L_2 \triangleq \rho_k(L_1 \cup L_2)$, en utilisant la fonction de coloriage standard.

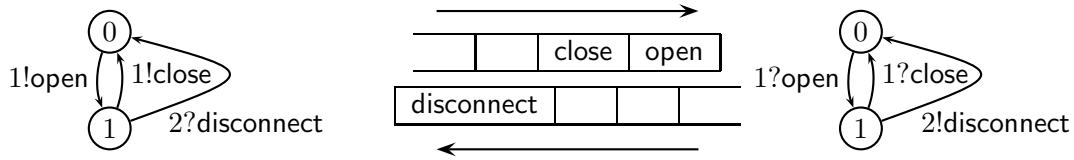
∇_k est un opérateur d'élargissement. La preuve de ce théorème repose sur le fait que le nombre d'états de l'automate $\mathcal{A}_L/\approx_k^{\text{col}}$ est borné par un nombre indépendant de l'automate $\mathcal{A}_L$. Par conséquent il n'y a qu'un nombre fini d'automates de ce type.

Analyse approchée des CFSM

Il s'agit d'une analyse "en avant". On part d'un état initial où les files sont vides, qui est représenté par le langage L_0. On attribue une valeur à k puis on calcule la suite croissante de langages définie par $L_{i+1} = L_i \nabla_k \text{Post}(L_i)$, jusqu'à convergence de cette suite. On obtient alors une sur-approximation de l'ensemble d'atteignabilité.

S'il n'y a qu'une seule file, il suffit d'utiliser les opérateurs définis précédemment. En revanche, lorsque le CFSM a N files, nous avons le choix entre utiliser N automates pour représenter séparément le contenu de chaque file, ou d'utiliser un seul automate, style QDD, pour représenter le contenu de toutes les files. Si on utilise un QDD, on veillera à identifier les parties de l'automate qui reconnaissent chaque file et à ne pas fusionner des états qui appartiennent à des parties différentes, lorsque l'on fait l'opération d'élargissement. Il suffit pour cela de modifier la fonction de coloriage standard, et de veiller à ce que deux états de deux parties différentes n'aient pas la même couleur. À part cette modification, la définition de l'opérateur d'élargissement est identique à celle des CFSM avec une seule file.

Dans le premier cas, on parlera d'analyse *non-relationnelle*. Dans le second cas, l'analyse sera dite *relationnelle* car elle permet de conserver certaines relations entre les contenus des différentes files. L'analyse relationnelle est plus précise, mais plus coûteuse, que l'analyse relationnelle, comme l'illustre l'exemple du protocole de connexion/déconnexion représenté sur la figure 5.



(a) Client (b) Files (c) Serveur

Figure 5: Le protocole de connexion/déconnexion.

Les résultats de l'analyse d'accessibilité, avec $k = 2$, sont donnés ici :

analyse non-relationnelle

$0, 0$	$o^* + o^*.c.(o^+.c)^*.o^*$	d^*
$1, 0$	$o^* + o^*.c.(o^+.c)^*.o^+$	d^*
$0, 1$	$o^* + o^*.c.(o^+.c)^*.o^*$	d^*
$1, 1$	$o^+ + o^*.c.(o^+.c)^*.o^+$	d^*

analyse relationnelle

$0, 0$	$(o.c)^*\#\varepsilon$
	$+ \quad (o.c)^*\#\varepsilon$
	$+ \quad (c.o)^*.o.c.(o.c)^*\#\varepsilon$
	$+ \quad c.(o.c)^*\#d$
$1, 0$	$\varepsilon\#d$
	$+ \quad o\#\varepsilon$
	$+ \quad (o.c)^*.o\#\varepsilon$
	$+ \quad (c.o)^*\#d$
	$+ \quad (c.o)^*.o.(c.o)^*\#\varepsilon$
$0, 1$	$c.(o.c)^*\#\varepsilon$
$1, 1$	$\varepsilon\#\varepsilon$
	$+ \quad (c.o)^*\#\varepsilon$

Dans le cadre de l'analyse relationnelle, on peut montrer que la seconde file contient au plus un message d, ce que l'analyse non-relationnelle ne détecte pas.

Bilan de l'analyse des CFSM par interprétation abstraite

Le treillis abstraits des langages réguliers permet d'obtenir des analyses d'accessibilité relativement précises, qui sont même exactes sur les quelques exemples de protocoles de communication que nous avons examinés. Et, contrairement aux semi-algorithmes d'accélération, nous pouvons garantir la terminaison de notre algorithme.

Nous aimerions effectuer un travail similaire pour un modèle plus complexe que celui des CFSM, dans lequel les automates peuvent contenir des variables et les messages peuvent porter des valeurs. Mais pour un tel modèle, les langages réguliers ne suffisent plus, aussi nous utiliserons des automates plus complexes, les automates de treillis.

Les automates de treillis

Les automates de treillis sont conçus pour manipuler facilement des langages définis sur un alphabet infini.

Définition et discussion

Les automates de treillis sont des automates finis dont les transitions sont étiquetées par les éléments d'un treillis atomique et non d'un alphabet fini. Ce treillis est supposé avoir un nombre infini d'atomes, qui constituent alors les lettres d'un alphabet infini.

Par exemple, l'automate de treillis représenté sur la figure 6 reconnaît les séquences de nombres $x_0 \dots x_{n-1}$ telles que :

- n est impair,

- $x_{2i} \in [0, 1]$,
- $x_{2i+1} \in [1, 2]$.

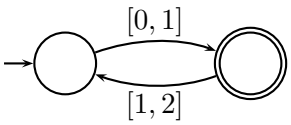


Figure 6: Exemple d'automate de treillis.

La définition simple des automates de treillis pose toutefois plusieurs problèmes. En particulier, on peut avoir deux automates de treillis équivalents en terme de langage mais dont les transitions sont incomparables. Par exemple, la figure 7 montre deux automates qui reconnaissent le même ensemble de mots à une lettre[2] $L = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(3,0),(3,1)\}$, mais qui sont incomparables: les transitions ne sont ni égales, ni incluses les unes dans les autres.

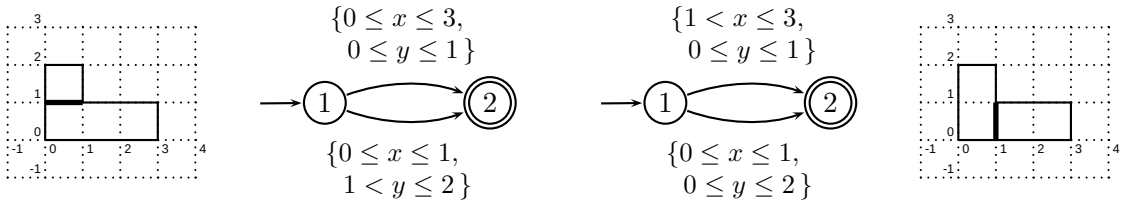


Figure 7: Deux automates de treillis équivalents mais incomparables.

Pour faciliter les opérations sur les automates de treillis, on introduit une partition finie de l'ensemble des atomes, donnée par une certaine fonction $\pi : \Sigma \to \Lambda$, où Σ est un ensemble fini et Λ notre treillis atomique. Les transitions seront alors fusionnées pour ne garder qu'une seule transition par classe d'équivalence. Cette fusion des transitions engendre une sur-approximation du langage reconnu. Par exemple, la figure 8 illustre l'automate obtenu quand on fusionne les transitions d'un des deux automates précédents. Le treillis utilisé est celui des polyèdres convexes [CH78].

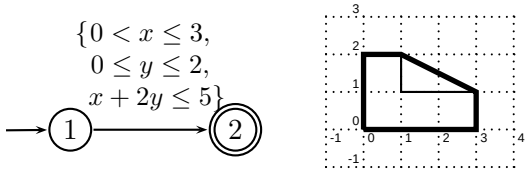
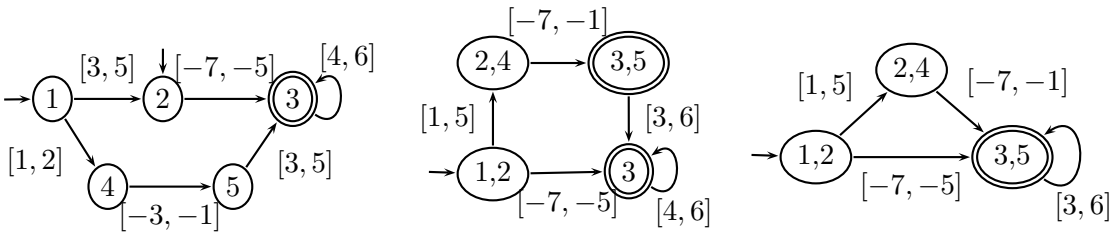


Figure 8: Automate de treillis fusionné selon la partition triviale $\sigma_0 \to \mathbb{R}^2$.

Cette notion d'*automates de treillis partitionnés* (PLA) permet aussi de définir la "forme" d'un automate $\mathcal{A}$, shape($\mathcal{A}$), qui est l'automate fini obtenu en remplaçant chaque transition par la classe d'équivalence à laquelle elle appartient. Cette forme

[2]Ici, les atomes sont les couples d'entiers.



(a) Automate de départ

(b) Automate déterministe

(c) Automate minimal

Figure 9: Déterminisation et minimisation d'un automate de treillis avec la partition $\{]-\infty, 0], [0, +\infty[\}$.

servira aux opérations de déterminisation et de minimisation, ainsi qu'à la définition d'un opérateur d'élargissement.

Opérations de déterminisation et de minimisation

L'algorithme de déterminisation des automates de treillis est semblable à celui des automates finis : on considère les ensembles d'états de l'automate de départ et, à chaque étape, on regarde pour chaque classe d'équivalence $\sigma \in \Sigma$ l'ensemble des états pouvant être atteints par une transition étiquetée par un élément de cette classe d'équivalence. La seule différence est que l'on fusionne au cours de ce processus, les transitions appartenant à la même classe d'équivalence. L'algorithme de minimisation suit le même principe : on minimise la forme de l'automate de treillis, en fusionnant au besoin les transitions. La figure 9 donne un exemple d'application de ces deux algorithmes.

Ces algorithmes engendrent des sur-approximations : les automates $\det(\mathcal{A})$ et $\min(\mathcal{A})$ reconnaissent des langages plus grands que $\mathcal{A}$. Cependant, ces sur-approximations sont optimales au sens où :

> Pour tout automate fusionné et déterministe $\mathcal{A}'$ basé sur la même partition que $\mathcal{A}$, $\mathcal{A} \sqsubseteq \mathcal{A}' \implies \det(\mathcal{A}) \sqsubseteq \mathcal{A}'$.

On a un résultat similaire pour la minimisation. Pour tout PLA $\mathcal{A}$, il y a un unique automate minimal $\widehat{\mathcal{A}}$ basé sur la même partition π tel que :

1. $\mathcal{A} \sqsubseteq \widehat{\mathcal{A}}$,
2. pour tout automate minimal $\mathcal{A}'$ basé sur la même partition π, $\mathcal{A} \sqsubseteq \mathcal{A}' \implies \widehat{\mathcal{A}} \sqsubseteq \mathcal{A}'$.

Nous avons alors une représentation canonique des langages reconnus par automates de treillis, sous forme d'automates minimaux, appelés *Normalized Lattice Automata* (NLA). Nous allons utiliser cette représentation pour définir un opérateur d'élargissement.

Opérateur d'élargissement

Pour définir un opérateur d'élargissement sur les automates de treillis, nous supposons que le treillis Λ est muni d'un opérateur d'élargissement ∇_Λ. On va alors combiner cet opérateur avec l'opérateur ρ_k étendu aux automates de treillis.

Cet opérateur ρ_k consiste, comme pour les automates finis, à fusionner les états équivalents pour la relation $\approx_k$. La relation $\approx_k$ est calculée sur la forme de l'automate de treillis, et lors du quotient, on fusionne les transitions selon la partition $\pi : \Sigma \to \Lambda$.

Enfin, si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont deux NLA définis sur la même partition, avec $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, on définit l'opérateur d'élargissement ∇_k ainsi :

$$\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \begin{cases} \widehat{\rho_k(\mathcal{A}_2)} & \text{si } \mathrm{shape}(\mathcal{A}_1) \neq \mathrm{shape}(\rho_k(\mathcal{A}_2)) \\ \mathcal{A}_1 \nearrow \mathcal{A}_2 & \text{sinon (ce qui signifie } \mathrm{shape}(\mathcal{A}_1) = \mathrm{shape}(\mathcal{A}_2)) \end{cases}$$

où $\mathcal{A}_1 \nearrow \mathcal{A}_2$ est le NLA $\mathcal{A}$ qui a le même ensemble d'états que $\mathcal{A}_1$ et $\mathcal{A}_2$, et des transitions définies par :

$$\frac{\sigma \in \Sigma \quad (q, \lambda_1, q') \in \delta_1 \quad (q, \lambda_2, q') \in \delta_2 \quad \lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)}{(q, (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma), q') \in \delta}$$

Bilan des automates de treillis

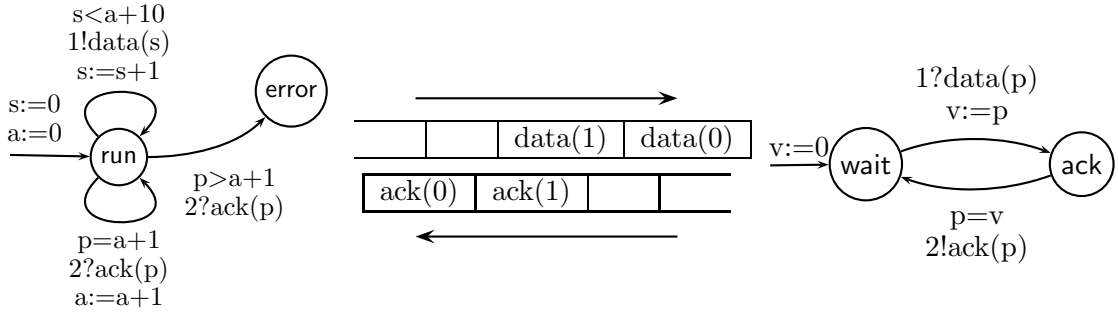
Les automates de treillis permettent de représenter assez simplement des langages réguliers sur les alphabets infinis. Ils étendent des abstractions, par exemples une abstraction des nombres entiers, à des séquences finies de nombres entiers. Nous pouvons ainsi vérifier des systèmes communiquant par files où les messages, dans les files, portent des valeurs que l'on sait abstraire.

Vérification des *Symbolic Communicating Machines* et analyse inter-procédurale

Vérification des *Symbolic Communicating Machines*

Le modèle des automates communicants symboliques (SCM) est une extension du modèle des CFSM ; les automates peuvent avoir un nombre fini de variables, et les messages, dans les files, sont aussi paramétrés. C'est un modèle assez proche de ceux proposés dans la littérature, comme par exemple [HSS+93, LRMS96]. La figure 10 est une modélisation d'un protocole de fenêtre glissante, exprimé dans le formalisme des SCM.

Dans la suite, on supposera que le SCM a n variables et p paramètres, que l'on suppose tous de type identique. Cette supposition sert à simplifier un peu l'écriture de la sémantique et ne restreint en rien la généralité de l'approche. On va utiliser le treillis des polyèdres convexes (ou tout autre treillis abstrait adapté aux domaines des variables), avec la notation $V^{(n)} = \mathsf{Pol}(\mathbb{Q}^n)$. Une valeur abstraite sera un couple formé d'un polyèdre de dimension n, pour les valeurs des variables, et d'un automate



(a) Émetteur (b) Files (c) Receveur

Figure 10: Protocole à fenêtre glissante simplifié

de treillis construit sur le treillis $V^{(p)} = \mathsf{Pol}(\mathbb{Q}^p)$, représentant le contenu possible des files.

La sémantique abstraite des SCM sera donnée naturellement, pour une transition, par la suite d'opérations suivante :

1. On fait l'intersection de la garde avec la valeur abstraite courante, obtenant ainsi les valeurs possibles des paramètres ;

2. On modifie alors l'automate de treillis représentant le contenu des files, de la même manière que pour les automates finis. On conserve ainsi, dans les files, les valeurs des paramètres ;

3. On répercute l'effet des affectations sur le polyèdre qui abstrait les valeurs des variables.

Cette sémantique "standard" a cependant un défaut : elle ne conserve aucun lien entre la valeur des paramètres (abstraites dans l'automate de treillis) et la valeur des variables. Ce qui donne, en pratique, des résultats assez mauvais quand il s'agit d'analyser des protocoles où la valeur des variables dépend fortement des valeurs passées dans les files.

Pour pallier à ce défaut, nous définissons une sémantique "non-standard" dans laquelle on conservera, dans l'automate de treillis, la valeur des variables et la valeurs des paramètres. Autrement dit, on aura un automate bâti sur le treillis $V^{(n+p)}$ au lieu du treillis $V^{(p)}$.

Avec une telle sémantique, il faut veiller à ce que les valeurs des variables, dans les files, correspondent bien aux affectations que l'on fait. Aussi la sémantique d'une affectation modifiera non seulement le polyèdre "à l'extérieur" de l'automate de treillis, mais aussi chaque transition de cet automate. Les opérations abstraites sont donc plus coûteuses, mais cela en vaut la peine, car cette sémantique permet d'obtenir de bien meilleures approximations.

Analyse inter-procédurale

L'analyse des SCM était la première motivation de notre travail sur les automates de treillis. Toutefois, nous estimons que ce type d'automates est aussi adapté à l'analyse inter-procédurale, où on tient compte non seulement du point de retour, mais aussi des valeurs des variables locales (paramètres) passées dans la pile lors de l'appel à des procédures.

Plus formellement, un programme est défini comme un ensemble de procédures ; chaque procédure P_i a un ensemble de variables locales $LVar_i$, un ensemble de paramètres d'appel $\vec{fp}_i \subseteq LVar_i$ et un ensemble de paramètres de retour $\vec{fr}_i \subseteq LVar_i$. Lors de l'appel d'une procédure, on empilera un registre d'activation $\langle k, e\rangle \in Act = K \times LEnv$ qui contient l'adresse de retour $k \in K$, et un environnement attribuant une valeur aux variables locales passées en paramètre. La pile sera donc un mot sur l'alphabet infini des registres d'activation.

Si l'on a une abstraction des environnements $\wp(LEnv) \leftrightarrows \Lambda$, on peut abstraire un ensemble de valeurs possibles de la pile d'appel par un automate de treillis, selon l'abstraction :

$$\wp(Act^+) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} K \to \Lambda \times \mathrm{Reg}(K \times \Lambda)$$

Pour le choix de la partition de $K \times \Lambda$, on peut prendre tout naturellement $\pi(k) = \{k\} \times \top_\Lambda$.

La sémantique abstraite se définit alors classiquement en empilant ou dépilant la valeur des registres d'activation. Ainsi, si on a une valeur abstraite $(Y, F) \in \Lambda \times \mathrm{Reg}(K \times \Lambda)$, après un appel au point de contrôle c, on aura une nouvelle valeur abstraite $(Y', (\{c\}, Y) \cdot F)$.

Avec cette sémantique, on peut faire des analyses en avant ou des analyses en arrière, grâce à un calcul de point-fixe. Contrairement à l'analyse des SCM, on n'a pas besoin ici d'une sémantique non-standard ; en effet, on utilise à la place l'égalité des valeurs entre paramètres formels et paramètres effectifs.

Bilan de la vérification des SCM et de l'analyse inter-procédurale

Une fois que l'on a défini les automates de treillis, il est assez facile de les employer pour vérifier des systèmes à files ou à pile avec des alphabets infinis. Pour les systèmes à files, à notre connaissance, il n'y a pas d'autres méthodes capables de traiter ce genre de systèmes avec des approximations comparables. Pour les systèmes avec piles d'appels, la possibilité de jouer avec les paramètres de l'opérateur d'élargissement en fait un outils très souple et capable de faire des analyses précises.

Un analyseur a été programmé pour faire de telles analyses, et est détaillé au chapitre 6. Il emploie une bibliothèque d'automates de treillis, programmée en Objective CAML.

Conclusion

Dans cette thèse, nous avons proposé des treillis abstraits, basés sur des langages réguliers, pour vérifier des systèmes communiquant par files.

Nous avons d'abord considéré les CFSM, qui ont un alphabet fini de messages, et nous avons utilisé un treillis basé sur les automates finis. L'analyse par interprétation abstraite des CFSM offre alors une alternative intéressante aux techniques d'accélération.

Puis nous avons étendu ces travaux à l'analyse des SCM, en utilisant des automates de treillis pour représenter les contenus de file, qui ont alors un alphabet infini de messages. Ces automates de treillis se sont également révélés utiles pour l'analyse inter-procédurale.

Nos travaux en cours portent sur :

- l'amélioration de l'analyseur, pour le rendre plus simple d'emploi et faire des analyses plus puissantes ;
- la synthèse de contrôleurs pour des systèmes à files, en utilisant les abstractions définies dans la thèse ;
- et l'utilisation des outils développés dans un cadre autre que l'interprétation abstraite. Par exemple, on peut réutiliser l'opérateur ρ_k défini pour les langages réguliers pour faire du raffinement d'abstractions.

Introduction

How can we trust a machine ? This simple question is, for several reasons, an important issue of fundamental computer science. The first reason is that nowadays, computers and other electronic devices are omnipresent: they are present, sometimes hidden, in cars, planes, trains, credit cards, mobile phones, etc. Our banking system, our telecommunication networks, our society relies on computers, on softwares, on machines so complex that they can only be conceived by teams of engineers.

The second reason is that those machines are sensitive to the smallest mistake. Unlike the humans, who can understand their mistakes and correct them, a computer does not think, does not wonder whether what it is currently doing is right or wrong. Since the cost of a single bug can be greater than 100 millions dollars, like the one that caused the explosion of the fist Ariane 5 rocket, one cannot tolerate the slightest bug in a critical system.

The third reason is that human beings cannot check a software without the help of an automatic tool, since a software may contain thousands or millions of instructions, each one being a possible cause of a bug. This justifies the development of some tools, which can check automatically all the instructions of a software.

When conceiving automatic verification tools, one is confronted to a theoretical limitation found by Alan Turing. He imagined a programmable machine, later called a *Turing machine*, and demonstrated that we cannot write a program determining if any other program terminates. The consequences of this theorem is that there is no hope of finding a program that can automatically verify a *Turing-powerful system*, *i.e.* a system that can do the same functions as a Turing machine. Examples of Turing-powerful systems are the computers, if we consider they have an infinite memory.

So, computer scientists and engineers have developed some automatic or semi-automatic techniques ensuring that the considered system behaves as expected. The first method is to perform a lot of tests on the system. The tests may be either manually or automatically generated. Testing is useful at detecting bugs, but cannot give a proof of correctness of the system. This proof is given by a formal verification of the system, like *model checking*.

In the early eighties, Edmund Clarke, Allen Emerson and Joseph Sifakis introduced model checking, and recently won the Turing Award for this work. The model checking framework provides the algorithmic means to determine whether an abstract model, *i.e.* a formal description of the system, satisfies a specification, *i.e.* a formal expression of the expected behavior of the system (*cf.* Section 1.2). It relies on an exhaustive

exploration of the state space of the model, and its result is the guarantee that the model satisfies its specification, or a counter-example. The principal limitation of this method is its complexity, leading to the *state explosion problem* on larger systems. The model checking techniques were improved by some efficient symbolic representation of the set of states, like BDDs [MO83]. But even with those improvements, the verification of infinite systems is only possible if one impose restrictions:

- on the model (*e.g.* we restrict ourselves to finite systems, or to well-structured systems), or
- on the kind of properties we can check.

Another approach to the verification of infinite systems is the one based on the *abstract interpretation* framework [CC77a]. The principle of abstract interpretation is to consider an abstraction of the state space, called an abstract domain, which has a lattice structure. In this abstract domain, we can easily obtain an over-approximation of the behavior of the system. A *widening operator* (*cf.* Section 1.4) ensures that this computation terminates, usually in only a few computation steps.

Compared to model checkers, the analyzers based on abstract interpretation are thus generally less costly, but can only answer "yes" or "I don't know" to the question "does this system satisfy this property?". This is due to the over-approximations made by the static analyzers (*cf.* Section 1.4). Static analyzers are generally employed to optimize compilers or to certify softwares.

The abstract interpretation theory is the framework of this thesis. But instead of verifying classical softwares, we aim at the verification of communication protocols and other asynchronous systems. Communication protocols are employed by the large-scale communication networks, like the Arpanet/Internet. [BZ83] introduced the model of Communicating Finite-State Machines (CFSMs) to model this kind of protocols.

This model, among other ones like Petri nets [NPW81], FIFO nets [FM82], or Message Sequence Charts (MSCs) [ITU99], is a formal description of the behavior of two (or more) machines communicating through unbounded FIFO queues. However, [BZ83] also proved that this model is Turing-powerful if there are at least two queues, which implies that most verification problems are undecidable when applied to a CFSM.

Despite this theoretical limitation, some researchers proposed some semi-algorithms for the verification of safety properties of FIFO channel systems. Those algorithms rely on the principle of *loop acceleration*: they first compute in what configuration we are after taking a given loop θ an arbitrary number of times. If this computation is possible for any loop of the transition system, then we have an efficient semi-algorithm aiming at model checking the FIFO channel system. The acceleration techniques for FIFO channel systems are detailed in Chapter 2.

However, this method can only give exact semi-algorithms, *i.e.* algorithms which may not terminate, but which give the exact result if they eventually terminate. One purpose of this thesis was to find an algorithm that always terminates but may only return an over-approximation of the actual result (*cf.* Chapter 3). This method is based on the abstract interpretation framework.

The main idea of this method is to represent all possible contents of the queues by regular languages. The actual symbolic representation of the queue content is a finite automaton. With this representation, the operations involving the queues are quite easy: sending a message in a queue, or receiving a message from the queue is simply adding or removing some states and/or transitions in the automaton representing the content of this queue. So, the collecting semantics (where one reasons in terms of set of configurations) of a CFSM is defined by a small set of automata operations (*cf.* Chapter 3).

The originality of this method is to define a widening operator adapted to the regular languages representing the queue contents. The principle of this operator is to merge states of the automaton that are equivalent with respect to the k-depth bisimulation relation.

This approximate reachability analysis gave good results when we tested it on some examples, but we want to tackle more challenging protocols. The CFSM processes admit only a finite control structure, whereas real protocols often have integer/real variables and timers. So we looked at an extension of the CFSM model: the Symbolic Communicating Machines (SCMs) model.

An algorithm aiming at the verification of SCMs has to deal not only with the same issues as the ones for CFSMs, but also with the following particularity of the SCM model. While the alphabet of messages of a CFSM is finite, the messages of a SCM can carry the value of an integer parameter (or other types of parameters). The content of the queues is therefore a finite word over the infinite domain of the parameters $\mathcal{D}_P$.

The classical approach to words over an infinite alphabet is to restrict the kind of operations on the infinite part (*e.g.* the only operation allowed is to test whether two values are equal) and to define operators (like "next letter" or "previous letter") so that the logic is decidable [NSV01].

We choose a different approach. Since there are well-known lattices to abstract $\mathcal{D}_P$, like the lattice of intervals or the lattice of convex polyhedra, we wonder whether we can combine those classical abstract lattices with the regular languages, so that we have an abstraction of $\mathcal{L}(\mathcal{D}_P) = 2^{(\mathcal{D}_P^*)}$ by a new kind of regular languages:

$$\begin{array}{ccc} \mathcal{D}_P & \xleftarrow{\gamma} & \Lambda \\ & \Downarrow & \\ \mathcal{L}(\mathcal{D}_P) & \xleftarrow{\Gamma} & \text{new kind of regular languages ?} \end{array}$$

We introduce the *lattice automata* to define this new abstract domain. A lattice automaton is like a finite automaton, except that the transitions are labeled by elements of a lattice Λ instead of letters of a finite alphabet Σ. The operations on lattice automata are similar to the ones on finite automata (*cf.* Chapter 4). Some classical operations, like determinization and minimization, are not always feasible, depending on the lattice Λ. If Λ is an atomic lattice (*cf.* Section 1.3), we can define "determinization" and "minimization" algorithms, using a partition of the atoms of Λ. Using the minimization algorithm, we can normalize any lattice automaton $\mathcal{A}$, and give a canon-

ical representation of the language recognized by $\mathcal{A}$. The actual abstract lattice we consider is the lattice of *normalized lattice automata* (NLAs).

The next step is to define a proper widening operator, which is mandatory since the lattice of NLAs is of infinite height (*cf.* Section 1.4). This widening operator is presented in Chapter 4. Its principle is to combine the widening operator defined in Chapter 3, and the widening operator of the lattice Λ.

We then define an approximated reachability analysis on some SCMs, employing this newly defined widening operator. The results are presented in Chapter 5. The experiments showed that with the standard abstract semantics of SCMs, the analysis was too imprecise. This is the reason why we defined a more sophisticated abstract semantics, where the abstract values, in the queues, are approximations of the values of the variables and of the parameters.

Chapter 5 also presents an inter-procedural analysis based on lattice automata. "Inter-procedural analysis" means that we can verify properties on programs with procedures calls. When a procedure is called, the return address[3] and the local variables of this procedure are pushed onto the stack. We can thus represent the content of this call stack by a finite word over the infinite alphabet $K \times LEnv$, where K is the set of control points of the program (the return address) and $LEnv$ is the set of environments, which give the values of the local variables. We then abstract words representing stack contents in the same way as what we did for words representing queue contents.

Finally, Chapter 6 presents an implementation of lattice automata and of the analyzer employed for the experiments of Chapter 5. The lattice automata library and the analyzer are written in the Objective CAML language.

[3]The return address is the control point of the program that comes just after the procedure call.

Chapter 1

Preliminaries

This chapter presents the key notions needed for the understanding of the sequel of the thesis.

We first present the model of *transition systems* and the model of *Communicating Finite-State Machines* (CFSMs). Transition systems are employed in all fields of fundamental computer science, whereas CFSMs are a useful model for communication protocols and asynchronous systems. CFSMs will be employed in Chapters 2 and 3, and an extension of this model, the Symbolic Communicating Machines, in Chapter 5.

We then explain, with some examples, how one can verify whether a transition system satisfies a *safety property*. In the following chapters, we only deal with safety properties.

We remind the definition of partial orders and lattices, and introduce the *abstract interpretation* framework. Abstract interpretation is the main framework of the methods developed in this thesis.

Finally, we briefly talk about regular languages and finite automata. Finite automata, as a canonical representation of regular languages, are employed in Chapters 2 and 3. We also define a new kind of finite-state automata in Chapter 4.

1.1 Transition Systems and Communicating Finite-State Machines

1.1.1 Transition Systems

Definition. The model of *transition system* is widely employed by computer scientists. It assumes that the modeled system has some identifiable states (or *configurations*) and can go from one configuration to another according to a transition rule. This discrete model[1] gives a mathematical description of the behavior of automata, robots and other machines, and computer programs. For example, we can model the behavior of a lamp with two configurations (*on* and *off*) and two transitions (*off* $\xrightarrow{switch\ on}$ *on*

[1] The word "discrete" means that the systems has not a continuous evolution but "jumps" from one configuration to another.

and $on \xrightarrow{switch\ off} off$). Figure 1.1 depicts a transition system modeling the behavior of the lamp.

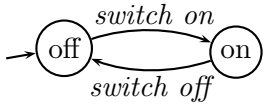


Figure 1.1: Modeling of a lamp

In the lamp example, transitions are labeled by *events*: it is a *discrete event system*, because the system changes when a discrete event occurs. The following definition formalizes this notion.

Definition 1.1 (Transition system) *A discrete transition system is a tuple* $\langle C, C_0, \rightarrow \rangle$ *where:*

- C *is a countable set of configurations,*
- $C_0 \subseteq C$ *is a set of initial configurations,*
- $\rightarrow \subseteq C \times C$ *is a transition relation.*

A labeled transition system (or discrete event system) is a transition system $\langle C, \Sigma, c_0, \rightarrow \rangle$ *with a set of labels* Σ *and a transition relation* $\rightarrow \subseteq C \times \Sigma \times C$.

A (labeled) *run* $c_0 \overset{(\sigma_1)}{\rightarrow} c_1 \overset{(\sigma_2)}{\rightarrow} c_2 \overset{(\sigma_3)}{\rightarrow} \ldots \overset{(\sigma_{n-1})}{\rightarrow} c_{n-1} \overset{(\sigma_n)}{\rightarrow} c_n$ starts in an initial configuration $c_0 \in C_0$ and respects the transition relation: $\forall i,\ (c_i, c_{i+1}) \in \rightarrow$, or $(c_i, \sigma_{i+1}, c_{i+1}) \in \rightarrow$ if the transition system is a labeled one. For a labeled transition system, its *traces* are the words $w = \sigma_1.\sigma_2 \ldots \sigma_n$, where $\sigma_1, \sigma_2, \ldots, \sigma_n$ are labels of a run $c_0 \overset{\sigma_1}{\rightarrow} c_1 \overset{\sigma_2}{\rightarrow} c_2 \overset{\sigma_3}{\rightarrow} \ldots \overset{\sigma_{n-1}}{\rightarrow} c_{n-1} \overset{\sigma_n}{\rightarrow} c_n$. A run may be either finite or infinite.

This model is quite general and can be used to define the semantics of some high-level models such as the next model we present, the Communicating Finite State Machines. Before defining this model, we introduce the notion of *reachability*.

Reachability. Let us consider a set of configurations $S \subseteq C$. The functions Post : $2^C \rightarrow 2^C$ and Pre : $2^C \rightarrow 2^C$ are defined by the transition relation as:

$$\begin{aligned} \text{Post}(S) &= \{c \in C | \exists c' \in S, c' \rightarrow c\} \\ \text{Pre}(S) &= \{c \in C | \exists c' \in S, c \rightarrow c'\} \end{aligned}$$

The configurations in Post(S) are the immediate successors of the configurations S, while the configurations in Pre(S) are the immediate predecessors of the configurations S. The sets of all successors and of all predecessors are given by the reflexive and transitive closure of $\rightarrow$, $\rightarrow^*$, which is defined recursively by:

1. $\forall c \in C, (c, c) \in \rightarrow^*$,
2. $(c_1, c_2) \in \rightarrow^*$ if and only if $\exists c_1' \in C, (c_1, c_1') \in \rightarrow \wedge (c_1', c_2) \in \rightarrow^*$.

Definition 1.2 (Reachability) *Let c and c' be two configurations of the transition system $\langle C, C_0, \rightarrow\rangle$. s' is* reachable *from s (resp.* co-reachable *from s) if $s \rightarrow^* s'$ (resp. $s' \rightarrow^* s$). The set $\mathrm{Post}^*(S)$ (resp. $\mathrm{Pre}^*(S)$) is thus the set of configurations that are reachable (resp. co-reachable) from S. In other words:*

$$\begin{aligned} \mathrm{Post}^*(S) &= \{c \in C | \exists c' \in S, c' \rightarrow^* c\} \\ \mathrm{Pre}^*(S) &= \{c \in C | \exists c' \in S, c \rightarrow^* c'\} \end{aligned}$$

The reachability set *of the transition system is defined as $\mathrm{Post}^*(C_0)$.*

Remark 1.1 $\mathrm{Post}^*(S)$ *is also the least fix-point of the function:*

$$\begin{aligned} \mathcal{F}: \ 2^C &\rightarrow 2^C \\ X &\mapsto \mathrm{Post}(X) \cup S \end{aligned}$$

One of the main objectives of this thesis is to solve this fix-point equation, when the transition system represents the semantics of a Communicating Finite State Machine.

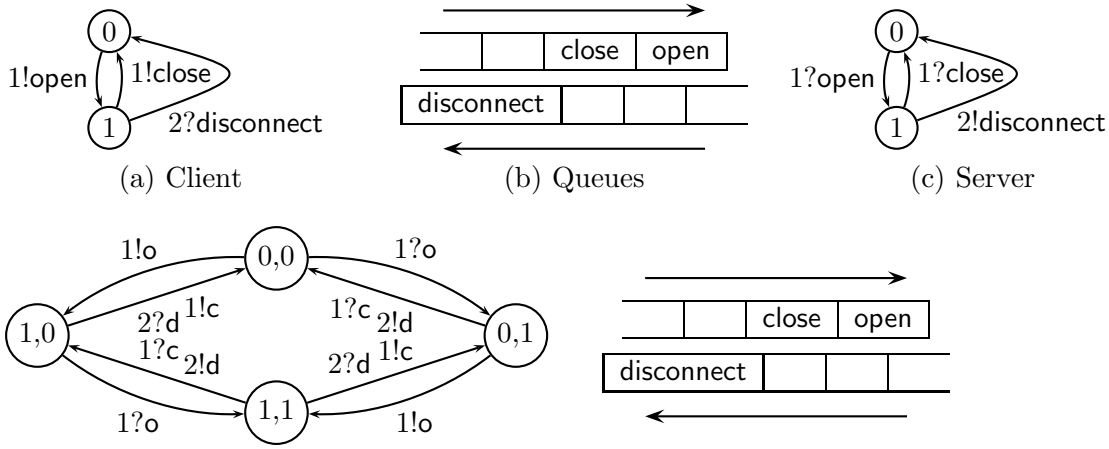
1.1.2 Communicating Finite-State Machines

The model of *Communicating Finite-State Machines* (CFSMs) is a quite simple model to describe distributed systems exchanging messages over an asynchronous network. This model consists in finite-state processes that exchange messages via N unbounded FIFO queues. For example, a communication protocol describing the communication between two computers can be modeled by a CFSM: the behavior of each computer is modeled by a finite-state process and the FIFO queues model the communication channels as well as the different buffers.

Formal definition. There are several definitions of this model in the literature. Some consider explicitly the different processes, whereas others use a global control structure. There are also some internal actions, *i.e.* transitions that are neither an input or an output. We present here a definition of the CFSM model with a global control structure and no internal actions, but the examples will detail explicitly the different processes.

Definition 1.3 *A Communicating Finite-State Machine with N channels is given by a tuple $\mathcal{M} = \langle C, \Sigma, c_0, \Delta\rangle$ where:*

- *C is a finite set of locations (control states);*
- *$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_N$ is a finite alphabet of messages, where Σ_i denotes the alphabet of messages that can be stored in queue i;*
- *$c_0 \in C$ is the initial location;*
- *$\Delta \subseteq C \times A \times C$ is a finite set of transitions, where $A = \bigcup_i \{i\} \times \{!, ?\} \times \Sigma_i$ is the set of actions. An action can be either:*
  - *an output $i!m$: "the message m is sent to the queue i", or*



(a) Client (b) Queues (c) Server

(d) Global CFSM: product of client and server processes

Figure 1.2: The connection/disconnection protocol

– *an input $i?m$: "the message m is received from the queue i".*

The control structure (C, c_0, Δ) of Definition 1.3 corresponds to an asynchronous product of some finite-state machines. Figure 1.2 depicts an example of CFSM, with graphical representation of the two processes and a representation with a single global control structure.

Example 1.1 *The connection/disconnection protocol [JR87] between two machines is the following (Figure 1.2): the client can open a session by sending the message* open *to the server. Once a session is opened, the client may close it on its own by sending the message* close *or on the demand of the server if it receives the message* disconnect*. The server can read the request messages* open *and* close*, and ask for a session closure.*

Semantics. The semantics of a CFSM (C, Σ, c_0, Δ) is given as a labeled transition system (LTS) $\langle Q, Q_0, A, \rightarrow \rangle$ where:

- $Q = C \times \Sigma_1^* \times \cdots \times \Sigma_N^*$ is the set of states;
- $Q_0 = \{\langle c_0, \varepsilon, \ldots, \varepsilon \rangle\}$ is the set of initial states;
- A is the alphabet of actions (*cf.* Definition 1.3).
- $\rightarrow$ is defined by the two rules:

$$\frac{(c_1, i!m, c_2) \in \Delta \qquad w_i' = w_i.m}{\langle c_1, w_1, \ldots, w_i, \ldots, w_N \rangle \xrightarrow{i!m} \langle c_2, w_1, \ldots, w_i', \ldots, w_N \rangle}$$

$$\frac{(c_1, i?m, c_2) \in \Delta \qquad w_i = m.w_i'}{\langle c_1, w_1, \ldots, w_i, \ldots, w_N \rangle \xrightarrow{i?m} \langle c_2, w_1, \ldots, w_i', \ldots, w_N \rangle}$$

A global configuration of a CFSM is thus a tuple $\langle c, w_1, \ldots, w_N \rangle \in C \times \Sigma_1^* \times \cdots \times \Sigma_N^*$ where c is the current location and w_i is a finite word on Σ_i representing the content of queue i. At the beginning, all queues are empty, so the *initial configuration* is $\langle c_0, \varepsilon, \ldots, \varepsilon \rangle$.

A configuration $\langle c, w_1, \ldots, w_N \rangle$ is thus reachable if $\langle c_0, \varepsilon, \ldots, \varepsilon \rangle \rightarrow^* \langle c, w_1, \ldots, w_N \rangle$. The reachability set is defined as usual. Computing this set is the purpose of the reachability analysis and the core of the verification of safety properties.

1.2 Verification of Safety Properties

The purpose of this section is not to give an exhaustive survey of verification techniques, but to explain, as simply as possible, how one can check whether a transition systems satisfies a *safety property*. A safety property is, intuitively, a property ensuring there is no critical failure.

We first discuss the verification of *invariance properties*, *i.e.* properties defined by a set of configurations. We then explain how to verify safety properties given by a finite automaton called an *observer*. Finally, we mention other kinds of properties, expressed in a temporal logic. In later chapters, we will focus on the verification of safety properties.

1.2.1 Invariance Properties

Let us consider a transition system $\langle C, C_0, \rightarrow \rangle$ and a partition of the configurations $C = C_{good} \uplus C_{bad}$. The system satisfies the invariance property if all runs stay in the set of "good" configurations C_{good}. Note that this property is given by a set of states instead of a set of allowed runs. The allowed runs are simply the runs of the transition system $\langle C_{good}, C_0 \cap C_{good}, \rightarrow_{/C_{good} \times C_{good}} \rangle$.

Indeed, the verification of an invariance property consists in computing the reachability set $\mathrm{Post}^*(C_0)$ and in checking one of the two equivalent conditions:

1. all the reachable configurations are "good": $\mathrm{Post}^*(C_0) \subseteq C_{good}$, or,
2. none of the "bad" configurations is reachable: $\mathrm{Post}^*(C_0) \cap C_{bad} = \emptyset$.

There is also a choice between a *forward analysis* and a *backward analysis*, as we can either:

1. go forward: we compute $\mathrm{Post}^*(C_0)$ and then determine whether $\mathrm{Post}^*(C_0) \cap C_{bad}$ is empty, or
2. go backward: we compute $\mathrm{Pre}^*(C_{bad})$ and then determine whether $\mathrm{Pre}^*(C_{bad}) \cap C_0$ is empty.

The main issue is the effective computation of $\mathrm{Post}^*(C_0)$ (or of $\mathrm{Pre}^*(C_{bad})$).

Forward analysis. The computation of $\text{Post}^*(C_0)$ is indeed the computation of the least fix-point of the function:

$$\begin{array}{rcl} 2^C & \to & 2^C \\ X & \mapsto & C_0 \cup \text{Post}(X) \end{array}$$

The computation of this fix-point is usually iterative: we start form C_0, then compute $C_0 \cup \text{Post}(C_0)$, $C_0 \cup \text{Post}(C_0) \cup \text{Post}(\text{Post}(C_0))$, ... This is a forward analysis[2], *i.e.* we start from an initial set of configurations, and we look forward to add more and more configurations.

Termination issues. If C is finite, the computation of $\text{Post}^*(C_0)$ will eventually terminate, as there is at most $2^{|C|}$ sets of configurations. When C is infinite or simply too big, the computation of $\text{Post}^*(C_0)$ does not terminate in general. Chapter 2 presents some symbolic methods to compute the reachability set when the system is a CFSM. In this section, we simply assume that one can effectively compute $\text{Post}^*(C_0)$ and thus check any invariance property. Let us emphasize on properties that are not simply invariance properties, but given by an observer.

1.2.2 Observers

A safety property ϕ may expressed by a set of runs: if the system S enables such a run, then it violates the property ϕ. The set of "bad runs" can be given by a transition system $\mathcal{O}_{\neg\phi}$ with a set of configurations labeled by "violate". This transition system is called an *observer* of the property ϕ.

When we have an observer $\mathcal{O}$ and a transition system $\mathcal{S}$, we identify the bad configurations of $\mathcal{S}$ by computing the synchronous product $\mathcal{O} \times \mathcal{S}$ of the two transition systems.

Definition 1.4 *The synchronous product of two labeled transition systems* $\mathcal{S}_1 = \langle C^1, \Sigma^1, C_0^1, \to^1 \rangle$ *and* $\mathcal{S}_2 = \langle C^2, \Sigma^2, C_0^2, \to^2 \rangle$ *is the labeled transition system* $\mathcal{S} = \langle C, \Sigma, C_0, \to \rangle$ *where:*

- $C = C^1 \times C^2$,
- $\Sigma = \Sigma^1 \cup \Sigma^2$,
- $C_0 = C_0^1 \times C_0^2$,
- $\to$ *defined by the rules:*
  1. *a common action* $a \in \Sigma^1 \cap \Sigma^2$ *occurs when it may occur in both systems:*

$$\frac{a \in \Sigma^1 \cap \Sigma^2,\ (c_1, a, c_1') \in \to^1 \ \bigwedge\ (c_2, a, c_2') \in \to^2}{((c_1, c_2), a, (c_1', c_2')) \in \to}$$

[2] On the contrary, the backward analysis starts from a given set of states X_0 and computes $X_0 \cup \text{Pre}(X_0)$, $X_0 \cup \text{Pre}(X_0) \cup \text{Pre}(\text{Pre}(X_0))$, etc.

2. *other actions occur according to the transition relation of one of the two systems:*

$$\frac{a \in \Sigma^1 \setminus \Sigma^2,\ (c_1, a, c_1') \in \rightarrow^1}{\forall c_2 \in C^2,\ ((c_1, c_2), a, (c_1', c_2)) \in \rightarrow}$$

and the symmetrical rule:

$$\frac{a \in \Sigma^2 \setminus \Sigma^1,\ (c_2, a, c_2') \in \rightarrow^2}{\forall c_1 \in C^1,\ ((c_1, c_2), a, (c_1, c_2')) \in \rightarrow}$$

Note that when $\Sigma^1 = \Sigma^2$, the behavior of the resulting system is the intersection of the behavior of each system. A observer is built so it does not interfere with the functioning of the observed system. Therefor, the observer must be complete, *i.e.* any sequence of events generated by $\mathcal{S}$ must be also allowed by the observer $\mathcal{O}$.

The bad configurations are then identified as the configurations $c \times c_{violate}$ where c is any configuration of the system $\mathcal{S}$ and $c_{violate}$ is any configuration of $\mathcal{O}$ labeled by "violate". Once bad configurations have been identified, we reduce the problem of checking whether ϕ is satisfied to the verification of an invariance property.

1.2.3 Beyond Safety Properties

We present here not only safety properties, but also more general properties that cannot be violate by a finite run of the system. One often expresses such properties by a formula of a *temporal logic*.

Formalization of a property in a temporal logic. Since we consider transition systems, we use *temporal logic*, like *Linear Temporal Logic (LTL)* [Pnu77] or *Computational Tree Logic (CTL)* [EC82], instead of a classical logic, because one wants to express properties on the evolution of the system. We introduce here a fragment of the Linear Temporal Logic to show how one can express more general properties on a transition system.

LTL is a logic operating on traces. We consider traces that are words over a finite alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_N\}$ and the following logical operators:

- the classical boolean operators: $\phi_1 \vee \phi_2, \phi_1 \wedge \phi_2, \neg\phi, \phi_1 \Rightarrow \phi_2$
- some temporal operators: $\mathbf{X}\phi, \mathbf{G}\phi, \mathbf{F}\phi, \phi_1 \mathbf{U} \phi_2$

The temporal operators are needed to express properties that hold not at the current step of the trace, but at a future step. The meaning of the temporal operators is:

- $\mathbf{X}\phi$, (Next): ϕ has to hold at the next step of the trace.
- $\mathbf{G}\phi$, (Globally): ϕ has to hold on the remaining of the trace.
- $\mathbf{F}\phi$, (Finally): ϕ eventually has to hold somewhere on the subsequent trace.
- $\phi_1 \mathbf{U} \phi_2$, (Until): ϕ_1 has to hold until ϕ_2 holds.

This set of operators is generally sufficient to formalize the expected behavior of a transition system; for example, we can express a causality: "a is finally followed by b" ($a \Rightarrow \mathbf{F}b$), or "$a$ immediately follows b or c ($b \vee c \Rightarrow \mathbf{X}a$). There is indeed a classification of logical properties in two kinds.

Safety and liveness properties.

1. *Safety* properties are intuitively the properties expressing that the system never goes wrong: it means that a bad event cannot occur, an "error" configuration cannot be reached, etc. For example, if σ_{error} is a property attached to a failure signal, then $\mathbf{G}\neg\sigma_{error}$ is a safety property expressing that this error never occurs.

2. *Liveness* properties are intuitively the properties expressing that the system will eventually go right: a procedure will terminate, a non-deterministic system is fair, etc. For example, if $\sigma_{success}$ is an event considered as a success, then $\mathbf{G}(\mathbf{F}\sigma_{success})$ is a liveness property expressing that this successful event can always happen.

The reasons for this distinction are:

1. This arguably corresponds to two kinds of expected behaviors of a system: we first expect the system does nothing wrong, and we also expect that it will finally perform its task.

2. The verification of safety properties are technically easier than the verification of liveness properties. Indeed, one can demonstrate that a safety property is violated by exhibiting a finite run, whereas liveness properties can be violated only by infinite runs.

3. A logical property can always be split between a safety part and a liveness part, and each part can be checked separately.

In the following, we will focus on the verification of a safety property on a labeled transition system $\langle C, \Sigma, C_0, \rightarrow\rangle$ with a finite alphabet Σ, and show how it can be reduced to the verification of an invariance property.

Example. We consider the set of atomic propositions $\Sigma = \{p, q, r\}$, the LTL formula $\phi = \mathbf{G}(p \Rightarrow \mathbf{X}q)$ and the transition system depicted on Figure 1.3. ϕ means that the following property should always hold: if a transition is labeled by p, then one of its successors must be labeled by q. We can see that the considered property is satisfied as long as the system does not reach one of the two configurations 4 and 5. So one can perform a reachability analysis which demonstrates that those two configurations are not reachable, thus the transition system is a model of the LTL formula ϕ.

1.2.4 Summary

Model Checking is the category of algorithmic tools and theoretical techniques aimed at determining whether a model satisfies a property described by a formula of a temporal

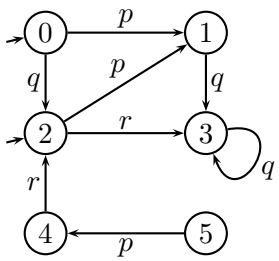


Figure 1.3: Example of transition system with atomic propositions

logic, or any similar formalism. We explained how one can check a safety property in three steps: the construction of an observer, the identification of the "bad configurations" and finally the reachability analysis. There are other kinds of properties and other verification methods, but the verification guideline presented here is the framework of the methods we present in Chapter 2 and close to the abstract interpretation framework.

1.3 Partial Orders and Lattices

We first recall some basic results on lattices and then introduce the abstract interpretation theory.

Let S be a set. A relation $\mathcal{R}$ is a subset of $S \times S$. We note $x\mathcal{R}y$ for $(x, y) \in \mathcal{R}$. A *partial order* is a relation $\mathcal{R} \subseteq S \times S$ satisfying three properties:

- reflexivity: $\forall x \in S,\ x\mathcal{R}x$,
- antisymmetry: if $x\mathcal{R}y$ and $y\mathcal{R}x$ then $x = y$,
- transitivity: if $x\mathcal{R}y$ and $y\mathcal{R}z$ then $x\mathcal{R}z$.

Classical examples of partial orders are:

- $\leq$ and $\geq$ on $\mathbb{N}$,
- $\subseteq$ on a power-set 2^S.

In the sequel, we use the generic notation $\sqsubseteq$ to denote a partial order.

Let S be a set and $\sqsubseteq$ a partial order on S. Let $X \subseteq S$. $y \in S$ is an upper bound (resp. lower bound) of X if $\forall x \in X$, $x \sqsubseteq y$ (resp. $y \sqsubseteq x$). z is the least upper bound (lub) of X if it is an upper bound and, for any other upper bound y, $z \sqsubseteq y$. The least upper bound does not always exist, but is unique when defined. The greatest lower bound (glb) is defined in the same way.

Definition 1.5 *$(S, \sqsubseteq)$ is a lattice if, for any pair of elements $x, y \in S$ the lub and the glb of the set $\{x, y\} \subseteq S$ are defined ($x \sqcup y$ denotes the lub and $x \sqcap y$ the glb). A lattice is*

bounded if S has a bottom element $\bot$ and a top element $\top$ satisfying $\forall x \in S,\ \bot \sqsubseteq x \sqsubseteq \top$ and is complete if for any subset $X \subseteq S$, the lub and glb of X, denoted by $\sqcup X$ and $\sqcap X$, exist.

Classical examples of lattices are:

- the power-set lattice: if E is a set, then $(2^E, \subseteq)$ is a lattice.
- the lattice of intervals: we define $\mathcal{I} = \{[a,b] | a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a < b\} \bigcup \{\emptyset\}$; $(\mathcal{I}, \subseteq)$ is a lattice.

These two lattices are both bounded and complete. In the sequel, we will consider only bounded lattices.

Fix-points. $x \in S$ is a fix-point of $f : S \to S$ if $f(x) = x$. The least fix-point (lfp) of f is the greatest lower bound of the fix-points of f, and the greatest fix-point (gfp) of f is the lowest upper bound of the same set. x is a post fix-point (resp. a pre fix-point) if $x \sqsupseteq f(x)$ (resp. $x \sqsubseteq f(x)$).

As we explained in the previous section, the reachability set $\mathrm{Post}^*(C_0)$ is the least fix-point of the function $X \mapsto \mathrm{Post}(X) \cup C_0$, which is a function on the complete lattice $(2^C, \subseteq)$.

We can compute the least fix-point of f thanks to the two following theorems.

Tarski's and Kleene's theorems. In this paragraph, we assume that $(S, \sqsubseteq)$ is a complete lattice.

Definition 1.6 *Let $f : S \to S$ be a function.*

- *f is* monotonic *if:*
$$\forall x, y \in S,\ x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$
- *f is* continuous *if:*
$$\forall X \subseteq S, f(\sqcup X) = \sqcup\{f(x) | x \in X\} \text{ and } f(\sqcap X) = \sqcap\{f(x) | x \in X\}$$

Note that if f is continuous, then it is monotonic because if $x \sqsubseteq y$, then $y = x \sqcup y$ and so $f(y) = f(x) \sqcup f(y) \sqsupseteq f(x)$.

Theorem 1.1 (Knaster-Tarski theorem) [3] *Assuming that f is a monotonic function, then:*
$$\mathrm{lfp}(f) = \sqcap\{x \mid x \sqsupseteq f(x)\}$$

[3] This is only a part of the "real" Knaster-Tarski theorem, the one we will employ in the sequel.

Proof: We consider $D = \{x \mid x \sqsupseteq f(x)\}$ and $l = \sqcap D$, which exists since $(S, \sqsubseteq)$ is a complete lattice.

For any $x \in D$, we have $x \sqsupseteq l$, so $x \sqsupseteq f(x) \sqsupseteq f(l)$ because f is monotonic. So $f(l)$ is a lower bound of D. By definition of l, we have $f(l) \sqsubseteq l$. We re-apply the monotonic function f: $f(f(l)) \sqsubseteq f(l)$. $f(l) \in D$ which means $l \sqsubseteq f(l) \sqsubseteq l$. So l is a fix-point of f and, since all fix-points of f are in D, l is the least fix-point of f. □

In other word, the least fix-point of f is lesser than any post fix-point of f. This theorem does not provide an effective computation of $\mathrm{lfp}(f)$, unlike the following one:

Theorem 1.2 (Kleene's theorem) *Assuming that f is a continuous function,* $\mathrm{lfp}(f)$ *is the limit of the sequence:*

$$\begin{array}{rcl} u_0 & = & \bot \\ u_{n+1} & = & u_n \sqcup f(u_n) \end{array}$$

Proof: The limit of this sequence is formally defined as $x = \bigsqcup_n f^n(\bot)$. Thus x is a fix-point of f, because:

$$\begin{array}{rcll} f(x) & = & f\left(\bigsqcup_n f^n(\bot)\right) & \\ & = & \bigsqcup_n f(f^n(\bot)) & f \text{ is continuous} \\ & = & \bigsqcup_n f^{n+1}(\bot) & \\ & = & \bot \sqcup \bigsqcup_{n \geq 1} f^{n+1}(\bot) & \\ & = & \bigsqcup_{n'} f^{n'}(\bot) & \text{renaming } n' \leftarrow n+1 \\ & = & x & \end{array}$$

x is a fix-point, let us demonstrate that it is the least fix-point. Let y be another fix-point of f. Since $\bot \sqsubseteq y$, we have $f(\bot) \sqsubseteq f(y) = y$. By induction, we obtain $\forall n,\ f^n(\bot) \sqsubseteq y$, so $x = \bigsqcup_n f^n(\bot) \sqsubseteq y$. □

This theorem gives a naive iterative algorithm for the computation of the least fix-point of f, without any guarantee of termination. Indeed, this computation may not terminate when the lattice $(S, \sqsubseteq)$ does not satisfy the *ascending chain condition*, *i.e.* each ascending sequence $u_0 \sqsubseteq u_1 \sqsubseteq \cdots \sqsubseteq u_n \sqsubseteq \ldots$ is stationary after a finite number of steps. This is the reason why we must perform some kind of approximations.

1.4 Abstract Interpretation

Abstract interpretation is a verification technique dealing with an abstract semantics rather than the concrete semantics of a program or a model of a system. Most verification issues can be reduced to the computation of the least fix-point of a function F in a semantic domain S, which has a lattice structure. Since this fix-point is often not computable, one may choose to compute an over-approximation of the fix-point, that may be sufficient to prove that a systems satisfies a safety property ; we explained

previously how the verification of a safety property can be reduced to a reachability analysis. So if no configuration of the over-approximation of the reachability set is forbidden, the property is satisfied. Otherwise, we can't tell if the property is violated or if the approximations made are too rough.

Principles of abstract interpretation. Two kinds of approximations are employed, in the abstract interpretation theory, to over-approximate the least fix-point of a function F:

1. Static approximations: instead of searching the least fix-point of F in the original lattice $(\Gamma, \sqsubseteq)$, we compute the least fix-point of its abstraction $F^\sharp$ in an *abstract lattice* $(\Lambda, \sqsubseteq^\sharp)$, linked to Γ thanks to a *Galois connection* (see below). The abstract lattice is chosen so that the computation of a fix-point in this lattice is easier than in the original lattice.

2. Dynamic approximations: when the abstract lattice does not satisfy the ascending chain condition, we try to "guess" the limit of the sequence $(u_n)_{n\in\mathbb{N}}$ with a *widening operator* which ensures the termination of the computation at the cost of some approximation.

We detail both approximations, from a mathematical point of view.

Galois connection. The function F is given by the semantics of the considered system. For example, the computation of the reachability set is the least fix-point of the function:

$$\begin{array}{rccc} F: & 2^C & \to & 2^C \\ & X & \mapsto & C_0 \cup \mathrm{Post}(X) \end{array}$$

In this example, the concrete lattice is the 2^C, and we want to abstract sets of configurations by elements of a simpler abstract lattice.

We assume that the concrete lattice $(\Gamma, \sqsubseteq)$ is a complete lattice, and that the *abstract lattice* $(\Lambda, \sqsubseteq^\sharp)$ is also a complete lattice. A *Galois connection* is given by two functions $\alpha : \Gamma \to \Lambda$, called the abstraction function, and $\gamma : \Lambda \to \Gamma$, called the concretization function, satisfying:

$$\forall s \in \Gamma, \forall a \in \Lambda,\ \alpha(s) \sqsubseteq^\sharp a \Leftrightarrow s \sqsubseteq \gamma(a)$$

This Galois connection is noted:

$$(\Gamma, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\Lambda, \sqsubseteq^\sharp)$$

Then $F^\sharp$ is defined as $F^\sharp \triangleq \alpha \circ F \circ \gamma$, with the property:

$$\mathrm{lfp}(F) \sqsubseteq \gamma(\mathrm{lfp}(F^\sharp))$$

So the computation of $\mathrm{lfp}(F^\sharp)$ gives an over-approximation of $\mathrm{lfp}(F)$.

Example 1.2 *$(2^{\mathbb{Z}}, \subseteq)$ is a complete lattice. The lattice of intervals $(\mathcal{I}, \subseteq)$ is also a complete lattice. We have a Galois connection $(2^{\mathbb{Z}}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{I}, \subseteq)$, where the abstraction function is $\alpha(P) = [min(P), max(P)]$ and the concretization function is the identity. Elements of the abstract lattice can be easily manipulated; but there are still infinitely increasing sequences, like $([-n, n])_{n\in\mathbb{N}}$. So the fix-point computation $\bigcup_{n\in\mathbb{N}} f^n(\bot)$ may not terminate, unless we use a widening operator.*

Widening operator. A widening operator ∇ extrapolates the limit of an increasing sequence from a finite number of terms, usually two, as in the following definition:

Definition 1.7 *Let $(\Lambda, \sqsubseteq)$ be a lattice. $\nabla : \Lambda \times \Lambda \to \Lambda$ is a widening operator if:*

1. *$\forall x, y \in \Lambda,\ x \sqsubseteq x\nabla y \ \wedge\ y \sqsubseteq x\nabla y$,*
2. *if $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \ldots$ is an increasing sequence, the increasing sequence defined as $y_0 = x_0$ and $y_{n+1} = y_n \nabla x_{n+1}$ is stationary after a finite number of steps.*

The first condition ensures that the widening is an over-approximation of the least upper bound, and the second one ensures the termination of the computation of the fix-point computation. Thus we have:

Theorem 1.3 *[CC77a] Let $(\Lambda, \sqsubseteq^\sharp)$ be a complete abstract lattice, and $f : \Lambda \to \Lambda$ be a function. The sequence defined as:*

$$\begin{array}{rcl} u_0 & = & \bot \\ u_{n+1} & = & u_n \nabla f(u_n) \end{array}$$

is stationary after a finite number of steps and its limit $u_\infty \sqsupseteq^\sharp \mathrm{lfp}(f)$.

Example 1.3 *The standard widening operator of the lattice of intervals $(\mathcal{I}, \subseteq)$ is defined as $[a_1, b_1]\nabla[a_2, b_2] =$*

- *$[a_1, b_1] \sqcup [a_2, b_2]$ if $[a_1, b_1] \not\subseteq [a_2, b_2]$,*
- *$[a, b]$ otherwise, with $a = a_1$ if $a_1 = a_2$ and $a = -\infty$ if $a_1 > a_2$, and $b = b_1$ if $b_1 = b_2$ and $b = +\infty$ if $b_1 < b_2$.*

The principle of this widening operator is to consider that if the interval grows, it may grow indefinitely.

This approximation is often not accurate, but it can be improved, by setting a priori a finite number of values $v_1 < v_2 < v_k$. In the definition of the widening , if $b_1 < b_2$, then b is the smallest value $v_i > b_2$ or $+\infty$ if $b_2 > v_k$.

This improved widening is called the widening threshold.

Representations framework. The previous Galois connection framework assumes that the abstract lattice is complete. Some abstract lattices, like the lattice of convex polyhedra, are not complete. In these cases, one may employ the weaker *representations framework* of [Bou92] instead of a classical Galois connection to formalize the abstraction. We recall the main results and invite the reader to see [Bou92] for more details.

Let $(\Lambda, \sqsubseteq)$ be a partially ordered set with a smallest element $\bot_\Lambda$, and $\gamma : \Lambda \to \Gamma$ a concretization function (*i.e.* γ is monotone and $\gamma(\bot_\Lambda) = \bot_\Gamma$).

Let $\nabla : \Lambda \times \Lambda \to \Lambda$ be an operator satisfying:

1. $\forall a, a' \in \Lambda,\ a \sqsubseteq a \nabla a'$ and $a' \sqsubseteq a \nabla a'$ (∇ is an upper bound operator);

2. $\forall (a_i)_{i\in\mathbb{N}} \in \Lambda^{\mathbb{N}}$ such that $\forall i \geq 0, a_i \sqsubseteq a_{i+1}$, the sequence $(a_i)_{i\in\mathbb{N}}$ defined as $a'_0 = a_0$ and $a'_{i+1} = a'_i \nabla a_{i+1}$ has a greatest element a_∞ (thus obtained after a finite number of steps).

Here ∇ replaces both the least upper-bound and the widening operator. The main preservation theorem is the following one:

Theorem 1.4 *Let $F : \wp(S) \to \wp(S)$ be a monotonic function and $F^\sharp : \Lambda \to \Lambda$ a monotonic function such that $\gamma \circ F^\sharp \supseteq F \circ \gamma$. The sequence $(a_i)_{i\in\mathbb{N}}$ defined by:*

$$\left\{ \begin{array}{rcl} a_0 & = & \bot \\ a_{i+1} & = & a_i \nabla F^\sharp(a_i) \end{array} \right.$$

has a greatest element a_∞, which is a post fix-point of $F^\sharp$. Moreover, $\gamma(a_\infty)$ is a post fix-point of F.

We give here a simple proof of this theorem. [Bou92] gave a proof of a more general theorem; in our case, Thm. 1.4 is sufficient.

Proof: We first prove that the sequence $(a_i)_{i\in\mathbb{N}}$ has a greatest element. If for some i, $F^\sharp(a_i) \sqsubseteq a_i$, then a_i is the limit of the sequence and a post fix-point of $F^\sharp$.

We assume by contradiction that $(a_i)_{i\in\mathbb{N}}$ has no greatest element. Then we have $\forall i : a_{i+1} = a_i \nabla F^\sharp(a_i) \sqsupseteq a_i$, and $(a_i)_{i\in\mathbb{N}}$ is an increasing sequence. Moreover, $a_0 = \bot \sqsubseteq F^\sharp(a_0)$ and the monotonicity of $F^\sharp$ implies that:

$$\begin{array}{rcl} a'_0 & = & a_0 \\ a'_{i+1} & = & F^\sharp(a_i) \end{array}$$

is an increasing chain. The definition of the widening operator ensures that:

$$\begin{array}{rcl} a''_0 & = & a_0 \\ a''_{i+1} & = & a''_i \nabla a'_{i+1} \end{array}$$

has a greatest element a_∞. Then we show by induction on i that $\forall i \geq 0,\ a''_i = a_i$: this is obvious for $i = 0$ and

$$\begin{array}{rcll} a''_{i+1} & = & a''_i \nabla a'_{i+1} & \\ & = & a_i \nabla a'_{i+1} & \text{(induction hypothesis)} \\ & = & a_i \nabla F^\sharp(a_i) & \text{(definition of } a'_{i+1}\text{)} \\ & = & a_{i+1} & \end{array}$$

Consequently, the sequence $(a_i)_{i\in\mathbb{N}}$ has a greatest element. This is a contradiction of the first assumption.

We now check that $\gamma(a_\infty)$ is a post fix-point of F.

$$\begin{array}{rcll} F(\gamma(a_\infty)) & \subseteq & \gamma(F^\sharp(a_\infty)) & \text{(assumption on } F^\sharp\text{)} \\ & \subseteq & \gamma(a_\infty) & \text{by } F^\sharp(a_\infty) \sqsubseteq a_\infty \text{ and monotonicity of } \gamma \end{array}$$

□

Computation of a reachability set. We aim at computing an over-approximation of the reachability set $\mathrm{Post}^*(S)$, in other words the least fix-point of the function:

$$\begin{array}{rccc} \mathcal{F}: & 2^C & \to & 2^C \\ & X & \mapsto & \mathrm{Post}(X) \cup S \end{array}$$

This is a monotonic function and the lattice 2^C is complete. According to the Knaster-Tarski theorem, the set of configurations $\mathrm{lfp}(\mathcal{F})$ is included in any post fix-point of $\mathcal{F}$. Thm. 1.4 gives such a post fix-point; we only need those two results to ensure that we effectively compute an over-approximation of the reachability set.

The representations framework will be the theoretical foundation of our abstractions, instead of the Galois connection, because the abstract lattice of regular languages is not complete (*cf.* Chapter 3).

1.5 Regular Languages and Finite Automata

This section is not an exhaustive presentation of the well-known theory of regular languages and finite automata. It focuses on the definition of finite automata and the presentation of three operations: the determinization, the minimization and the quotient of a finite automaton by an equivalence relation on states. The reason for this presentation is that we will define similar algorithms for lattice automata (*cf.* Chapter 4). The reader can refer to [HJU79] for more details on finite automata and regular languages.

Words and languages. Let Σ be a finite set, called an *alphabet*; elements of Σ are called *letters*. Σ^* is the free monoid on Σ, i.e. the set of finite *words* (finite concatenation of letters). ε is the empty word and a word w of length n will be written $w = a_1.a_2 \ldots a_n$. We focus on the simplest languages: the regular languages, which can be recognized by finite automata.

Finite automata. A finite automaton is a tuple $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ where:

- Σ is a finite set (the *alphabet*),
- Q is a finite set of states,
- $Q_0 \subseteq Q$ is the set of initial states,
- $Q_f \subseteq Q$ is the set of final states,
- $\delta \subseteq Q \times \Sigma \times Q$ is the set of transitions.

We note $q \xrightarrow{a} q'$ if $(q, a, q') \in \delta$. A finite word $w = a_0.a_1 \dots a_n \in \Sigma^*$ is recognized by $\mathcal{A}$ if there exists $n+1$ states: $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ with $q_0 \in Q_0$ and $q_n \in Q_f$. $L(\mathcal{A}) \subseteq \Sigma^*$ is the language recognized by the automaton $\mathcal{A}$. A language is *regular* if it is recognized by a finite automaton. $\mathrm{Reg}(\Sigma)$ is the set of regular languages.

Classical set operations like union or intersection can be easily implemented in terms of automata operations. For example, the union of two regular languages L_1 and L_2 recognized by two automata $\mathcal{A}_1 = \langle \Sigma, Q^1, Q_0^1, Q_f^1, \delta^1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q^2, Q_0^2, Q_f^2, \delta^2 \rangle$ is recognized by the automaton $\mathcal{A}_3 = \langle \Sigma, Q^3, Q_0^3, Q_f^3, \delta^3 \rangle$, where:

- $Q^3 = Q^1 \cup Q^2$ assuming the two sets of states are distinct,
- $Q_0^3 = Q_0^1 \cup Q_0^2$,
- $Q_f^3 = Q_f^1 \cup Q_f^2$,
- $\delta^3 = \delta^1 \cup \delta^2$.

The finite automaton representation is not canonical, *i.e.* a same regular language can have several different automata representations, which may be an issue for the definition of more complex operations. Hopefully, regular languages have a canonical representation in terms of minimal deterministic automata (MDAs).

Deterministic finite automata. A finite automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ is *deterministic* if:

1. there is a single initial state: $\mathrm{card}(Q_0) = 1$,
2. for any state $q \in Q$ and any letter $a \in \Sigma$, if $q \xrightarrow{a} q_1$ and $q \xrightarrow{a} q_2$, then $q_1 = q_2$.

A classical algorithm transforms a non-deterministic automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ into a deterministic finite automaton $\mathcal{A}' = \langle \Sigma, X, X_0, X_f, \Delta \rangle$ recognizing the same language, with:

- $X = 2^Q$,
- $X_0 = \{Q_0\}$,
- $X_f = \{E \subseteq Q | E \cap Q_f \neq \emptyset\}$,

- $\Delta = \{(E_1, a, E_2) | \exists q_1 \in E_1, q_2 \in E_2, (q_1, a, q_2) \in \delta\}$

This algorithm is exponential because each state $x \in X$ of the resulting automaton corresponds to a set of states $x \in 2^Q$ of the initial automaton.

When an automaton is deterministic, its transitions set δ defines a transition function (also called δ):

$$\begin{array}{rccl} \delta: & Q \times \Sigma & \rightarrow & \Sigma \\ & (q, a) & \mapsto & q' \text{ where } q' \text{ is the only state such that } (q, a, q') \in \delta \end{array}$$

The transitive closure of this function is defined recursively as:

$$\begin{array}{rccl} \delta^*: & Q \times \Sigma^* & \rightarrow & \Sigma \\ & (q, \varepsilon) & \mapsto & q \\ & (q, a.w) & \mapsto & \delta^*(\delta(q, a), w) \text{ with } a \in \Sigma \text{ and } w \in \Sigma^* \end{array}$$

This definition gives a simpler characterization of the language recognized by a deterministic finite automaton ; a word $w \in \Sigma^*$ is recognized by a deterministic automaton $\mathcal{A} = \langle \Sigma, Q, \{q_0\}, Q_f, \delta \rangle$ if $\delta^*(q_0, w) \in Q_f$.

Deterministic finite automata are not a canonical representation of regular languages, since two different deterministic automata can recognize the same regular language. This canonical representation is the result of an operation called minimization.

Minimal deterministic automata. A deterministic automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ is *minimal* if any other deterministic automaton $\mathcal{A}' = \langle \Sigma, Q', Q'_0, Q'_f, \delta' \rangle$ recognizing the same language $L(\mathcal{A})$ has more states: $\text{card}(Q) \leq \text{card}(Q')$. For a given language L, there is a unique minimal deterministic finite automaton (MDA) recognizing it (up to isomorphism). One obtains this automaton by quotienting the automaton by the *Myhill-Nerode equivalence relation.*

First we present the quotient operation, for any equivalence relation $\equiv$; this operation consist roughly in merging states belonging to the same equivalence class:

Definition 1.8 (Quotient of an automaton) *Let $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ be a finite automaton and $\equiv \subseteq Q \times Q$ an equivalence relation. The quotient automaton $\mathcal{A}/\equiv = \langle \Sigma, \overline{Q}, \overline{Q_0}, \overline{Q_f}, \overline{\delta} \rangle$ is defined as:*

- $\overline{Q} \subseteq 2^Q$ *is the set of equivalence classes of* $\equiv$*. The class of a state $q \in Q$ is $\bar{q} \subseteq Q$,*
- $\overline{Q_0} \triangleq \{\bar{q} \mid \bar{q} \cap Q_0 \neq \emptyset\}$,
- $\overline{Q_f} \triangleq \{\bar{q} \mid \bar{q} \cap Q_f \neq \emptyset\}$,
- $\overline{\delta} = \{(\bar{q}, a, \overline{q'}) \mid (q, a, q') \in \delta\}$.

Note that $L(\mathcal{A}) \subseteq L(\mathcal{A}/\equiv)$, no matter the equivalence relation $\equiv$ is. With a proper equivalence relation, $\mathcal{A}/\equiv$ is minimal and deterministic if $\mathcal{A}$ is deterministic.

Considering a deterministic automaton $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$, we can define an equivalence relation $\equiv \subseteq Q \times Q$: $q \equiv q'$ if and only if:

$$\forall w \in \Sigma^*, \begin{cases} \delta^*(q,w) \text{ is defined whenever } \delta^*(q',w) \text{ is defined} \\ \text{and} \\ \delta^*(q,w) \in Q_f \Leftrightarrow \delta^*(q',w) \in Q_f \end{cases}$$

This is obviously an equivalence relation, called the Myhill-Nerode equivalence relation.

Proposition 1.1 *If $\mathcal{A}$ is deterministic, with all states reachable from Q_0 and co-reachable from Q_f, and $\equiv$ is the Myhill-Nerode equivalence relation, then $\mathcal{A}/\equiv$ is minimal and deterministic and $L(\mathcal{A}) = L(\mathcal{A}/\equiv)$.*

Proof:

- [$\mathcal{A}/\equiv$ **is deterministic**] We first prove that $\mathcal{A}/\equiv = \langle \Sigma, \overline{Q}, \overline{Q_0}, \overline{Q_f}, \overline{\delta} \rangle$ is deterministic if $\mathcal{A} = \langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ is deterministic.

  1. Since $\mathcal{A}$ is deterministic, $Q_0 = \{q_0\}$ so $\overline{Q_0} = \{\overline{q_0}\}$,
  2. If we have two transitions $(\bar{q}, a, \overline{q_1}), (\bar{q}, a, \overline{q_2}) \in \overline{\delta}$, it means that we have two transitions $(q, a, q_1), (q', a, \overline{q_2}) \in \delta$ with $q \equiv q'$. We show that $q_1 \equiv q_2$. Let $w \in \Sigma^*$ such that $\delta^*(q_1, w)$ is defined. So $\delta^*(q, a.w)$ is defined and $\delta^*(q', a.w) = \delta^*(q_2, w)$ is as well defined, because $q \equiv q'$. For the same reason $\delta^*(q_1, w) \in Q_f \Leftrightarrow \delta^*(q_2, w) \in Q_f$.

  It means that $\mathcal{A}/\equiv$ is deterministic.

- [$\mathcal{A}/\equiv$ **recognizes the same language as** $\mathcal{A}$] By construction, $L(\mathcal{A}) \subseteq L(\mathcal{A}/\equiv)$. Let us prove the converse by a recurrence. Let $w = a_1 \dots a_n$ be a word recognized by $\mathcal{A}/\equiv$; there are $n+1$ states of $\mathcal{A}/\equiv$ such that $\overline{q_0} \xrightarrow{a_1} \overline{q_1} \xrightarrow{a_2} \dots \xrightarrow{a_n} \overline{q_n}$ accepts the word w. We must prove that there is a run $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ of $\mathcal{A}$ with $q_0 \in Q_0$ and $q_f \in Q_f$.

  - There is $q_n \in \overline{q_n}$ being an final state of $\mathcal{A}$, according to the construction of $\mathcal{A}/\equiv$,
  - We assume that there is a path $q_i \xrightarrow{a_{i+1}} q_{i+1} \xrightarrow{a_{i+2}} \dots \xrightarrow{a_n} q_n$ of $\mathcal{A}$ with $\forall j \geq i,\ q_j \in \overline{q_j}$ and $q_n \in Q_f$. Since $\overline{q_0} \xrightarrow{a_1} \overline{q_1} \xrightarrow{a_2} \dots \xrightarrow{a_n} \overline{q_n}$ is a path accepting w in $\mathcal{A}/\equiv$, the definition of $\mathcal{A}/\equiv$ states that there are two states $q'_{i-1} \in \overline{q_{i-1}}$ and $q'_i \in \overline{q_i}$ such that $q'_{i-1} \xrightarrow{a_i} q'_i$ is a transition of $\mathcal{A}$. Since $q_i \equiv q'_i$, there is a path $q'_i \xrightarrow{a_{i+1}} q'_{i+1} \xrightarrow{a_{i+2}} \dots \xrightarrow{a_n} q'_n$ of $\mathcal{A}$ with $\forall j \geq i,\ q'_j \in \overline{q_j}$ and $q'_n \in Q_f$.

  By induction we prove that there is a run $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ with $q_n \in Q_f$ and $q_0 \in \overline{q_0}$. So w is recognized by $\mathcal{A}$; $L(\mathcal{A}/\equiv) \subseteq L(\mathcal{A})$.

- [$\mathcal{A}/\equiv$ **is minimal**] We assume by contradiction that there is a deterministic automaton $\mathcal{A}'$ recognizing the same language as $\mathcal{A}$ with strictly less states than $\mathcal{A}/\equiv$. According to the two previous parts of the proof, $\mathcal{A}'/\equiv$ is a deterministic automaton recognizing the same language as $\mathcal{A}'$. It also has less states than $\mathcal{A}'$, and thus strictly less states than $\mathcal{A}/\equiv$. For each state q of $\mathcal{A}/\equiv$, there is a word w_q leading to this state from the initial states. Those words also lead to some states in $\mathcal{A}'/\equiv$, and since this automaton has strictly less states, at least two words w_{q_1} and w_{q_2} lead to the same state of $\mathcal{A}'/\equiv$. Thus $q_1 \equiv q_2$, which is impossible.

  So no deterministic automaton can have strictly less states than $\mathcal{A}/\equiv$.

□

This unique automaton representation allows the effective manipulation of regular languages; a function applying to automata also defines a function applying on regular languages.

1.6 Conclusion

The chapter presented the basic definitions and results needed in the following chapters: we presented the CFSM model studied in Chapter 2, the principle of the verification of safety properties and the theoretical framework of our work presented in Chapters 3, 4 and 5: the abstract interpretation.

We also recalled some notions of the lattice theory and the finite automata theory.

The main issue of the verification of safety properties is, for the systems we consider, to have an efficient representation of the set of configurations and, more generally, to compute the limit of an increasing sequence of regular sets of configurations, coded by finite automata. We focus on this issue in the following chapter.

Chapter 2

Acceleration Techniques for FIFO Channel Systems and Extrapolations of Sequences of Automata

This chapter presents some work in the field of acceleration techniques for the analysis of FIFO channel systems, modeled by a CFSM.

In Section 1.1, we gave the semantics of a CFSM; a configuration of the system is given by a control point and the content of all queues. We want to have an efficient representation of sets of configurations and a reachability algorithm, so that we can check whether a given safety property holds (*cf.* Section 1.2). The acceleration techniques we present in this chapter provide reachability semi-algorithms. Other techniques, which were not initially aimed at the verification of FIFO channel systems, are useful since they can compute the limit of a sequence of symbolic values, when those values are represented by automata.

In Section 2.1, we explain the principle of acceleration techniques, which aim at computing, in a single step, the configurations reachable from a set of configurations S by iterating a loop θ an arbitrary number of times. The efficiency of acceleration techniques highly depends on the expressive power of the symbolic structure employed to represent sets of configurations. We detail what the acceleration techniques can do when one represents the queue contents by:

- a Queue-Contents Decision Diagram (QDD),
- a Constrained Queue-Contents Decision Diagram (CQDD),
- a Simply Regular Expression (SRE),
- a Semi-Linear Regular Expression (SLRE).

The second section emphasizes on some methods to guess the limit of a sequence of symbolic values represented by a sequence of automata. The aim is usually to compute

a representation of the set of states $\text{Post}^*(S)$ (resp. $\text{Pre}^*(S)$) reachable (resp. co-reachable) from a regular set of configurations S, *i.e.* to compute the limit of the sequence defined as:

$$\begin{aligned} U_0 &= S \\ U_{i+1} &= \text{Post}(U_i) \cup U_i \end{aligned}$$

where each term U_i is represented by an automaton.

In some cases, this computation can be done without any approximation: for example, the computation of $\text{Pre}^*(S)$, where S is a regular set of configurations of a Pushdown System (PDS) is exact [BEM97].

But in the general case, this exact computation is not feasible and we have the choice between:

- a method trying to build a finite automaton, which is exact but may not terminate, or
- an extrapolation method, which terminates but often computes an over-approximation of the limit of the sequence of finite automata.

We present one exact method and two extrapolation methods ; the automata considered in this last section will be either transducers[1] or synchronous observers[2].

2.1 Accelerations Techniques for the Analysis of FIFO Channels Systems

2.1.1 Principle of Acceleration Techniques

We first present the principles of acceleration techniques. Acceleration is a verification technique for transition systems which have an infinite set of configurations but a finite control structure. For example, a CFSM $\mathcal{M} = \langle C, \Sigma, c_0, \Delta \rangle$ has a finite control structure C but defines an infinite transition system (*cf.* Section 1.1).

The core of acceleration techniques is to consider the loops of the control structure. A loop of the transition system is a sequence of transitions $\theta = (c_0, a_0, c_1), (c_1, a_1, c_2), \ldots (c_n, a_n, c_0)$. We write $(p_0, w_0) \rightarrow_\theta (p_0, w_n)$ if there is a sequence of configurations satisfying :$((c_0, w_0) \xrightarrow{a_0} (c_1, w_1) \xrightarrow{a_1} \ldots \xrightarrow{a_n} (c_0, w_n)$. Given a set of configurations $S = \{(c_0, w)\}$, we want to compute $\text{Post}^*_\theta(S) = \{(c_0, w') | \exists (c_0, w) \in S, (c_0, w) \rightarrow^*_\theta (c_0, w')\}$. This computation raises several issues.

Representations. The first issue is to find a symbolic representation of the set of configurations S that can also represent $\text{Post}^*_\theta(S)$. This representation must also be manageable, *i.e.* the algorithms needed for the computation of $\text{Post}^*_\theta(S)$ must have a reasonable complexity.

[1] A transducer is an automaton transforming a word into another word. In this chapter, they are employed to encode a transition function.

[2] See Section 1.2.

In practice, we restrict ourselves to a class $\mathcal{C}$ of sets of configurations, such as regular sets, that can be represented by automata. The chosen class $\mathcal{C}$ shall be stable for Post^*_θ, if possible, which means: $\forall S \in \mathcal{C}, \forall \theta, \mathrm{Post}^*_\theta(S) \in \mathcal{C}$. Generally, the proof of this property is constructive and gives the algorithm computing $\mathrm{Post}^*_\theta(S)$ for any set S and any loop θ.

Loop acceleration. The computation of $\mathrm{Post}^*_\theta(S)$ starts by the computation of the first steps $\mathrm{Post}_\theta(S)$, $\mathrm{Post}_\theta(S) \cup \mathrm{Post}_\theta(\mathrm{Post}_\theta(S))$, ... If a property ensures that the sequence $(\bigcup_{i \leq n} \mathrm{Post}^i_\theta(S))_{n \in \mathbb{N}}$ is eventually stationary, then one can compute this sequence until the limit is reached.

However, in the general case, one has to find automatically the limit of this sequence. An acceleration algorithm can compute directly $\bigcup_{i=0}^{\infty} \mathrm{Post}^i_\theta(S)$ after looking at the first terms of the sequence. Each class $\mathcal{C}$ of sets of configurations has its own acceleration algorithm.

Such algorithms may assume some restrictions on θ, *i.e.* some loops cannot be accelerated. This is the case of the QDD representation. Other acceleration algorithms, like the one for CQDDs, work without any restriction on θ.

Termination. Since there are potentially an infinite number of loops, the computation of a reachability set with an acceleration method may not terminate even if the effect of each loop can be effectively computed. The problem is to choose what set of loops must be accelerated, and in what order the acceleration must be applied.

This issue is illustrated on Figure 2.1. We represent the queue contents by regular languages. The two elementary loops labeled by $!a$ and $!b$ can be easily accelerated, generating the languages a^* and b^* respectively; in the general case, if $w \subseteq \{a, b\}^*$ is a loop, the language generated is simply $\mathrm{Post}_{w^*}(L) = L.w^*$. The language of the whole system, $(a + b)^*$, cannot however be obtained by the acceleration of a loop. This example shows how a nested loop may be an obstacle an analysis based on the acceleration principle cannot overcome without a transformation of the automaton.

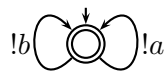


Figure 2.1: Nested loops

In the following, we detail some acceleration techniques based on different symbolic representations, aimed at the analysis of CFSMs:

1. the *Queue-Contents Decision Diagrams* (QDDs),
2. the *Constrained Queue-Contents Decision Diagrams* (CQDDs),
3. the *Simple Regular Expressions* (SREs),
4. the *Semi-linear Regular Expressions* (SLREs).

2.1.2 Queue-Contents Decision Diagrams

A *Queue-Contents Decision Diagram* (QDD) [BG97, BGWW97] is simply a finite automaton recognizing concatenations of words $w_1 \sharp w_2 \sharp \dots \sharp w_N$, so a regular set of configurations sharing the same control point can be represented by a QDD. This allows to encode some relations between the contents of several queues.

Example 2.1 *Considering a system with two queues with alphabets* $\Sigma_1 = \{a, b\}$ *and* $\Sigma_2 = \{c, d\}$*. The QDD depicted on Figure 2.2 defines the language* $L = a^* \sharp (c + d)^* + a^* b (a + b)^* \sharp c^* d (c + d)^*$ *and encodes the relation "if there is at least one* b *in the first queue, then there is at least one* d *in the second queue".*

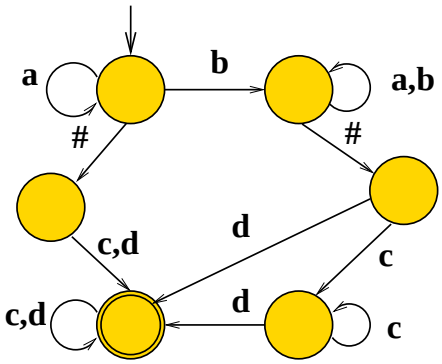


Figure 2.2: Example of QDD

Language recognized by a QDD. The ordering of the queues may be arbitrary, it does not alter the expressive power of the QDDs ; QDDs recognize languages of the form:

$$\bigcup_{1 \le j \le k} \prod_{1 \le i \le N} L_{i,j}$$

where $L_{i,j} \in \mathrm{Reg}(\Sigma_i)$ is a regular set built on the alphabet Σ_i.

Remark 2.1 *The name "QDD" refers to the famous Binary Decision Diagrams (BDDs)[MO83]; like the BDDs, the QDDs decide the value of the queue contents in a fixed order.*

Some conditions on the loop θ (see below) ensure that this set is stable for Post^*_θ. We now give an overview of the acceleration algorithm, with some assumptions on θ.

Loops involving a single FIFO channel. When all the operations of a loop θ involve a single queue, the computation of $\mathrm{Post}^*_\theta(L)$, where L is a regular set represented by a *QDD*, is performed by adding states and transitions, and modifying the set of initial and final states of the automaton representing L.

We give here the sketch of the algorithm, the reader can refer to [BGWW97] for more details. One can first compute the QDD representing $\text{Post}_\theta(L)$: sending a message a adds a new final state and some transitions labeled by a to the automaton whereas receiving a message modifies the set of initial states (Figure 2.3). One can compose these modifications and obtain the QDD recognizing $\text{Post}_\theta(L)$.

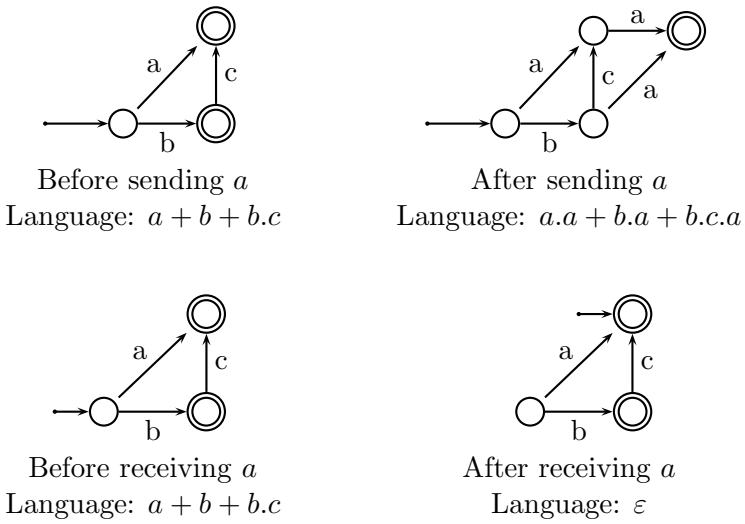


Figure 2.3: *Send* and *Receive* operations on a QDD with a single queue

The principle of the acceleration of QDDs is that multiple applications of Post_θ generate some periodicity in the states and transitions added by the "send" operations. Once a periodicity is detected, a loop is generated in the QDD, simulating the application of Post any number of times.

This algorithm also gives the constructive proof of the following theorem:

Theorem 2.1 *If θ is a sequence of operations affecting a single queue q_i, and L a regular language on Σ_i, then $\text{Post}_\theta(L)$ and $\text{Post}^*_\theta(L)$ are regular.*

Remark 2.2 *A QDD can represent any regular language built on the alphabet of a single queue, but not all regular languages built on the alphabet of several queues. For example, if $a \in \Sigma_1$ and $b \in \Sigma_2$, there is no QDD representing the regular language $(a\sharp b)^*$.*

Loops involving several channels. When there are several channels involved, there is no hope of having a similar result, since $\text{Post}^*_\theta(L)$ may be non-regular. For example, the sequence $\theta = 1!a, 2!b$ generates a non-regular set: $\text{Post}^*_\theta(\{\varepsilon\}) = \{a^n\sharp b^n | n \in \mathbb{N}\}$.

Definition 2.1 *Let θ_i be the sequence of operations of θ affecting only the queue q_i, $\theta_{i!}$ (resp. $\theta_{i?}$) the sequence of outputs (resp. inputs) of θ involving the queue i. θ_i is* counting *if either:*

- $|\Sigma_i| > 1$ *and* $|\theta_{i!}| > 0$,

- *or* $|\Sigma_i| = 1$ *and* $|\theta_{i!}| > |\theta_{i?}|$.

θ is counting if one of the θ_i's is counting.

Such a sequence is counting because there are some regular languages L such that we know how many times the loop was executed just by looking at the result of the execution, *i.e.* $\forall k, l \geq 0, \text{Post}^k_{\sigma_i}(L) \neq \text{Post}^l_{\sigma_i}(L) \implies k \neq l$.

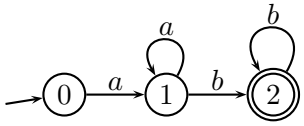
Example 2.2 *Let θ be a loop consisting in sending 2 messages a and receiving one message a in the first queue, and sending a message b in the second queue: $\theta = 1!a \cdot 1!a \cdot 1?a \cdot 2!b$. This loop is counting since it verifies the second condition. We can also see that, if we take $L = \{\varepsilon\}$, then $\text{Post}^k_\theta(L) = \{a^k \sharp b^k\}$, and so $\text{Post}^k_{\sigma_i}(L) \neq \text{Post}^l_{\sigma_i}(L) \implies k \neq l$.*

[BGWW97] gives an acceleration algorithm to compute $\text{Post}^*_\theta(L)$ if L is a regular language and θ is not a counting loop. There is no hope to do better with a QDD representation, because QDDs can only represent regular sets and $\text{Post}^*_\theta(L)$ may be non-regular if θ is a counting loop. If one wants to accelerate any loop, one must choose a more expressive representation, for example the CQDDs.

2.1.3 Constrained Queue-Contents Decision Diagrams

A Constrained Queue-Contents Decision Diagram (CQDD)[BH99] associates *simple automata*, which are finite automata with explicit loops, and Presburger formulas, the variables of which counting how many times each loop of the simple automata is taken.

Example 2.3 *A CQDD representing the set $\{a^n.b^n | n \geq 1\}$ is composed of the simple automaton depicted in Figure 2.4 and of the Presburger formula $x_{(1,a,1)} = x_{(2,b,2)}$.*



$$\mathcal{F} : x_{(1,a,1)} = x_{(2,b,2)}$$

Figure 2.4: Example of a CQDD

More formally, a simple automaton is a finite automaton $\langle \Sigma, Q, Q_0, Q_f, \delta \rangle$ where:

1. $Q = \{q_0, q_1, \ldots, q_m\} \cup \bigcup_{i=0}^{m} P_i$; The states $\{q_0, q_1, \ldots, q_m\}$ are the entry points of the loops: $q_i \xrightarrow{a_{i,1}} p_i^1 \xrightarrow{a_{i,2}} \ldots \xrightarrow{a_{i,l_i}} p_i^{l_i} \xrightarrow{a_{i,l_i+1}} q_i$. P_i is the set of states of loop number i, except q_i: $P_i = \{p_i^1, \ldots, p_i^{l_i}\}$,

2. $Q_0 = \{q_0\}$,
3. $Q_f = \{q_m\}$,
4. δ is composed of loop transitions (at most one loop for each q_i: $q_i \stackrel{a_{i,1}}{\rightarrow} p_i^1 \stackrel{a_{i,2}}{\rightarrow} \ldots \stackrel{a_{i,l_i}}{\rightarrow} p_i^{l_i} \stackrel{a_{i,l_i+1}}{\rightarrow} q_i$) and exactly one transition between each couple (q_i, q_{i+1}).

The loop on the final state q_m is special; it can be either a single loop of length greater than 1, or multiple loops of length 1. A simple automaton is *restricted* if there is no loop of length 1 on q_m.

The words accepted by a restricted simple automaton are of the form $u_1 v_1^* u_2 v_2^* \ldots u_m v_m^* u_{m+1}$ where u_i's and v_i's are non-empty words, except u_1 and u_{m+1} which may be empty.

A CQDD is given by a finite set of accepting components, each accepting component being:

1. a set of deterministic *restricted simple automata* (one for each queue of the CFSM);
2. a formula of Presburger arithmetics, with variables corresponding to the transitions of the automata.

Even if the variables of the Presburger formula can correspond to any transitions, they are only useful when each variable corresponds to one loop of a simple automaton, as shown in Example 2.3.

CQDDs can represent non-regular sets, like the previous example, and therefore can handle sets of configurations a QDD cannot represent. However, the opposite is also true. Some sets of configurations can be represented by a QDD, but not by a CQDD, because some regular sets are not representable by simple automata. For example, the set $(ab^*c)^*$ is a regular set a simple automaton cannot represent.

Representation of a set of configurations. Intuitively, each components of a CQDD recognize a set of configuration $(c, L_1, \ldots, L_N)$ sharing the same control point.

A set of configurations $(c, L_1, \ldots, L_N)$ sharing the same control point is accepted (resp. reverse accepted) by a CQDD if one of its components $\langle(\mathcal{A}_1, \ldots, \mathcal{A}_N), f\rangle$ recognizes the multi-language $(L_1, \ldots, L_N)$ (resp. $(L_1^r, \ldots, L_N^r)$). A set of configurations $\mathcal{C}$ is *CQDD representable* (resp. *CQDD reverse representable*) if there is a CQDD accepting (resp. reverse accepting) exactly the configurations of $\mathcal{C}$.

Acceleration with CQDDs. The CQDD representation relies on a Presburger formula constraining the values of the variables corresponding to the transitions of each automaton. If C is a set of configurations represented by a CQDD, computing $\text{Post}_\theta^*(\mathcal{C})$ is quite easy. We introduce a new parameter n, counting how many times the loop θ is executed. Let us consider a CQDD with N queues and a single accepting component $\langle(\mathcal{A}_1, \ldots, \mathcal{A}_N), f\rangle$. After n executions of θ, the set of reachable configurations is represented by N automata $(\mathcal{A}'_1, \ldots, \mathcal{A}'_N)$, with N constraints $g_1, \ldots, g_N$ expressing the

relation between the variables corresponding to the transitions of $\mathcal{A}_i$ and the ones of $\mathcal{A}'_i$, depending on n. Then $\text{Post}^*_\theta(\mathcal{C}) = \bigcup_{n\in\mathbb{N}} \text{Post}^n_\theta(\mathcal{C})$ is represented by the CQDD $\langle(\mathcal{A}_1,\ldots,\mathcal{A}_N), f \wedge \bigwedge_{i=1}^N g_i\rangle$, n being a free variable of the formula $f \wedge \bigwedge_{i=1}^N g_i$.

This forms the sketch of the proof of the following theorem:

Theorem 2.2 *[BGWW97] For every CQDD representable (resp. reverse representable) set of configurations C, and every loop θ of the control structure, the set of configurations* $\text{Post}^*_\theta(C)$ *(resp.* $\text{Pre}^*_\theta(C)$*) is CQDD representable (resp. reverse representable) and effectively constructible.*

2.1.4 Simply Regular Expressions

[ABJ98, AAB99] develop an acceleration technique for the analysis of lossy channel systems. We remind that a lossy channel system is a CFSM where message can be lost at any time in the queues. Queue contents are represented by *simply regular expressions*.

Definition 2.2 (Simply regular expressions) *An* atomic expression *over a finite alphabet Σ is a regular expression of the form:*

- $(a + \varepsilon)$ *with* $a \in \Sigma$*, or of the form*
- $(a_1 + a_2 + \cdots + a_m)^*$ *with* $m > 1$ *and* $\forall 1 \leq i \leq m, a_i \in \Sigma$

A product *p over Σ is a (possibly empty) concatenation $e_1 \cdot e_2 \cdot \cdots \cdot e_n$ of atomic expressions over Σ.*

A simple regular expression *(SRE) over Σ is a sum $p_1 + \cdots + p_n$ of products over Σ.*

Each SRE represents the content of one queue at a given control point. For a CFSM $\mathcal{M} = \langle C, \Sigma, c_0, \Delta\rangle$ with N queues, a symbolic state is a mapping of the form $c \mapsto (e_1, \ldots e_N)$, where $e_1, \ldots, e_N$ are SREs. Unlike the QDDs and CQDDs, this symbolic representation is a non-relational one, because one cannot link the content of one queue with the content of another one.

Despite this limitation, SREs are an effective representation of queue contents: there is a normal form, which can be computed in linear time. Moreover, entailment[3] among SREs can be computed in quadratic time, whereas checking language inclusion of two automata may be exponential [ABJ98].

Another advantage of SREs is that the set of configurations reachable from a given configuration c is representable by a SRE. However, this does not mean that one can always compute the SRE representing this set of reachable states. Indeed, [CFI96] demonstrated that there is no such algorithm in general.

This limitation explains why one should use a semi-algorithm based on acceleration. Acceleration with lossy channel systems is quite easy with an SRE representation, since it can handle any loop of the control structure.

[3] This operation checks the language inclusion.

Theorem 2.3 *[ABJ98] Let θ be a loop of the control structure and E a regular set of configurations represented by a SRE e. The SRE e' representing* $\mathrm{Post}^*_\theta(E)$ *can be computed in quadratic time.*

2.1.5 Semi-Linear Regular Expressions

[FIS03] gives a survey of acceleration techniques for CFSM and proposes its own representation of queue contents: the *Semi-Linear Regular Expressions* (SLREs). Like SREs, the SLREs represent regular sets of words with a particular type of regular expressions.

Definition 2.3 *Let Σ be a finite alphabet.*

1. *A* Linear Regular Expression *is a regular expression of the form $x_0.y_0^*.x_1.y_1^* \dots x_{n-1}.y_{n-1}^*.x_n$, where each x_i is a word on Σ and each y_i is a non-empty word on Σ.*

2. *A* Semi-Linear Regular Expression *is a finite sum of linear regular expressions.*

SLREs are expressive enough to represent languages that can be both represented by QDDs and CQDDs. According to [FIS03], this property indicates that SLREs are the core of known representations for FIFO channel contents.

The SLRE representation has the same property as QDDs: one can always accelerate a loop involving a single channel, and a loop involving several channels can be accelerated if and only if it is not counting.

Note that SLREs is not an extension of SREs; even if those two representations look similar, their expressive power is incomparable and SLREs are harder to manipulate: deciding the inclusion is a *co-NP complete* problem whereas its complexity is linear for the SREs.

2.2 Extrapolation of a Sequence of Automata

In this part, we present some techniques not directly aimed at the verification of CFSMs, but sharing the same idea of computing the limit of a sequence of symbolic values represented by some automata.

2.2.1 Verification of Pushdown Systems

A Pushdown System (PDS) is an automaton with a finite set of locations and an unbounded stack. Its semantics is given by an infinite transition system.

Definition 2.4 *A Pushdown System (PDS) $\mathcal{P}$ is a tuple $\langle P, P_0\Gamma, \Delta\rangle$ where:*

- *P is a finite set of control states (locations),*
- *P_0 is the set of initial location*
- *Γ is a finite stack alphabet,*

- $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ *is a finite set of rules.*

The rules of Δ are written $(q, \gamma) \hookrightarrow (q', v)$, with $q, q' \in P$, $\gamma \in \Gamma$ and $v \in \Gamma^*$.

In our context, one of the uses of Pushdown Systems is to model programs with procedure calls. In this case, the valuations of the global variables is encoded in the control structure P, and Γ is the set of the control points of the procedures. We label transition rules with actions, *i.e.* the instruction of the program in each control point. For example, the instruction a at control point p_0 (next control point is p_1) is modeled by the rule: $(x, p_0) \overset{a}{\hookrightarrow} (x', p_1)$, where x is the value of the global variables before the instruction a, and x' the value of the variables after the instruction a.

Example 2.4 *The procedure m (main) can either perform the action a and call the procedure p, or call the procedure q. The procedure p performs the action b and calls the procedure q. The procedure q performs the action c and stops (Figure 2.5). In this example, there is no global variable, so the PDS will have a single control location x_0.*

The three procedures are modeled by the following PDS $\mathcal{P} = \langle P, \Gamma, \Delta \rangle$:

- $P = \{x_0\}$
- $\Gamma = \{m_0, m_1, m_2, m_3, p_0, p_1, q_0, q_1\}$,
- Δ *is given by the rules:*

  1. *intra-procedure rules:*

$$\begin{array}{rcl} (x_0, m_0) & \overset{a}{\hookrightarrow} & (x_0, m_1) \\ (x_0, p_0) & \overset{b}{\hookrightarrow} & (x_0, p_1) \\ (x_0, q_0) & \overset{c}{\hookrightarrow} & (x_0, q_1) \end{array}$$

  2. *call rules:*

$$\begin{array}{rcl} (x_0, m_0) & \overset{call\ q}{\hookrightarrow} & (x_0, q_0.m_2) \\ (x_0, m_1) & \overset{call\ p}{\hookrightarrow} & (x_0, p_0.m_3) \\ (x_0, p_1) & \overset{call\ q}{\hookrightarrow} & (x_0, q_0.p_2) \end{array}$$

  3. *return rules:*

$$\begin{array}{rcl} (x_0, q_1) & \hookrightarrow & (x_0, \varepsilon) \\ (x_0, p_2) & \hookrightarrow & (x_0, \varepsilon) \end{array}$$

Semantics. A configuration of a PDS is given by a location and the content of the stack. At the beginning, the system is in a location p_0 with an empty stack. $\mathcal{P}$ defines an infinite transition system $\langle C, c_0, \rightarrow \rangle$ where $C = P \times \Gamma^*$, $c_0 = (p_0, \varepsilon)$ is the initial configuration and $\rightarrow$ is defined by the rule:

$$\frac{(q, \gamma) \overset{a}{\hookrightarrow} (q', v) \quad u \in \Gamma^*}{(q, \gamma.u) \overset{a}{\rightarrow} (q', v.u)}$$

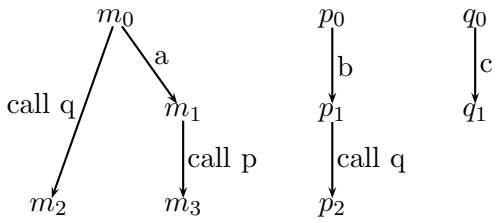


Figure 2.5: The three procedures m, p and q

$\mathcal{P}$-automata. A $\mathcal{P}$-automaton represents a regular set of configurations of the PDS $\mathcal{P}$. It is a finite automaton $\mathcal{A} = \langle Q, \Gamma, P, Q_f \rangle$, the initial states of which are the locations of $\mathcal{P}$. A configuration (p, w) belongs to the set represented by $\mathcal{A}$ if the word w is accepted by $\mathcal{A}$ starting from the initial state p.

This $\mathcal{P}$-automaton is a finite representation of a possibly infinite set of configurations. Moreover, if S is a regular set of configurations represented by $\mathcal{A}$, one can transform $\mathcal{A}$ so that it represents $\mathrm{Pre}^*(S)$, the set of predecessors of S. Another transformation gives the $\mathcal{P}$-automaton representing $\mathrm{Pre}^*(S)$.

Computation of $\mathrm{Pre}^*(S)$. Let S be a regular set of configurations represented by a $\mathcal{P}$-automaton. We want to compute the $\mathcal{P}$-automaton representing $\mathrm{Pre}(S)$. [BEM97] presents an algorithm for the automaton transformation. We give here the general idea of this algorithm.

If $(q, \gamma) \hookrightarrow (q', v)$ is a rule of the PDS $\mathcal{P}$ and $(q', v.u)$ is a configuration of S, then $(q, \gamma.u)$ is a configuration of $\mathrm{Pre}(S)$. Considering the automaton representing S, there is a run labeled by $v.u$, starting from the state q' and ending to some final state q_f. We split this run as $q' \xrightarrow{v}{}^* q'' \xrightarrow{u}{}^* q_f$. So, if we add a transition (q, γ, q''), then the word $\gamma.u$ is recognized starting from state q. We repeat this operation for all rules and all initial states, and we obtain an automaton representing $\mathrm{Pre}(S)$.

We iterate this operation and we obtain $\mathcal{P}$-automata representing $\mathrm{Pre}^2(S)$, $\mathrm{Pre}^3(S)$, etc, until there is $k \in \mathbb{N}$ so that $\mathrm{Pre}^k(S) = \mathrm{Pre}^{k+1}(S)$. The existence of this stabilization of the sequence is guaranteed because the number of states of the $\mathcal{P}$-automata is fixed, so we cannot add transitions indefinitely. We thus obtain the $\mathcal{P}$-automaton representing $\mathrm{Pre}^*(S)$.

Computation of $\mathrm{Post}^*(S)$. A $\mathcal{P}$-automaton representing $\mathrm{Post}^*(S)$ may be computed in a similar way. The only difference is that we may add some states before performing the algorithm. For each rule $(q, \gamma) \hookrightarrow (q', v)$ with $v = \gamma_1.\gamma_2 \ldots \gamma_n$, we add $n-1$ states to the automaton representing S: $q_1 \xrightarrow{\gamma_2} q_2 \xrightarrow{\gamma_3} \ldots \xrightarrow{\gamma_{n-1}} q_{n-1}$. Then if there is a configuration $(q, \gamma.u)$ in S, then there is a transition (q, γ, q'') in the $\mathcal{P}$-automaton recognizing S. We add two transitions (q, γ_1, q_1) and (q_{n-1}, γ_n, q''): the configuration $(q', v.u)$ of $\mathrm{Post}(S)$

is recognized by the new $\mathcal{P}$-automaton.

2.2.2 Transducers and Regular Model Checking

We consider a labeled transition system $\langle C, \Sigma, \rightarrow \rangle$, with $C = E^*$. A *transducer* is an automata encoding the transition relation $\rightarrow \subseteq E^* \times E^*$.

Definition 2.5 *A transducer is an automaton* $T = \langle Q, E, Q_0, Q_f, f : Q_f \rightarrow \Sigma, \Delta \rangle$ *where* f *maps each final state with a label* $\sigma \in \Sigma$ *and* $\Delta \subseteq Q \times (E \times E) \times Q$ *is the set of transitions*[4].

So a transducer is a finite automaton, with an alphabet $E \times E$ and a label in the set Σ attached to all final states. It encodes a set of transitions $\rightarrow \subseteq E^* \times \Sigma \times E^*$ as: $(w_i, \sigma, w_o) \in \rightarrow$ if there is a path of the transducer $q_0 \overset{a_{i,1}/a_{o,1}}{\rightarrow} q_1 \overset{a_{i,2}/a_{o,2}}{\rightarrow} \dots \overset{a_{i,n}/a_{o,n}q_n}{\rightarrow}$ with:

1. $q_0 \in Q_0$,
2. $w_i = a_{i,1}.a_{i,2} \dots a_{i,n}$,
3. $w_o = a_{o,1}.a_{o,2} \dots a_{o,n}$,
4. $q_n \in Q_f$ and $f(q_n) = \sigma$.

The labeling function f is optional, existing only when the transition system is labeled by Σ.

Example 2.5 *Some processes access a common resource via a token. One and only one process can have this token: each process has two states: 1 and 0, depending whether the process has the token or not. At the beginning, the first process has the token, and a transition of the system is informally: "a process can give the token to the following process". A configuration of this transition system is encoded by a word* $w \in \{0, 1\}^*$*, and the transition function by the transducer depicted in Figure 2.6.*

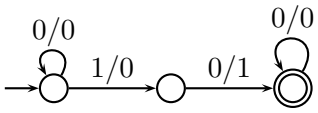


Figure 2.6: Example of a transducer

[4]We consider only length-preserving transducers: the input word and the output one have the same length.

Since the transducer T encodes the transition relation, seen as a relation $T \subseteq E^* \times E^*$, we want to have some algorithms operating on the transducer, computing:

- the reachability set $T^*(S)$, where S is a regular set of configurations,
- the transitive closure T^+.

The transitive closure of the relation T directly give the reachability set, but has other applications, like the detection of a loop ; a configuration c is part of a loop if $(c, c) \in T^+$.

Regular Model Checking. [BJNT00] solves both issues, with two different techniques:

- An automata theoretical construction of the transitive closure, in other words a representation of T^+.
- A widening based technique which aims at computing $T^*(S)$ for a regular set of configurations S.

Construction of T^+ The first technique relies on the construction of *column transducer*, the states of which are sequences of states of the original transducer $T = \langle Q, Q_0, Q_f, \delta \rangle$. The name "column transducer" comes from the construction $T^+ = \langle Q^+, Q_0^+, Q_f^+, \delta^+ \rangle$. The idea is the following: if two configurations $(w, w') \in E^* \times E^*$ belong to T^+, then there are $m+1$ configurations $w = w^0, w^1, \ldots, w^m = w'$ with $(w^i, w^{i+1}) \in T$. We write $w^i = a_1^i \ldots a_n^i$. Since (w^i, w^{i+1}) is accepted by the transducers, we can find m accepting runs of the original transducer, that define a single run of the column transducer:

$$\begin{array}{ccccccccc}
q_0^1 & \xrightarrow{(a_1^0,a_1^1)} & q_1^1 & \xrightarrow{(a_2^0,a_2^1)} & q_2^1 & \cdots & q_{n-1}^1 & \xrightarrow{(a_n^0,a_n^1)} & q_n^1 \\
q_0^2 & \xrightarrow{(a_1^1,a_1^2)} & q_1^2 & \xrightarrow{(a_2^1,a_2^2)} & q_2^2 & \cdots & q_{n-1}^2 & \xrightarrow{(a_n^1,a_n^2)} & q_n^2 \\
 & \vdots & \vdots & & & & \vdots & & \\
q_0^m & \xrightarrow{(a_1^{m-1},a_1^m)} & q_1^m & \xrightarrow{(a_2^{m-1},a_2^m)} & q_2^m & \cdots & q_{n-1}^m & \xrightarrow{(a_n^{m-1},a_n^m)} & q_n^m
\end{array}$$

The states of the column transducer are the columns $q_0^1 q_0^2 \ldots q_0^m$, $q_1^1 q_1^2 \ldots q_1^m$, ..., $q_n^1 q_n^2 \ldots q_n^m$, and the transitions are defined as $q_{i-1}^1 \ldots q_{i-1}^m \xrightarrow{(a_i^0.a_i^m)} q_i^1 \ldots q_i^m$ if there are m transitions of the original transducer $q_{i-1}^1 \xrightarrow{(a_i^0,a_i^1)} q_i^1$, $q_{i-1}^2 \xrightarrow{(a_i^1,a_i^2)} q_i^2$, ... , $q_{i-1}^m \xrightarrow{(a_i^{m-1},a_i^m)} q_i^m$. In this example, the accepting run is thus $q_0^1 \ldots q_0^m \xrightarrow{(a_1^0.a_1^m)} \cdots \xrightarrow{(a_n^0.a_n^m)} q_n^1 \ldots q_n^m$.

This definition, however, may generate an infinite number of states and transitions. One can attempt to "minimize" this transducer, but without guarantee of termination in general. [JN00] gives a sufficient condition of termination: if T has *local depth* k (it never needs to rewrite any letter of a word more than k times to relate two words), then the transitive closure T^+ can be recognized by a transducer of size exponential

in k. Since the CFSMs are Turing-powerful, we can always find, for any bound k, a CFSM generating a transducer that has not local depth k.

We now focus on the second technique, which relies on a regular extrapolation of the growth of a set of configurations.

Extrapolation technique. Let $S \subseteq E^*$ be a set of configurations and T a regular relation represented by a transducer. The problem is how to guess an over-approximation of $T^*(S)$ in a few computation steps. The principle of this extrapolation is to compare S and $T(S)$ in order to identify a growth of the set of configurations, and then repeat this growth.

The extrapolation assumes that the two conditions are satisfied:

- (C1): $S = S_1.S_2$ and $T(S) = S_1.I.S_2$,
- (C2): $S_1.I^*.S_2 = T(S_1.I^*.S_2) \cup S$.

The first condition means that we can identify an increment in $T(S)$, compared to S. The second condition ensures that $S_1.I^*.S_2$ is a fix-point of the function $X \mapsto T(X) \cup S$, thus $T^*(S) \subseteq S_1.I^*.S_2$.

The extrapolation algorithm simply consists in determining, in the automaton representing $T(S)$, the three parts S_1, I and S_2, then adding transitions to repeat the increment part and finally checking whether the modified automaton satisfies the second condition. Some heuristics help the identification of the three parts of the automaton, like cutting the automaton between two strongly connected components.

This extrapolation is sometimes exact, *i.e.* $T^*(S) = S_1.I^*.S_2$: if T is a *simple relation* [BJNT00] and C1 and C2 hold, then $T^*(S) = S_1.I^*.S_2$.

Abstract Regular Model Checking. The two methods presented here work if the relation encoded by the transducer T is a simple relation or has a local depth k, but their complexity may be exponential or relies on heuristics to find an increment in the transducer. The use of some finite abstractions can reduce the complexity of the algorithm, if one do not worry about the exactness of the analysis. [BHV04] proposes to collapse the states of a transducer according to an equivalence relation. The equivalence relation must have a finite number of equivalence classes; this bound on the number of states of the collapsed automaton ensures that the computation will eventually terminate. The paper proposes two equivalence relations:

1. One based on a finite set of predicates $\mathcal{P}$. Each predicate $P \in \mathcal{P}$ defines a language $L(P)$. For any state q, $L(T, q_1)$ is the language the transducer T would recognize if q was its unique initial state. Two states q_1 and q_2 of the transducer T are equivalent if for any predicate P, $(L(P) \cap L(T, q_1) \neq \emptyset) \Leftrightarrow (L(P) \cap L(T, q_2) \neq \emptyset)$.
2. The other one based on finite-length languages. Two states q_1 and q_2 are equivalent if the languages the transducer T would recognize if q_1 (resp. q_2) were its unique initial state have the same set of words of length lesser or equal to a parameter k. In other words, q_1 and q_2 are equivalent if: $L^{\leq k}(T, q_1) = L^{\leq k}(T, q_2)$.

Both abstractions are parametric and may be applied to classical finite automata instead of transducers, and are an alternative to the operator ρ_k we present in Chapter 3.

2.2.3 Extrapolation of an Increasing Sequence of Transducers

[BLW03] proposes an extrapolation method for transducers which can be applied to finite automata. This extrapolation applies to an increasing sequence of transducers, in terms of language inclusion, and relies on two equivalence relations, one forward, the other backward, between the states of a transducer T^i and its successor T^{i+1}.

The principle of this extrapolation is to detect which part of T^{i+1} can be seen as an increment to T^i. Assuming this increment can be repeated infinitely often, one may add transitions to capture this regular incrementation. The detection of this incremental part of the automaton is determined by some equivalence relation between the states of T^i and the states of T^{i+1}.

Equivalence relations. We consider a sequence of deterministic and minimal transducers $T^0, T^1, \ldots$ with $T^i = \langle Q^i, \Sigma \times \Sigma, q_0^i, Q_f^i, \delta^i \rangle$. The forward equivalence relation $E_f^i \subseteq Q^i \times Q^{i+1}$ is defined by the language recognized by T^i and T^{i+1} when their initial state is altered: $(q, q') \in E_f^i$ if $\widetilde{T^i} = \langle Q^i, \Sigma \times \Sigma, \{q\}, Q_f^i, \delta^i \rangle$ and $\widetilde{T^{i+1}} = \langle Q^{i+1}, \Sigma \times \Sigma, \{q'\}, Q_f^{i+1}, \delta^{i+1} \rangle$ recognize the same language.

The backward equivalence relation $E_b^i \subseteq Q^i \times Q^{i+1}$ is defined by the language recognized by T^i and T^{i+1} with a single final state: $(q, q') \in E_b^i$ if $\widetilde{T^i} = \langle Q^i, \Sigma \times \Sigma, q_0^i, \{q\}, \delta^i \rangle$ and $\widetilde{T^{i+1}} = \langle Q^{i+1}, \Sigma \times \Sigma, q_0^{i+1}, \{q'\}, \delta^{i+1} \rangle$ recognize the same language.

Definition 2.6 (incrementally larger) *T^{i+1} is* incrementally larger *than T^i if the two relations E_f^i and E_b^i cover all states of T^i: $\forall q \in Q^i, \exists q' \in Q^{i+1}$ such that $(q, q') \in E_f^i \bigvee (q, q') \in E_b^i$.*

Note that, in this case, the unicity of the state q' can be proved since both T^i and T^{i+1} are minimal, the equivalence relation thus defines two functions, also noted E_b^i and $E_f^i : Q^i \to Q^{i+1}$. In the following, we assume that for each i, T^{i+1} is incrementally larger than T^i.

Identification of the different parts of the transducer. With this condition, we note Q_1^i the set of states of Q^i covered by E_f^i and Q_2^i the other states, which are covered by E_b^i since Q^{i+1} is incrementally larger than Q^i. The two functions E_b^i and E_f^i, applied to the sets Q_1^i and Q_2^i, define what the increment part of the transducer T^{i+1} is. Precisely, we have a partition of Q^{i+1} of size three defining an increment:

- The head part $Q_H^{i+1} = E_b^i(Q_2^i)$,
- The tail part $Q_T^{i+1} = E_f^i(Q_1^i) \setminus Q_H^{i+1}$,
- The increment part Q_I^{i+1}(the other states).

We now consider T^{i+2} ; a similar partition of size four is obtained by:

- $Q_H^{i+2} = E_b^i(Q_H^{i+1})$,
- $Q_T^{i+2} = E_f^i(Q_T^{i+1})$,
- $Q_{I_0}^{i+2} = E_b^{i+1}(Q_I^{i+1})$,
- $Q_{I_1}^{i+2} = E_f^{i+1}(Q_I^{i+1})$[5].

Each computation step generates a new increment part: T^{i+j} has a head part Q_H^{i+j}, a tail part Q_T^{i+j} and j increment parts $Q_{I_0}^{i+j} \dots Q_{I_{j-1}}^{i+j}$.

Extrapolation. The principle of the extrapolation is to add transitions from $Q_H^{i+j} \cup Q_{I_0}^{i+j}$ to other increments parts ; if there is a transition (q, a, q') with $q \in Q_H^{i+j} \cup Q_{I_0}^{i+j}$ and $q' \in Q_{I_k}^{i+j}$, then one adds k transitions from q labeled by a to states corresponding to q'[6] in each increment parts Q_l^{i+j} with $0 \leq l \leq k$. Note that if $q \in Q_{I_0}^{i+j}$, then the extrapolation adds a self-loop on q. We denote by T_e^* this extrapolated transducer.

Example 2.6 *Let us consider some processes $P^0, P^1, P^2, \dots$ sharing a common resource. Access to this resource is modeled by a token. At the beginning, P^0 has access to the resource, then it gives the token to P^1, then P^1 gives the token to P^2 etc. We consider the following encoding: a global state of this system is modeled by a finite word $w = x_0 \dots x_n \in \{0,1\}^*$, $x_i = 1$ if P^i has the token and $x_i = 0$ otherwise.*

At the beginning, the global state is thus $1000\dots$. The token then goes from one process to the following: $0100\dots$, $0010\dots$, etc. The transition function is represented by the initial transducer T^0 depicted on Figure 2.7. The computation of the transitive closure of this transition function is the computation of the sequence of transducers $T^0, T^1 = T^0 \cup T^0 \circ T^0, T^2 = T^1 \cup T^0 \circ T^1, \dots, T^{i+1} = T^i \cup T^0 \circ T^i, \dots$. Without extrapolation, this computation does not terminate if the number of processes is unbounded.

The two first steps of the computation, T^1 and T^2, are depicted on Figure 2.7. The extrapolation of T^2 gives the least upper bound of the sequence, T_e^*.

Correctness. This extrapolation, which consists in adding transitions and thus repeating increments, is safe (*i.e.* $L(\bigcup_{i\geq 0} T^i) \subseteq L(T_e^*)$) if the extrapolated transducer T_e^* satisfies : $L(T_e^* \circ T_e^*) \subseteq L(T_e^*)$ [BLW03].

Adaptation to the Analysis of FIFO channel systems. Even if we do not use a transducer to model the transitive function induced by a CFSM, the same extrapolation may give an over-approximation of the limit of an increasing sequence of automata representing the queue contents, defined recursively by $\mathcal{A}_{i+1} = \text{Post}(\mathcal{A}_i) \cup \mathcal{A}_i$, assuming the function Post is monotonic.

[5] The actual partition is indeed defined by removing from $Q_{I_0}^{i+2}$ the states belonging to Q_H^{i+2}, etc.

[6] this correspondence is built during the detection of the increment. For example,if $k = 1$ the state corresponding to $q' \in Q_{I_1}^{i+j}$ in $Q_{I_0}^{i+j}$ is $E_b^{i+j}((E_b^{i+j})^{-1}(q'))$.

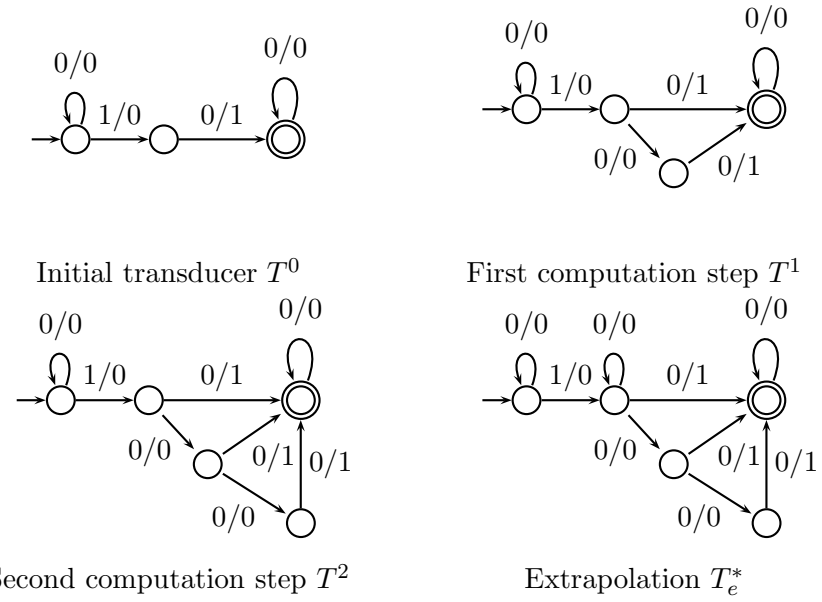


Figure 2.7: Computation of the transducers

2.2.4 Verification of Linear Networks of Processes

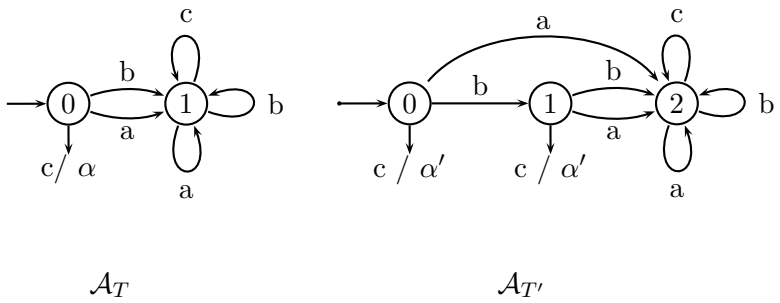
[LHR97], which analyzes linear networks of processes, introduces another widening-like operator for finite automata. The goal of that paper is the model checking of regular networks of processes of unknown sizes. Each process is a finite-state machine. The global state of a network of size k is represented by a word of size k, each letter representing the state of a process. Regular sets of states are thus represented by regular languages.

A synchronous observer is here a Mealy machine (a finite automaton with outputs) that can emit an error signal. Transitions are labeled by an input letter and an output, which can be either nothing or an error signal α. The observer defines a regular set of traces violating a safety property.

The paper focuses on under-approximations of greatest fix-points in the lattice of sets of regular traces, and defines an extrapolation operator which works like a widening operator, trying to guess the limit of a decreasing sequence of sets of traces $T_0 \supseteq T_1 \supseteq \cdots \supseteq T_n \supseteq \ldots$.

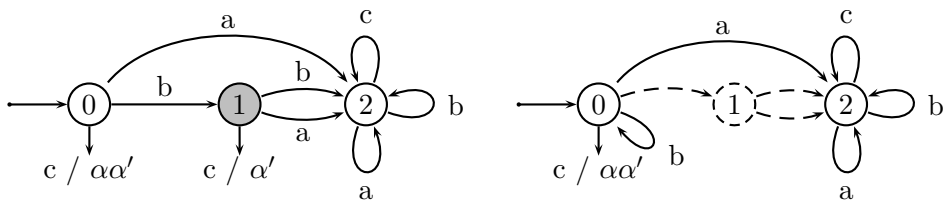
The principle of this binary operator is the following. For two sets of traces $T \supseteq T'$, the synchronous observer $\mathcal{A}_{T'}$ recognizing T' emits an error signal more often than $\mathcal{A}_T$, the one recognizing T. In some states of the observer $\mathcal{A}_{T\times T'}$, the error signal can be emitted by $\mathcal{A}_{T'}$ but not by $\mathcal{A}_T$. Theses states are the parts of the automata that change from one step to another. The extrapolation operator anticipates the future changes by suppressing those states and redirecting the transitions leading to one of the deleted states to a state where both automata can emit an error signal. This operation generates some loops in the automata, as shown in the following example.

Example 2.7 *Let $T = (a + b)(a + b + c)^*$ and $T' = (a + b(a + b))(a + b + c)^*$ be two regular sets of traces such that $T \supseteq T'$. Observers recognizing T and T' are depicted on Figure 2.8, and the synchronous observer $\mathcal{A}_{T\times T'}$ is depicted on Figure 2.9. The state* **1** *of $\mathcal{A}_{T\times T'}$ is a state where $\mathcal{A}_{T'}$ can emit an error message on a c input whereas $\mathcal{A}_T$ cannot. This state is therefore removed, and the transition leading to that state, labeled by b is rerouted to the state* **0** *where both $\mathcal{A}_T$ and $\mathcal{A}_{T'}$ can emit an error signal α Figure 2.9.*



$\mathcal{A}_T$ $\mathcal{A}_{T'}$

Figure 2.8: Observers $\mathcal{A}_T$ and $\mathcal{A}_{T'}$



$\mathcal{A}_{T\times T'}$ $\mathcal{A}_{T\nabla T'}$

Figure 2.9: Synchronous product $\mathcal{A}_{T\times T'}$ and widening $\mathcal{A}_{T\nabla T'}$

Application to the extrapolation of a sequence of automata. This work is not aimed at the verification of FIFO channel systems, however it introduces an extrapolation operator which can also be employed on a synchronous observer representing queue contents.

This observer is defined like an automaton recognizing the queue content L, and outputs the error signal when a word is in L. However, this assumes that L is prefix-closed, a false assumption since the language of a queue is not prefix-closed. This is the reason why this approach is not adapted to the kind of systems we focus on in the following chapters.

2.3 Conclusion

The first section of this chapter was a presentation of the acceleration techniques for the analysis of FIFO channel systems. Those techniques, using symbolic representations like QDDs, CQDDs, SREs or SLREs, have a common drawback: even when they can compute $\mathrm{Post}^*_\theta(S)$ for any loop θ and any set of configurations S, the computation of the set of the reachable states may not terminate.

We want to keep a representation of the queue contents by an automaton, but with a guarantee that the reachability analysis terminates. So we looked at other techniques, not directly aimed at the analysis of CFSMs, but designed to compute the limit of a sequence of automata.

[BJNT00] presented two techniques to obtain the limit of a sequence of transducers: one is exact, but does not terminates for any system modeled by a CFSM. The other is an extrapolation which consists in guessing what part of the transducer represent an increment, and then in adding transitions so that this increment can be repeated infinitely often.

A similar idea was developed in [BLW03]: the authors detect the increment parts by partitioning the states of each transducer, and then add loops in the transducer.

In the next chapter, we will develop a slightly different approach. Instead of detecting an increment and repeating it, we use a quotient operation, which merges some states of the automata according to an equivalence relation, as in [BHV04]. We will explain how this simple operation is sufficient to obtain a regular over-approximation of the limit of the sequence of automata.

Chapter 3

Regular Languages as an Abstract Lattice

The previous chapter presented the acceleration techniques for the verification of FIFO channel systems, and their limitations due to the undecidability issue. In this chapter, we propose an approximated reachability analysis of FIFO channel systems. When analyzing CFSMs, this consists in replacing in dataflow equations, sets of FIFO channel configurations by an element of an abstract lattice. Such a transformation results in conservative approximations: we will be able to prove that a safety property is satisfied, or that a state is non-reachable, but not to prove that a property is not satisfied or that a state is effectively reachable.

The abstractions we propose in this chapter are based on regular languages, which exhibit, among nice properties, the closure under all Boolean operations, and a canonical representation: the deterministic and minimal finite automata (MDAs).

All standard operations on sets, such as language union, intersection or inclusion are performed on automata. Moreover, we need a widening operator ∇, since this lattice is of infinite height; this operator is also performed on automata. We discuss the properties and the effects of ∇, with some examples. Finally, we apply this analysis on some examples of communication protocols and compare our results with acceleration techniques.

3.1 The Lattice of Regular Languages

The representation of queue contents of CFSMs in terms of languages was discussed in the previous chapter. We focus here on the abstraction of queue contents in terms of regular languages. We remind that the set of regular languages over an alphabet Σ, $\mathrm{Reg}(\Sigma)$, with the partial order defined by the inclusion, is a lattice where:

- the least upper bound is the union of two regular languages,
- the greatest lower bound is the intersection of two regular languages,
- the bottom element is the empty language,

- the top element is the regular language Σ^*.

3.1.1 Classical Operations and Abstraction

As mentioned in Section 1.5, regular languages are represented by finite automata. The minimal deterministic finite automata recognizing a language L is its canonical representation. The operation $\cup, \cap$ as well as the inclusion test $\subseteq$ are performed on the automata, and their complexity is recalled in Figure 3.1, assuming the automata have n states and m transitions. The result of an union is a non-deterministic automaton and the result of the intersection is deterministic, but not minimal (assuming the original automata are deterministic and minimal). Figure 3.1 also presents two operations needed for the definition of the abstract semantics of a CFSM: the left derivation and the right concatenation.

The *left derivation* [Brz64] of a regular language L is performed w.r.t. a letter a:

$$L/a = \{w \in \Sigma^* | a.w \in L\}$$

It defines the abstract semantics of an input transition of a CFSM. The result is a non-deterministic automaton.

The *right concatenation* of a regular language L and a letter a is the language:

$$L.a = \{w.a | w \in L\}$$

It defines the abstract semantics of an output transition of a CFSM. The result is a deterministic automaton (assuming the original automaton was deterministic and minimal) but not minimal.

Algorithm	Worse Case Complexity (time)
inclusion test [1]	$O(2^n)$
inclusion test [2]	$O(n \log n)$
determinization	$O(2^n)$
minimization[3]	$O(n \log n)$
union[4]	$O(n+m)$
intersection	$O(n^2+m^2)$
left derivation[4]	$O(n+m)$
right concatenation[4]	$O(n+m)$

Figure 3.1: Complexity of the operations on finite automata

[1] General case,
[2] If the input automata are in canonical form,
[3] The input automaton is assumed to be deterministic,
[4] The complexity is due to the building of a new automaton, not the operation itself.

Those operations are similar to the ones for QDDs, presented in Chapter 2. In this chapter, we also investigate how one can abstract a queue content L by a regular

language $L' \supseteq L$. However, there is no real Galois connection since the abstract lattice is not complete.

Proposition 3.1 $(\mathrm{Reg}(\Sigma), \subseteq)$ *is of infinite height, and is not complete.*

Proof: Let us consider the sequence of regular languages $(L_n)_{n\in\mathbb{N}}$:

$$\forall n \geq 0,\ L_n = \bigcup_{k\leq n} \{a^k.b^k\}$$

Since $\forall n \geq 0,\ L_n \subseteq L_{n+1}$, this sequence is an infinite increasing chain. Thus $(\mathrm{Reg}(\Sigma), \subseteq$) is of infinite height. Moreover,

$$\bigcup_{n\in\mathbb{N}} L_n = \{a^i.b^i \mid i \in \mathbb{N}\}$$

is not a regular language; so $(\mathrm{Reg}(\Sigma), \subseteq)$ is not complete. □

This example also raises the issue of the existence of an abstraction function, the decreasing languages sequence $L_k = \varepsilon + a.b + a^2.b^2 + \cdots + a^k.b^k + \cdots + a^k.a^*.b^k.b^*$ is a sequence of regular over-approximation of the non-regular language $(a^n.b^n)$, which is the limit of this sequence; so there is no "best regular over-approximation" for this language.

This is the reason why we employ the representation framework, presented in Section 1.4, instead of a Galois connection:

$$(\mathcal{L}(\Sigma), \subseteq) \overset{\gamma}{\longleftarrow\!\!\!\longrightarrow} (\mathrm{Reg}(\Sigma), \subseteq)$$

with the concretization function $\gamma = Id$.

Prop. 3.1 also justifies the definition of a widening operator, because of the infinite height of this abstract lattice.

3.1.2 The Widening Operator ∇_k

Principles of this widening operator. Since a regular language has a canonical automaton representation, we can define the widening operator as an automata operation. In [Fer01], a widening operator for regular languages was mentioned but not developed. We adapted this operator to regular languages representing the content of a FIFO channel. Because of the FIFO operations, the widening operator should remain precise for both the beginning and the end of the queue.

The main idea is to merge some "equivalent" states (w.r.t. a particular equivalence relation) of the automaton, obtaining a new automaton recognizing a greater language, with a known bound on the number of states. This ensures that this operator respects the definition of a widening operator.

Definition of the equivalence relation. This equivalence relation may be viewed as the refinement of an initial equivalence relation given by a color function. The refinement relies on a bounded bisimulation relation.

Definition 3.1 (Bisimulation of depth k) *Let $(Q, \Sigma, Q_0, Q_f, \delta)$ be a minimal deterministic automaton, and* $\mathrm{col} : Q \to [1..N_{\mathrm{col}}]$ *a color function defining an equivalence relation $q_1 \approx_{\mathrm{col}} q_2 \Leftrightarrow \mathrm{col}(q_1) = \mathrm{col}(q_2)$. For $k \geq 0$, the bisimulation of depth k finer than $\approx_{\mathrm{col}}$ is defined inductively by: $\forall q_1, q_2 \in Q$,*

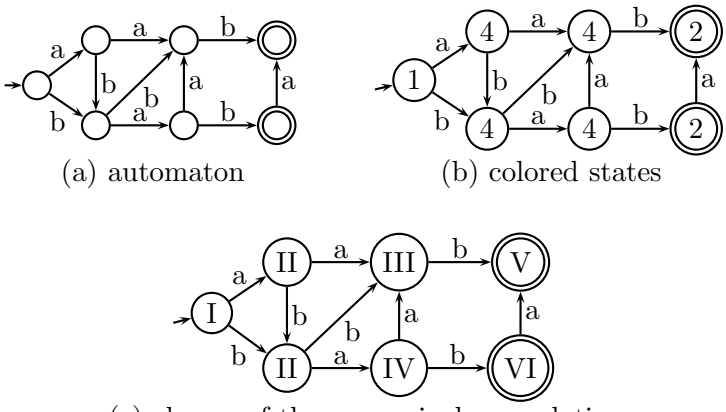
$$q_1 \approx_0^{\mathrm{col}} q_2 \quad \textit{iff} \quad q_1 \approx_{\mathrm{col}} q_2$$

$$q_1 \approx_{k+1}^{\mathrm{col}} q_2 \quad \textit{iff} \quad \begin{cases} q_1 \approx_k^{\mathrm{col}} q_2 \\ \forall a \in \Sigma, \forall q_1' \in Q,\ q_1 \xrightarrow{a} q_1' \implies \exists q_2' \in Q :\ q_2 \xrightarrow{a} q_2' \wedge q_1' \approx_k^{\mathrm{col}} q_2' \\ \forall a \in \Sigma, \forall q_2' \in Q,\ q_2 \xrightarrow{a} q_2' \implies \exists q_1' \in Q :\ q_1 \xrightarrow{a} q_1' \wedge q_1' \approx_k^{\mathrm{col}} q_2' \end{cases}$$

In the sequel, we consider the following *standard color function*, which separates initial and final states from other states:

$$\mathrm{col}(q) = \begin{cases} 1 \text{ if } q \in Q_0 \cap Q_f, \\ 2 \text{ if } q \in Q_f \setminus Q_0, \\ 3 \text{ if } q \in Q_0 \setminus Q_f, \\ 4 \text{ otherwise} \end{cases}$$

This standard color function ensures that initial states and final states cannot be merged by a quotient operation. This operation was defined in Section 1.5, and defines an operator ρ_k.



(a) automaton (b) colored states

(c) classes of the $\approx_2$ equivalence relation

Figure 3.2: Example of the equivalence classes of $\approx_2$

Definition and properties of the operator ρ_k^{col}. This operator is simply the quotient of an automaton by the equivalence relation $\approx_k^{\mathrm{col}}$.

Definition 3.2 (Operator ρ_k^{col}) *Given a bisimulation relation $\approx_k^{\text{col}}$ of depth k the operator $\rho_k^{\text{col}} : \text{Reg}(\Sigma) \to \text{Reg}(\Sigma)$ is defined by quotienting $\mathcal{M}(L)$, the MDA of L:*

$$\rho_k^{\text{col}}(L) = \mathcal{L}(\mathcal{M}(L)/\approx_k^{\text{col}})$$

ρ_k^{col} is extensive, *i.e.* $L \subseteq \rho_k^{\text{col}}(L)$, as being defined by a quotient automaton, and it is idempotent as a consequence of $\approx_k^{\text{col}}$: since each state of $\mathcal{M}(L)/\approx_k^{\text{col}}$ corresponds to an equivalence class of $\approx_k^{\text{col}}$, this automaton equals to its quotient by $\approx_k^{\text{col}}$.

As $\approx_{k+1}^{\text{col}}$ is a refinement of $\approx_k^{\text{col}}$, we also have $\forall L \in \text{Reg}(\Sigma) : \rho_{k+1}^{\text{col}}(L) \subseteq \rho_k^{\text{col}}(L)$.

Note that ρ_k is a particular case of ρ_k^{col}, with *col* being the standard coloring function.

Definition of the widening operator ∇_k. This operator is the application of the previous operator ρ_k^{col} to the union of two languages:

Definition 3.3 (Widening operator ∇_k^{col}) *Given an integer $k \geq 0$ and a color function* col*, we define a binary operator $\nabla_k^{\text{col}} : \text{Reg}(\Sigma) \times \text{Reg}(\Sigma) \to \text{Reg}(\Sigma)$:*

$$L_1 \nabla_k^{\text{col}} L_2 \triangleq \rho_k^{\text{col}}(L_1 \cup L_2)$$

Proposition 3.2 *∇_k^{col} is a widening operator for $\text{Reg}(\Sigma)$ in the sense of [CC92]:*

1. *$L_1 \cup L_2 \subseteq L_1 \nabla_k^{\text{col}} L_2$;*
2. *For any increasing chain $(L_0 \subseteq L_1 \subseteq \dots)$, the increasing chain defined by $L'_0 = L_0$, $L'_{i+1} = L'_i \nabla_k^{\text{col}} L_{i+1}$ is not strictly increasing (it stabilizes after a finite number of steps).*

∇_k is just a particular case of ∇_k^{col}, with *col* being the standard coloring function.

Proof: Since ρ_k^{col} is extensive, we have $L_1 \cup L_2 \subseteq L_1 \nabla_k^{\text{col}} L_2$. We just have to prove that $\text{card}(Q/\approx_k^{\text{col}})$ is lesser or equal to a constant depending on k and $|\Sigma|$. Thus the set $\{\rho_k^{\text{col}}(L) \,|\, L \in \text{Reg}(\Sigma)\}$ is finite.

For a state q, we consider the "execution tree of depth k":

1. each node is labeled with a triple (a, x, i), with $a \in \Sigma \cup \{-\}$, $x \in Q$ and $i \geq 0$;
2. the root is labeled with $(-, q, k)$;
3. if there is a node labeled with (l, x, i), with $i > 0$, $l \in \Sigma$ and a transition $(x, a, y) \in \delta$, we create a son labeled by $(a, y, i-1)$.

We color a node labeled by (a, x, i) with $\text{col}(x)$. According to the definition of $\approx_k^{\text{col}}$, two states q_1 and q_2 are in the same equivalence class if they have the same colored execution tree. So there are as many equivalence classes as colored execution trees.

Since the maximum branching degree of this tree is $|\Sigma|$, there are at most $2^{|\Sigma|^k}$ uncolored trees (proof by induction on k), each tree has at most $|\Sigma|^{k+1}$ nodes, so each tree can be colored in at most $N_{\text{col}}^{|\Sigma|^{k+1}}$ ways. Thus, we have $\text{card}(Q/\approx_k^{\text{col}}) \leq N_{\text{col}}^{|\Sigma|^{k+1}} \times 2^{|\Sigma|^k}$. □

Remark 3.1 *When the notion of widening operator was introduced in the seventies [CC77a], the idea was to guess the limit of a post-fix-point computation. However, our "widening operator" ∇_k doesn't try to guess the limit of a sequence of languages. Even if ∇_k satisfies the mathematical properties of a widening operator, one may argue that it must be considered as an upper bound operator rather than a genuine widening operator.*

3.1.3 Effects of the Widening Operator

With the formal definition given above, the reader may have some difficulties to fancy the effects of the widening operator. Let L be a regular language describing a queue content, k a given integer. We want to give an intuition of the shape of the language $\rho_k(L)$. We illustrate by some examples the approximations we make.

Let us consider $\mathcal{M}(L)$ the minimal deterministic automaton that recognizes the language L. For each example, we depict:

1. $\mathcal{M}(L)$,
2. the partition of states generated by $\approx_k$,
3. the quotient automaton $\tilde{\mathcal{M}}_L$ recognizing $\rho_k(L)$.

Number of messages. The multiple repetition of the same message $aa\ldots a$ may be over-approximated by an infinite repetition of the same message a^*. If the analyzed system allows arbitrary-long channel contents, this approximation can guess the limit of the fix-point computation.

Example 3.1 *The language $L = \{aaaa\}$ is recognized by the automaton $\mathcal{M}(L)$ depicted in Figure 3.3. The two states marked with an X (Figure 3.4) are in the same equivalence class (with $k = 1$), so they are merged. Here, the effect of the operator ρ_k is to create a loop: $\rho_1 L = \{a^i | i \geq 3\}$ (Figure 3.5). In this example, the information we loose is the number of messages 'a'.*

Sum of languages. If $L = L_1 + L_2$, the widening operator may merge some words of L_1 with words of L_2. This approximation may simplify the automaton.

Example 3.2 *The language $L = \{aac + bad\}$ is recognized by the automaton $\mathcal{M}(L)$ depicted in Figure 3.6. The two states marked with an X (Figure 3.7) are in the same equivalence class (with $k = 1$), so they are merged. The result is the language $\nabla_1 L = \{(a+b)a(c+d)\}$ (Figure 3.8. In this example, we loose the property "if we have an 'a' at the beginning of the queue, then we have a 'c' at the end, else a 'd' ".*

Suffixes and prefixes. Due to the FIFO structure of the communication channels, our approximation must be more precise at the beginning and the end of the words representing the queue-contents. The following proposition guarantees that the prefixes and suffixes are preseserved when applying ρ_k to a language.

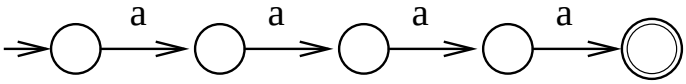


Figure 3.3: Automaton recognizing $L = \{aaaa\}$

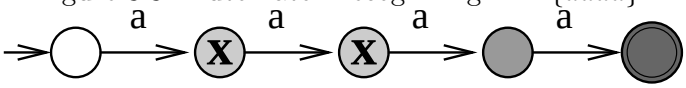


Figure 3.4: Equivalence classes of $\approx_1$

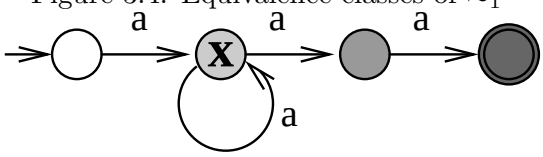


Figure 3.5: Automaton recognizing $\rho_1(L) = \{aaaa^*\}$

Proposition 3.3 *Let L be a regular language and k a fixed integer. Then L and $\rho_k(L)$ have the same set of prefixes of length 1 and the same set of suffixes of length less than or equal to k.*

Proof: Let $\mathcal{M} = (Q, Q_0, Q_f, \delta)$ be the minimal deterministic automaton recognizing L and $(\tilde{Q}, \Sigma, \tilde{Q_0}, \tilde{Q_f}, \tilde{\delta})$ the quotient automaton. Since $L \subseteq \rho_k(L)$, the prefixes (and suffixes) of L are also prefixes (and suffixes) of $\rho_k(L)$. We must prove the converse.

If $a \in \Sigma$ is a prefix of length 1 of a word of $\rho_k(L)$, then there is $\overline{q_0} \in \tilde{Q_0}$ $\overline{q} \in \tilde{Q}$ such that $(\overline{q_0}, a, \overline{q}) \in \tilde{\delta}$. The definition of the standard color function ensures that q_0 is also an initial state of $\mathcal{M}$, so a is a prefix of length 1 of L.

To finish the proof, we prove by induction that, if there is a sequence $\overline{q_1} \stackrel{a_1}{\to} \overline{q_2} \stackrel{a_2}{\to} \ldots \stackrel{a_l}{\to} \overline{q_f}$ allowed by the quotient automaton, with $\overline{q_f} \in \tilde{Q_f}$ and $l \leq k$, then there are $l+1$ states $q_1 \in \overline{q_1}, q_2 \in \overline{q_2}, \ldots, q_f \overline{q_f}$ such that: $q_1 \stackrel{a_1}{\to} q_2 \stackrel{a_2}{\to} \ldots \stackrel{a_k}{\to} q_f$ is allowed by $\mathcal{M}$ (obvious for $k = 0$).

Let $\overline{q_1} \stackrel{a_1}{\to} \overline{q_2} \stackrel{a_2}{\to} \ldots \stackrel{a_l}{\to} \overline{q_f}$ be a sequence verifying the conditions. Then there are l states $q_2 \in \overline{q_2}, \ldots, q_f \in \overline{q_f}$ such that: $q_2 \stackrel{a_2}{\to} \ldots \stackrel{a_l}{\to} q_f$ is allowed by $\mathcal{M}$ (induction hypothesis).

Moreover, $\exists q_1' \in \overline{q_1}, q_2' \in \overline{q_2}$, $q_1' \stackrel{a_1}{\to} q_2'$ (definition of the quotient automaton). So we have two states $q_2 \approx_k q_2'$ According to the definition of $\approx_k$, since $|a_2 \ldots a_l| \leq k$, $\exists q_3' \in \overline{q_3} \ldots q_f' \in \overline{q_f}$ such that $q_2' \stackrel{a_2}{\to} \ldots \stackrel{a_l}{\to} q_f'$ is allowed by $\mathcal{M}$. So $q_1' \stackrel{a_1}{\to} q_2' \stackrel{a_2}{\to} \ldots \stackrel{a_l}{\to} q_f'$. □

Monotonicity. We give here a negative result:

Proposition 3.4 *If $k \geq 1$, ρ_k is not monotonic.*

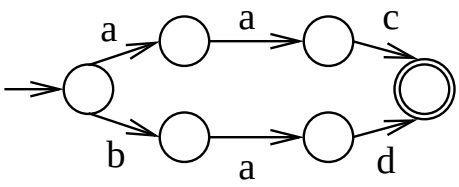


Figure 3.6: Automaton recognizing $L = \{aac + bad\}$

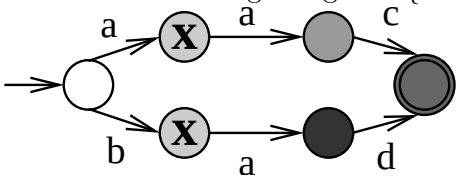


Figure 3.7: Equivalence classes of $\approx_1$

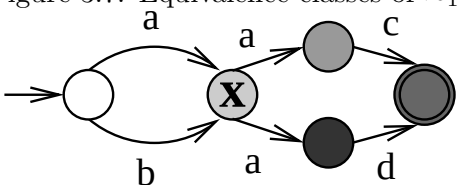


Figure 3.8: Automaton recognizing $\rho_1(L) = \{(a + b)a(c + d)\}$

This proposition is proved by the following counter-example. Let $L_1 = a.a.a^{k+1}$ and $L_2 = a.(a+b).a^{k+1}$ be two regular languages, with $L_1 \subset L_2$. We have $\rho_k(L_1) = a^*.a^{k+1}$ and $\rho_k(L_2) = a.(a + b).a^{k+1} = L_2$.

Remark 3.2 *ρ_0 is monotonic.*

This negative result does not matter in the abstract interpretation framework (some classical widening operators are also not monotonic, like the one of the lattice of convex polyhedra [CH78]. However, this is something we must keep in mind when re-using ρ_k in another framework.

3.2 Approximated Reachability Analysis of CFSMs

The analysis of CFSMs was the first motivation of this work on the lattice $(\mathrm{Reg}(\Sigma), \subseteq)$. In this section, we explain how we may obtain an approximated reachability analysis of CFSMs. The first step is the analysis of CFSMs with a single queue.

3.2.1 Analysis of CFSMs with a Single Queue

When a CFSM has a single queue, its state space is $\wp(C \times \Sigma^*)$, where C represents the set of control states and Σ the alphabet of messages. This state space is isomorphic to

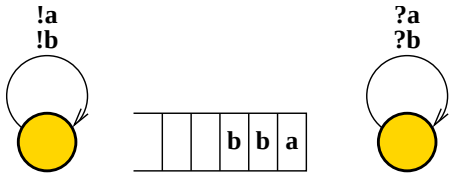


Figure 3.9: The infinite buffer model

$C \to \wp(\Sigma^*)$. The reachability set is the least fix-point of a function $F(L) = L_0 \cup \text{Post}(L)$, where $L_0 = \{\varepsilon\}$ is the initial content of the queue.

Example 3.3 *The CFSM depicted in Figure 3.9 models the "infinite buffer" protocol:* a sender process *can store a message in the queue, and* a receiver process *can read messages. In this example, there are only two kinds of messages:* a *and* b*.*

The CFSM resulting of the product of the two automata has a single control state. The reachability set of this CFSM is the least fix-point of the semantic function:

$$\begin{array}{rrcl} F: & \wp(\Sigma^*) & \to & \wp(\Sigma^*) \\ & L & \mapsto & \varepsilon \cup [\![!a]\!](L) \cup [\![?a]\!](L) \cup [\![!b]\!](L) \cup [\![?b]\!](L) \end{array}$$

We remind the definition of the semantic functions $[\![!a]\!]$ and $[\![?a]\!]$:

$$\begin{array}{rrcl} [\![!a]\!]: & \wp(\Sigma^*) & \to & \wp(\Sigma^*) \\ & L & \mapsto & L.a = \{w.a | w \in L\} \end{array}$$

$$\begin{array}{rrcl} [\![?a]\!]: & \wp(\Sigma^*) & \to & \wp(\Sigma^*) \\ & L & \mapsto & L/a = \{w | a.w \in L\} \end{array}$$

Example of reachability analysis. We detail step by step in Figure 3.10 the computation of the least fix-point of the function F defined in Example 3.3. This fix-point is found after three computation steps and one widening step, with a parameter $k = 0$.

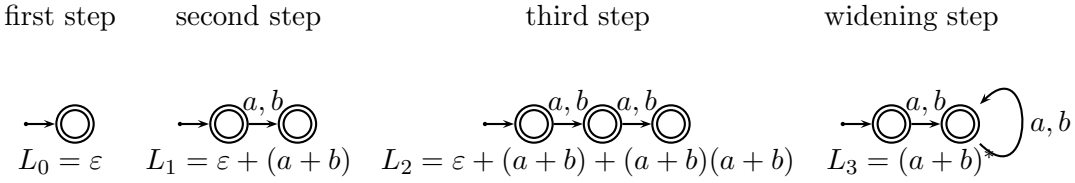


Figure 3.10: Computation of the least fix-point of F

After the third computation step, we apply the widening operator which merges the last two states of the automaton representing the set of queue-contents. As a result, the computation stops and the language L_3 of Figure 3.10 is the exact reachability set.

This example illustrates the abstractions of queue contents, as well as the effects of the widening operator. However, most CFSMs modeling real protocols have at least two FIFO channels.

3.2.2 Analysis of CFSMs with Several Queues

Representations. In the case of a single queue system, the content of the queue is represented by a finite word, and a set of contents by a language. When there are several queues, we must choose a way to represent the multiple queue-contents by a single representation.

We may choose to represent N words $w_1, \ldots, w_N$ by:

1. a vector of words $\langle w_1, \ldots, w_N \rangle$
2. a concatenation of words $w_1 \sharp \ldots \sharp w_N$, where $\sharp$ is a special letter separating the different queue contents.
3. an "interlaced" word formed by the sequence first letter of w_1, first letter of w_2 ..., first letter of w_N, second letter of w_1, second letter of w_2, etc. If the N words have different length, we use a special "blank" letter $-$. For example, we have three words $w_1 = aba$, $w_2 = c$ and $w_3 = deef$, we will represent them by the word $w = (acd)(b-e)(a-e)(--f)$

The three representations are equivalent in terms of expressiveness. The two first representations will be the basis of two kinds of analysis. The third one may be used for the representation of sets of integer vectors (e.g. within the NDD framework [WB95]), but we discard it because it introduces some "blank" letters each time a message is sent or received, it thus complicates the computation of the bisimulation relation.

Relational and non-relational lattices So far, we defined an abstract lattice for the representation a single queue-content. When there are N queues, we must choose whether we analyze each queue independently or we analyze all the queues together. In the first case, we may re-use N times the previous lattice. In the second case, we must find a proper abstract lattice. This is a general issue when using abstract interpretation techniques. We only need a *non-relational lattice* (i.e. a lattice without any relation between the contents of the different queues) in the first case, whereas a *relational lattice* (i.e. a lattice that keeps some relations between the contents of the different queues) is needed in the second case. For example, for the analysis of a program with several integer variables, one can use a non-relational lattice based on intervals [CC77a], or a relational one based on convex polyhedra [CH78].

The non-relational analysis is easier, but less precise. In our case, we use N "small" automata to represent the queue contents independently. For the relational analysis, we use one "big" automaton to represent all the queue contents. Thus the analysis will preserve some relationships between the different queue-contents. The reason why the non-relational analysis will be faster is that the operations we use are not linear (they

may be exponential) so it is faster to do N times those operations on small automata rather than one time on a big automaton.

3.2.3 A Non-Relational Lattice

Considering a system with N queues, we can represent the contents of all queues by a vector of finite words $\langle w_1, \ldots, w_N \rangle$. Generally speaking, a set of vectors $X \subseteq E^N$ can be approximated by a single vector $\langle X_1, \ldots, X_N \rangle \in \wp(E)^N$, where X_i is the projection of X on the dimension i. Moreover, a language $L \in \mathcal{L}(\Sigma)$ is approximated by a vector of languages $\alpha(L) = \langle L_1, \ldots, L_N \rangle$ where L_i is the projection of L on the alphabet of the queue i, Σ_i.

Example 3.4 *Considering a set of 3 vectors of words* $v_1 = \langle aa, cd \rangle$, $v_2 = \langle ab, cd \rangle$ *and* $v_2 = \langle aa, cc \rangle$*, we over-approximate it by the vector of languages* $\langle L_1, L_2 \rangle$ *with* $L_1 = \{aa + ab\}$ *and* $L_2 = \{cd + cc\}$*.*

Thus we first abstract the lattice $\mathcal{L}(\Sigma)$ by $\mathcal{L}(\Sigma_1) \times \cdots \times \mathcal{L}(\Sigma_N)$, loosing all relations between the contents of the different queues. Then we can reuse the regular languages lattice and we obtain the following connection:

$$\wp(\Sigma_1^* \times \ldots \times \Sigma_N^*) \underset{\alpha_c}{\overset{\gamma_c}{\leftrightarrows}} \mathcal{L}(\Sigma_1) \times \cdots \times \mathcal{L}(\Sigma_N) \xleftarrow{\gamma} \mathrm{Reg}(\Sigma_1) \times \cdots \times \mathrm{Reg}(\Sigma_N)$$

The operations in this lattice are the same as the operations on regular languages. For the widening operator, given a fixed integer k, we may extend the former definition to:

$$\langle L_1, \ldots, L_N \rangle \overrightarrow{\nabla} \langle L'_1, \ldots, L'_N \rangle \triangleq \langle L_1 \nabla L'_1, \ldots, L_N \nabla L'_N \rangle$$

If needed, we can also define $\overrightarrow{\nabla}$ with different values of k (one value for each single queue content).

Remark 3.3 *In this lattice, the upper bound ("the union") is no longer exact, because of the Cartesian product. For example, the upper bound of the languages* $\langle a, c \rangle$ *and* $\langle b, d \rangle$ *is the language* $\langle a + b, c + d \rangle$

3.2.4 A Relational Lattice

The QDD representation. We introduce a special letter $\sharp$, and represent the contents of all queues by a concatenation of words separated by $\sharp$.

Example 3.5 *Considering three queue contents represented by the words* $abaa \in \Sigma_1^*$*,* $cdec \in \Sigma_2^*$ *and* $fgg \in \Sigma_3^*$*, we represent all the contents by a single word* $abaa\sharp cdec\sharp fgg \in \Sigma^*$*.*

Since QDDs are finite automata, the abstract lattice $(\mathrm{Reg}(\Sigma), \sqsubseteq^\sharp)$ can be defined as before, except that the widening operator ∇ must avoid merging the different queue contents.

Effective representation and manipulation of QDDs. A QDD is a finite automaton with a special character $\sharp$. This character marks the separation between two queue-contents. Let $\mathcal{M} = (Q, \Sigma, Q_0, Q_f)$ be a QDD. A word $w = w_1 w_2 \dots w_N$ is effectively accepted by a QDD if $\exists q_0 \in Q_0, q_f \in Q_f,\ \delta(q_0, w') = q_f$ where $w' = w_1 \sharp w_2 \sharp \dots \sharp w_N$.

Thus each state $q \in Q$ of the QDD can be associated to one (and only one) queue-content. This association is given by a function $a : Q \to [1 \dots N]$. On the other hand, for an integer i, we can define $Q_i = \{q \in Q | a(q) = i\}$. The sub-automaton $\mathcal{M}_i = (Q_i, \Sigma_i, Q_{0,i}, Q_{f,i}, \delta_{/Q_i \times Q_i})$ recognizes the projection on the alphabet Σ_i of the language recognized by $\mathcal{M}$.

A state $q \in Q_i$ is *initial for a queue i* if a separation transition leads to this state (i.e. $\exists q' \in Q_{i-1}, q' \xrightarrow{\sharp} q$). The initial states for the first queue are the initial states of the automaton. A state $q \in Q_i$ is *final for a queue i* if a separation transition starts from this state (i.e. $\exists q' \in Q_{i-1}, q \xrightarrow{\sharp} q'$). The final states for the last queue are the final states of the automaton.

Concerning the standard operations ($\sqcup$, $\sqcap$, Send, Receive), and inclusion test $\subseteq$, they are natural extension of their counterpart for an automaton representing a single queue. In particular, they do not induce any approximation.

The widening operator for QDDs. We will use the same widening operator as before, but using $4n$ colors instead of 4. For a state q, we define its color $c(q)$ as:

- $c(q) = 4 * a(q)$ if q is both an initial and a final state for the queue $a(q)$,
- $c(q) = 4 * a(q) + 1$ if q is a final (but not initial) state for the queue $a(q)$,
- $c(q) = 4 * a(q) + 2$ if q is an initial (but not final) state for the queue $a(q)$,
- $c(q) = 4 * a(q) + 3$ otherwise.

A natural question arises: is such an operator invariant w.r.t. the ordering of queues ? The answer is unfortunately negative, as illustrated by the following example.

Example 3.6 *Let us consider a CFSM with 2 channels, $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{c, d\}$, and the language: $L = aaaac + baaad$. This language L_1 is recognized by a QDD (Figure 3.11). Choosing $k = 2$ for our widening operator, the two states marked with an "x" are merged, as well as the two states marked with a "y". Thus $\rho_2(L_1) = (a + b)aa(c + d)$. If we change the order, we start from the language $L' = caaaa + dbaaa$ recognized by a second QDD (Figure 3.12), which gives after widening $\rho_2(L') = L'$. In this case, the second ordering preserve the information, unlike the first one.*

Consequently, the precision of our analysis depends on the ordering. From a "philosophical" point of view, a widening operator which would be independent of the order would have been more satisfactory, but we did not find out yet such a widening operator, with good properties w.r.t. precision and efficiency. In a different setting however, where

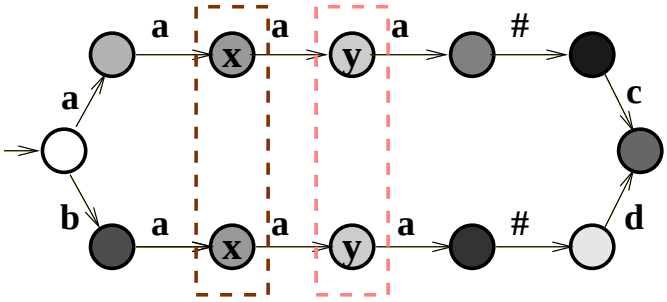


Figure 3.11: Equivalence classes for the first QDD

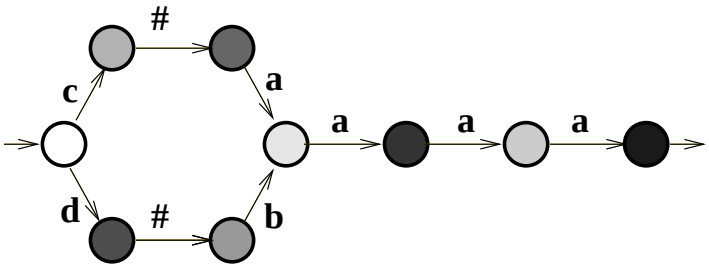


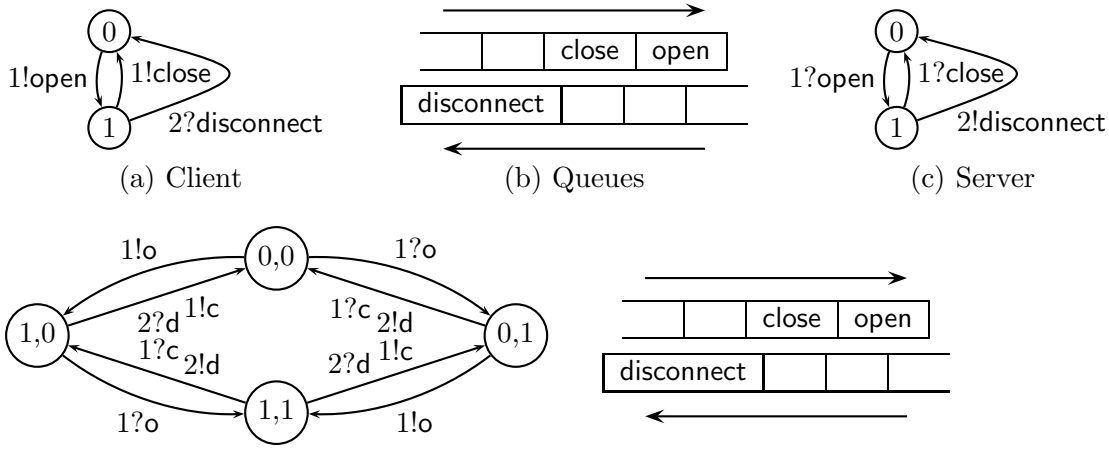
Figure 3.12: Equivalence classes for the second QDD

the aim is not to approximate but to compress the size of the representation, notice that the BDD representation for Boolean functions strongly depends on the order of variables.

If we relax efficiency, we may obtain such a widening operator by considering all the permutations on queues. We would apply the widening on each representation, and then take the intersection of all the results. We conjecture that the obtained operator would have the property of a widening operator. Notice however that the number of permutations is exponential, and that such a solution would imply the ability to convert from the representation with one order to a representation with a different order. If we consider again the similarity with BDDs, such a reordering would be certainly more complex in our case than for BDDs, as we have cyclic graphs while BDDs are acyclic graphs.

3.3 Applications

In this section, we detail some examples of reachability analyses of FIFO channel systems, and compare the results with acceleration techniques. Most examples come from the modeling of simple communication protocols, or are toy examples designed to illustrate the properties of our approach.



(a) Client (b) Queues (c) Server

(d) Global CFSM: product of client and server processes

Figure 3.13: The connection/disconnection protocol

3.3.1 Examples of Protocols and their Analyses

The connection/disconnection protocol. We remind the definition this protocol, which was first given in Chapter 1.

Example 3.7 *The connection/disconnection protocol [JR87] between two machines is the following (Figure 3.13): the client can open a session by sending the message* open *to the server. Once a session is opened, the client may close it on its own by sending the message* close *or on the demand of the server if it receives the message* disconnect*. The server can read the request messages* open *and* close*, and ask for a session closure.*

This protocol is a good example of what happens when the contents of the different channels strongly depend on each other; in this case, the non-relational analysis is worse (both for precision and efficiency) than the relational one. We choose $k = 2$ to define the ∇ operator in both cases.
Here is the result of the relational analysis (7 iteration steps):

$0,0$	$(o.c)^*\#\varepsilon$
	$+\quad (o.c)^*\#\varepsilon$
	$+\quad (c.o)^*.o.c.(o.c)^*\#\varepsilon$
	$+\quad c.(o.c)^*\#d$
$1,0$	$\varepsilon\#d$
	$+\quad o\#\varepsilon$
	$+\quad (o.c)^*.o\#\varepsilon$
	$+\quad (c.o)^*\#d$
	$+\quad (c.o)^*.o.(c.o)^*\#\varepsilon$
$0,1$	$c.(o.c)^*\#\varepsilon$
$1,1$	$\varepsilon\#\varepsilon$
	$+\quad (c.o)^*\#\varepsilon$

Here is the result of the non-relational analysis (8 iteration steps):

$0,0$	$o^* + o^*.c.(o^+.c)^*.o^*$	d^*
$1,0$	$o^* + o^*.c.(o^+.c)^*.o^+$	d^*
$0,1$	$o^* + o^*.c.(o^+.c)^*.o^*$	d^*
$1,1$	$o^+ + o^*.c.(o^+.c)^*.o^+$	d^*

The first analysis (but not the second one) shows that there is at most one d in the second queue, and that there is the same number of o and c (+/- 1) in the first queue[1].

The reason why the non-relational analysis is far worse in this case is that it cannot detect that there is at most one d in the second queue, thus many non-reachable states are explored by this analysis.

A toy example. The previous example shows that the non-relational analysis may be longer than the relational one. However, this is an exception. In general, this kind of analysis is faster. We provide an example where the non-relational analysis is much faster than the relational one. This toy example (Figure 3.14) is a CFSM with 4 queues, where we can only add messages. We can prove that the content of the queues is $a^{n_a}\sharp b^{n_b}\sharp c^{n_c}\sharp d^{n_d}$, with $n_a = n_b$ and $n_c = n_d$. If we choose $k = 5$ (for the widening definition), the non-relational analysis stops in about one second and leads to the result (a^*, b^*, c^*, d^*).

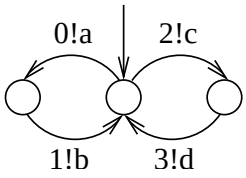


Figure 3.14: A toy example

The relational one takes several minutes for a similar result. In fact, the relational analysis will consider all the languages $L_{i,j} = \{a^i\#b^i\#c^j\#d^j\}$ with $i, j = 1..5$ before performing any widening operation.

The Alternating Bit Protocol. The Alternating Bit Protocol (ABP) is a data-transmission protocol, between a sender S and a receiver R. S transmits some data package m through a FIFO channel C[2], and R and S exchange some information (one-bit messages) through two channels K and L (Figure 3.15).

[1]Indeed, this protocol is a faulty one, there is a correct version available in [JR87]. This correct version can be easily analyzed, since the length of the queue-content is bounded.

[2]This transmission is often modeled by two abstract actions SEND and RECEIVE instead of this third channel C.

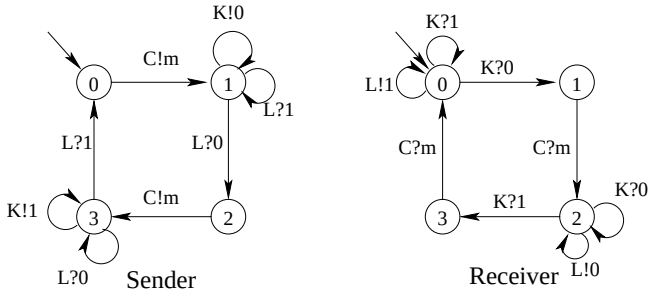


Figure 3.15: The Alternating Bit Protocol

We performed a relational analysis of the CFSM modeling this protocol:

Sender	Receiver	Contents K#L#C
0	0	$1^*\#1^*\#\varepsilon$
0	1	$\emptyset$
0	2	$\emptyset$
0	3	$\emptyset$
1	0	$1^*0^*\#1^*\#m$
1	1	$0^*\#1^*\#m$
1	2	$0^*\#1^*0^*\#\varepsilon$
1	3	$\emptyset$
2	0	$\emptyset$
2	1	$\emptyset$
2	2	$0^*\#0^*\#\varepsilon$
2	3	$\emptyset$
3	0	$1^*\#0^*1^*\#\varepsilon$
3	1	$\emptyset$
3	2	$0^*1^*\#0^*\#m$
3	3	$1^*\#0^*\#m$

This result was obtained in 8 iteration steps,using ρ_0 to define the widening operator. It shows that the control states $(0,1)$, $(0,2)$, $(0,3)$, $(1,3)$, $(2,0)$, $(2,1)$, $(2,3)$ and $(3,1)$ are not reachable and that there is at most one message in data channel C. Comparing this result with the experimental results given in [BG97, AAB99] shows that we obtain the exact result.

A non-regular example. Our abstraction can deal with cases where the reachability set is not regular. Let us consider the CFSM depicted in Figure 3.16. Each process can send a message a or c, and a synchronization is guaranteed by the messages b and d.

In location $(0/0)$, the content of the queues will be $a^n\sharp\varepsilon\sharp c^n\sharp\varepsilon$ with $n \geq 0$. This set is non-regular, and thus cannot be represented by a regular expression. Our method will find an over-approximation of the exact reachability set. In location $(0/0)$ the

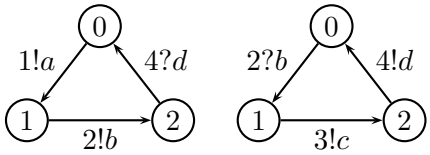


Figure 3.16: A non-regular protocol

queue-content we found, with ∇_1, is represented by the language:

$$L_{(0/0)} = \varepsilon\sharp\varepsilon\sharp\varepsilon\sharp\varepsilon + a\sharp\varepsilon\sharp c\sharp\varepsilon + aaa^*\sharp\varepsilon\sharp ccc^*\sharp\varepsilon$$

This example shows that our method may give a good over-approximation of a non-regular reachability set.

A protocol with nested loops. Figure 3.17 depicts a protocol, which is an abstraction of systems exchanging frames composed of several packets.

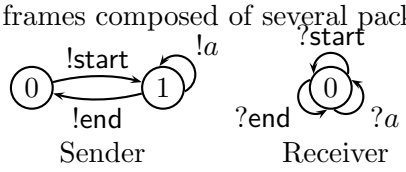


Figure 3.17: Nested loop

The sender first sends a start message, then sends any number of a messages and ends the frame with an end message. The receiver can read any message at any time.

Our analysis, with ∇_1, shows that, when the sender is in location 0, the content of the queue is:

$$L_0 = \varepsilon + (s + \varepsilon)a^*e(sa^*e)^*$$

Here the ability of representing regular expressions with nested Kleene's closures is important; in this case we even obtain the exact reachability set.

Lossy channels. Even if we built our analysis for CFSMs with perfect channels, it can also deal with lossy channels.

First, we can define an operator on automata simulating the possible lost of messages. This operator is simply adding an epsilon transition between states linked with a normal transition. Since we often need to determinize the automata, this introduction of so many epsilon transitions is really a burden.

So we simulate the loose of messages by altering the semantics of the send operation: $[\![!a]\!](L) = L \cup L.a$. With only this alteration, we have a system equivalent to a lossy channel system, *i.e.* any state reachable with the *lossy* semantics is also reachable with this semantics *perfect'*, which is identical to the semantics of perfect FIFO channel systems, excepted that a message can be lost when it is inserted in a queue.

Proposition 3.5 *Let S be a set of configurations where all lossy channels are empty. Then* $\mathrm{Post}^*_{lossy}(S) = \mathrm{Post}^*_{perfect'}(S)$

Proof: $\text{Post}^*_{perfect'}(S) \subseteq \text{Post}^*_{lossy}(S)$ because the loss of a message is possible only when the message is inserted, whereas the lossy semantics allows this loss to happen at any time.

Then we consider a configuration $(c, w_1, \ldots, w_N) \in \text{Post}^*_{lossy}(C)$. We have a sequence of operations $(c, \varepsilon, \ldots, \varepsilon) \to \cdots \to (c, w_1, \ldots, w_N)$. Each steps is either an input or an output, and the loss of some messages. Since the queues are empty at the beginning, we take exactly the same operation, but without inserting the messages that are later lost in the queues. This lead to the same final configuration $(c, w_1, \ldots, w_N)$. □

This semantics is not exactly equivalent to a lossy channel system, because we must assume the queues are empty at the beginning. However, this assumption holds when we perform a standard reachability analysis and it is far less costly than the first method. Moreover, we can define CFSMs with both perfect channels and lossy ones. This is the reason why we implemented this altered semantics.

3.3.2 Comparison with Acceleration Techniques

Compared to acceleration techniques, the main advantage of the method based on abstract interpretation is that our analysis will always terminate. But the guarantee of termination is useless if the result (an approximation of the exact result) is not precise enough.

The abstract values are regular languages; so our representation of queue contents is as expressive as all the representations based on regular languages (the QDDs) or regular expressions (SLREs). The only representation that is incomparable is the CQDDs. The expressiveness of the representation, however, is only one of the aspects of the precision of the analysis. We must also take into account the approximation we make during the computation, or, in the case of acceleration techniques, the kind of CFSM we are able to analyze.

In Tab. 3.1 we compare the techniques mentioned in the previous chapter, with our non-relational and relational analysis, on some examples of communication protocols: the ABP and the connection/disconnection protocol, and two toy examples: the non-regular one and the one with nested loop. We did not consider the method of [ABJ98], which assumes lossy channels.

- *yes* means that the reachability analysis gives the exact result.
- *no* means that the reachability analysis does not terminate.
- *approx* means that the reachability analysis gives an over-approximation of the reachability set.

The only case where our relational method gives less satisfactory results than another method, which is also the only case where the result is not exact, is the Non-Regular protocol. On this protocol, the CQDD method can compute the exact reachability set $\bigcup_{n\geq 0} a^n\sharp\epsilon\sharp c^n\sharp\epsilon$, whereas we approximate it, using ∇_k, by $\bigcup_{0\leq n\leq k+2} a^n\sharp\epsilon\sharp c^n\sharp\epsilon \cup$

Example	Acceleration techniques			Regular languages with widening	
	SLRE [FIS03]	QDD [BG97]	CQDD [BH99]	non-relational	relational
(1) ABP	yes	yes	yes	yes	yes
(2) Conn./disconn.	yes	yes	yes	approx[a]	yes
(3) Non-regular	no[a,b]	no[a,b]	yes	approx[b]	approx[b]
(4) Nested loops	no[b]	yes	no[b]	yes	yes

(a) counting loops [BG97] that cannot be accelerated
(b) exact set not representable

Table 3.1: Comparison of acceleration techniques with abstract interpretation-based analyzes

$a^{k+2}a^*\sharp\epsilon\sharp c^{k+2}c^*\sharp\epsilon$, which is not so bad. On the other hand, none of the other methods delivers results for all the examples.

3.3.3 Beyond Reachability Analysis

Observers. As we explained it in Section 1.2, we can verify safety properties when we are able to compute the reachability set, or an over-approximation of the reachability set. We first take the observer $\mathcal{O}_{\neg p}$, and do the synchronous product $\mathcal{O}_p \times \mathcal{M}$. We then compute the reachability set to know if $\mathcal{M}$ violates p. Unlike what is presented in Section 1.2, the observer is a CFSM instead of a transition system. We give here the definition of the synchronous product of two CFSMs.

Definition 3.4 (Synchronous product of two CFSMs) *Let $\mathcal{M}^1 = \langle C^1, \Sigma, c_0^1, \Delta^1\rangle$ and $\mathcal{M}^2 = \langle C^2, \Sigma, c_0^2, \Delta^2\rangle$ be two CFSMs. The synchronous product $\mathcal{M}^1 \times \mathcal{M}^2$ is defined as $\mathcal{M} = \langle C, \Sigma, c_0, \Delta\rangle$:*

- $C = C^1 \times C^2$,
- $c_0 = (c_0^1, c_0^2)$,
- Δ *is defined by the rules:*

$$\frac{(q^1, \text{act}, q'^1) \in \Delta^1 \quad (q^2, \text{act}, q'^2) \in \Delta^2}{((q^1, q^2), \text{act}, (q'^1, q'^2)) \in \Delta}$$

where act *is an input $i?m$ or an output $i!m$.*

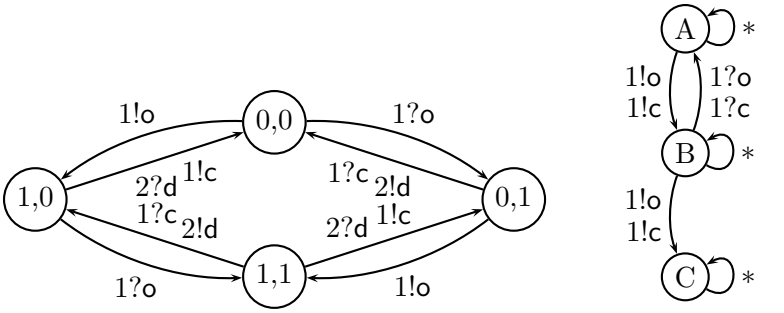
Remark 3.4 *If $[\![\mathcal{M}^1]\!]$ (resp. $[\![\mathcal{M}^2]\!]$) is the transition system that gives the semantics of $\mathcal{M}^1$ (resp. $\mathcal{M}^2$), then $[\![\mathcal{M}^1 \times \mathcal{M}^2]\!] = [\![\mathcal{M}^1]\!] \times [\![\mathcal{M}^1]\!]$.*

In general, the behavior of the synchronous product is a restriction of the behavior of each CFSM. Since the observer is not allowed to alter the behavior of a system, it must be complete, *i.e.* it must accept any sequence of transitions of the observed CFSM.

Verdicts. In our case, we only have an over-approximation of the reachability set. On one hand, this is a drawback, because we cannot be sure the property is violated. On the other hand, if the result of our analysis shows that no bad state is reachable, we are sure that the protocol does not violate the property p. In the first case, we may refine the abstraction in order to be more precise, and perform a new analysis.

Example 3.8 *Let us take back the example of the connection/disconnection protocol. The observer in Figure 3.18 checks the property "there is never more than one message in the first queue"; it is basically a counter with a sink state. The transitions labeled by* $*$ *are any transition not explicitly mentioned on the observer.*

The reachability analysis show that this property might be violated. Since we know that our analysis corresponds to the result of the acceleration techniques on this example, it is thus exact and we are sure the property is violated. This example illustrates the interest of combining acceleration techniques with our method, in order to prove that a property is violated.



(a) CFSM $\mathcal{S}$ (b) Observer $\mathcal{O}$

Figure 3.18: CFSM and its observer

3.4 Conclusion

In this chapter, we detailed the abstract lattice of regular languages, and showed how to perform reachability analysis of CFSMs using this lattice. We analyzed some classical examples of communication protocols, and compared our method to the acceleration techniques.

The lattice of regular languages is indeed well-known, except the widening operator. As we said, a similar operator was mentioned in [Fer01]. We adapted it to the analysis of FIFO channel systems, and studied its properties. This widening is also parametric, depending on an initial partition and on an integer parameter k. It might be employed for the analysis of systems using stacks; with a representation of a stack by a finite word, pushing or popping elements are simple automata operations. In this case, the suffix conservation property guarantees that the widening does not induce some approximation on the last pushed elements. We will discuss this possibility in Chapter 5.

Compared to other automatic analyzes of CFSMs, our method has significant advantages. First, it can be applied to any CFSM and always terminates. It is technically simple, based on standard abstract interpretation techniques and well-known concepts like regular languages and bisimulation of depth k. Despite its simplicity, that we consider as a strength, our method is often as accurate as acceleration techniques on standard examples, and it can deal with counting loops [BG97]. It is however unable to certify by itself whether the obtained result is exact or not (which is a limitation common to abstract interpretation techniques). Last but not least, this approach is more amenable to the combination of FIFO channels with other unbounded datatypes, like counters, in the spirit of [JHR99] (see the following chapters). Indeed, it seems very difficult to accelerate loops where FIFO operations are guarded by numerical tests on counters and where counters are conversely updated depending on the FIFO channel contents.

For CFSMs, our method is a good alternative to acceleration based techniques. The two approaches may actually be seen as complementary. Typically, one can first try to get the exact reachability set using acceleration techniques and then apply our method in case of failure. A more interesting combination consists in using acceleration techniques to add meta-transitions in the original model, when possible, and to apply our method to the augmented system. Since we have a QDD representation, we can reuse the QDDs acceleration technique in this context.

The examples of protocols are quite simple, but complex protocols often have integer and boolean variables, and parametrized messages, which are not easily handled by the CFSM model. This is the reason why we extend this work to more complex models, in Chapter 5. This extension needs a representation of regular languages over an infinite alphabet: the lattice automata.

Chapter 4

Lattice Automata

This chapter introduces the notion of *lattice automata*, an abstract representation for languages over infinite alphabets. The notion of lattice automata emerged when we wanted to generalize the verification of communicating protocols to the ones where messages carry integer values. If one wants to perform an approximated reachability analysis like the one presented in the last chapter, there are two possibilities:

1. to choose a finite abstraction of the values of the parameters, and perform the same analysis as before,

2. to define a similar abstract lattice, but over an infinite alphabet instead of a finite one.

We preferred the second approach, which is more generic, and defined an automata representation on languages over infinite alphabets. The main idea is quite simple: instead of labeling each transitions by an element of a finite alphabet Σ, each transition is labeled by an element of an atomic lattice, representing a possibly infinite set of atoms. This definition of lattice automata enables finite automaton-like manipulations of regular languages over an infinite alphabet.

Some other researchers introduced automata working on an infinite alphabet, like register automata [KF94], pebble automata [MSV00] or data automata [BMS$^+$06]. They are associated to some decidable logic, with the idea that a word with data satisfies the logical formula if it is recognized by the corresponding automata. Since our aim is to extend the analysis of Chapter 3, those automata are not adapted to our needs.

Outline. We first formally define the lattice automata, and we advocate the idea of using a partition of the lattice in order to define a projected finite automaton which acts as a guide for defining determinization and minimization operations. We motivate our choices and show that they lead to a robust notion of approximation, in the sense that normalization is an upper-closure operation and can be seen as a best upper-approximation in the lattice of normalized lattice automata. We then define the other

classical operations such as union and intersection, before defining a suitable widening operator which allows to consider lattice automata as a fully equipped abstract domain.

4.1 Definition and Discussion

Lattice automata are finite automata, the transitions of which are labeled by elements of an atomic lattice $(\Lambda, \sqsubseteq)$ instead of elements of a finite and unstructured alphabet. We remind that an atomic lattice $(\Lambda, \sqsubseteq)$ is a lattice with a set of *atoms* $At(\Lambda) \subseteq \Lambda$ such that:

- each atom a is greater only to $\bot$: $\lambda \sqsubseteq a \implies \lambda = a \vee \lambda = \bot$,
- each element λ, except $\bot$, is the upper bound of the atoms lesser than itself: $\lambda = \bigsqcup\{a \in At(\Lambda) | a \sqsubseteq \lambda\}$.

Most abstract domains used in the abstract interpretation framework are atomic: for example, the lattice of intervals is atomic : its atoms are the singletons $[x, x]$ with $x \in \mathbb{Q}$ and we have $[a, b] = \sqcup\{[x, x] \mid a \leq x \leq b\}$.

Example 4.1 *A simple non-atomic lattice is the one defined by the set* $\{\bot, \top, a, b\}$ *and the partial order:* $\bot \sqsubseteq a \sqsubseteq b \sqsubseteq \top$ *Figure 4.1. The only atom of this lattice is* a*, and* $\sqcup\{\lambda \in At(\Lambda) \mid \lambda \sqsubseteq b\} = a \neq b$.

Figure 4.1: Example of a non-atomic lattice

Lattice automata recognize languages on atomic elements of this lattice. For instance, the interval automaton of Fig. 4.2 recognizes all sequences of rational numbers $x_0 \dots x_{n-1}$ where n is odd, $x_{2i} \in [0, 1]$ and $x_{2i+1} \in [1, 2]$. Such an automaton can be used to represent possible contents of a queue containing rational numbers.

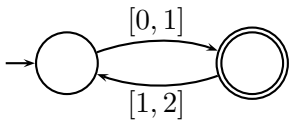


Figure 4.2: An interval automaton

However, the classical operations on finite automata cannot be extended as is on lattice automata. It is the case of the determinization and minimization operations. Consequently, we propose another notion of determinization which is in some sense an optimal approximation of the classical operation.

4.1.1 Basic Definition

Definition 4.1 (Lattice automaton) *A lattice automaton is a tuple* $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ *where:*

- Λ *is an atomic lattice, the order of which is denoted by* $\sqsubseteq$*;*
- Q *is a finite set of states;*
- $Q_0 \subseteq Q$ *and* $Q_f \subseteq Q$ *are the sets of initial and final states;*
- $\delta \subseteq Q \times (\Lambda \setminus \{\bot\}) \times Q$ *is a finite transition relation.*[1]

A finite word $w = a_0 \dots a_n \in At(\Lambda)^*$ *is accepted by the lattice automaton if there exists a sequence* $q_0, q_1, \dots, q_{n+1}$ *such that* $q_0 \in Q_0$, $q_{n+1} \in Q_f$, *and* $\forall i \leq n,\ \exists (q_i, \lambda_i, q_{i+1}) \in \delta :\ a_i \sqsubseteq \lambda_i$.

The set of words recognized by a lattice automaton $\mathcal{A}$ is denoted by $L_{\mathcal{A}}$. The inclusion relation between languages induces a partial order on lattice automata:

Definition 4.2 (Partial order $\sqsubseteq$ on lattice automata) *The partial order* $\sqsubseteq$ *between lattice automata is defined as* $\mathcal{A} \sqsubseteq \mathcal{A}'$ *iff* $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$.

Remark 4.1 (Downward closure of transitions) *With the Definition 4.1, if there exists a transition* (q, λ, q') *in* δ, *then any* (q, λ', q') *with* $\lambda' \sqsubseteq \lambda$ *may be added without modifying the recognized language. Hence, such transitions are redundant.*

From now on, we assume that all transitions of a lattice automaton are maximal in the following sense: $(q, \lambda, q') \in \delta \wedge (q, \lambda', q') \in \delta$ implies that λ and λ' are not comparable.

Remark 4.2 (Words on atoms) *We restrict the recognized words of a lattice automaton to be composed of* atoms *because we want the two automata of Fig. 4.6 to be equivalent in terms of their recognized languages. If words were composed of any elements of the lattice, the word composed of the single element* $\{0 \leq x \leq 3, 0 \leq y \leq 1\}$ *would be recognized by the first automaton but not the second one. In the same spirit, we require that the lattice* Λ *is atomic, so that any element* $\lambda \in \Lambda$ *labeling a transition is isomorphic to the set of atoms it dominates.*

Remark 4.3 (Status of $\bot$) $\bot$ *cannot label a transition, not only as a consequence of the previous remark, but also because this corresponds to the intuition that* $\bot$ *does not represent any element and denotes a notion of emptiness.*

Definition 4.3 *A lattice-based regular language is a language recognized by a lattice automaton* $\mathcal{A}$. *We denote by* $\mathrm{Reg}(\Lambda)$ *the set of lattice-based regular languages.*

[1] No transition is labeled with the bottom element $\bot$.

4.1.2 Discussion

Definition 4.1 raises however a number of problems. We expose them before introducing our solution to them. The first problem is related to the bounded branching degree property: in a deterministic finite automaton, there are at most $|\Sigma|$ transitions outgoing from a state. However, with definition 4.1, the branching degree of lattice automata is not bounded because the number of atoms is possibly infinite, as shown in Fig. 4.3.

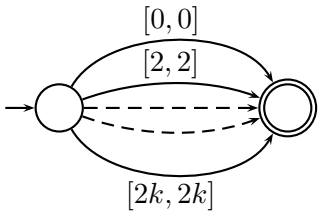
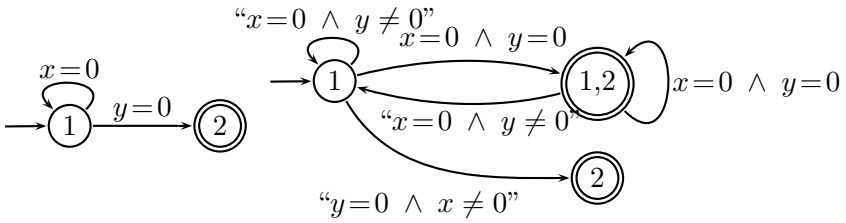


Figure 4.3: A family of interval automata $\mathcal{A}_k$ with unbounded branching degree

The second problem is related to the notion of determinism. The classical notion of determinism can be extended in a straightforward way as follows:

Definition 4.4 (Deterministic lattice automaton) *A lattice automaton* $\langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ *is* deterministic *if it has a unique initial state and if* $(q, \lambda_1, q_1) \in \delta \wedge (q, \lambda_2, q_2) \in \delta \implies \lambda_1 \sqcap \lambda_2 = \bot$.

Can now any lattice automaton be made deterministic while still recognizing the same language ? The answer is negative, as illustrated by the example of Figure 4.4, which uses the lattice of affine equalities [Kar76].



(a) non-deterministic automaton (b) deterministic automaton ?

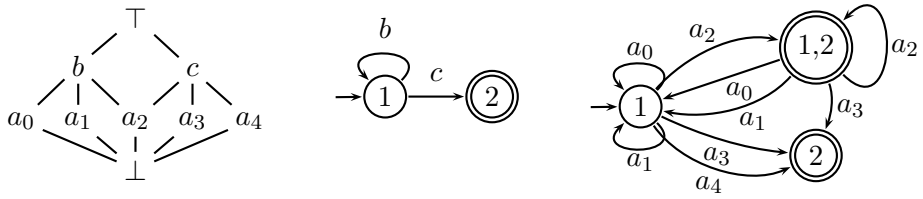
Figure 4.4: Attempt to determinize a lattice automaton on the lattice of affine equalities

Example 4.2 *The lattice of affine equalities, which could also be called the lattice of affine subspaces, is the lattice formed by the conjunctions of affine equalities on the space* $\mathbb{R}^n$ *; Given a finite set of variables* $\{x_1, \ldots, x_n\}$*, an element of the lattice of affine equalities is defined by a conjunction of linear equalities* $a_1x_1 + \cdots + a_nx_n = 0$*, with* $a_1, \ldots, a_n \in \mathbb{R}$*. The lattice of affine equalities is also called the lattice of affine subspaces. It is an atomic lattice, ordered by the classical* $\subseteq$*, and the atoms are the elements of* $\mathbb{R}^n$*.*

The automaton depicted on Figure 4.4 recognizes the words $p_1 \dots p_m$ where $p_1 \dots p_{m-1}$ belongs to the line define by $x = 0$ and p_m to the one defined by $y = 0$. This automaton is non deterministic because in state 1, the atom $\{(0,0)\}$ can trigger both transitions.

An attempt to determinize this automaton would be to split the transitions, and take into account the case $x = 0 \wedge y = 0$. However, the other transitions would be defined by linear inequalities instead of linear equalities. Thus, there is no hope to find a deterministic lattice automaton accepting exactly the same language.

The problem here is that elements of an atomic lattice cannot be complemented in general. With some lattices, like the lattice of intervals where an element can be complemented by a finite union of intervals, this problem can be overcome by splitting transitions, *cf.* Fig. 4.5, but we are back then to the branching degree problem.



(a) Lattice (b) non-deterministic automaton (c) acceptable deterministic automaton

Figure 4.5: Attempt to determinize a lattice automaton

Example 4.3 *Figure 4.5 illustrates another attempt to determinize a lattice automaton. The lattice is depicted on Figure 4.5(a) and the automaton on Figure 4.5(b). This automaton is non-deterministic because in location 1, the atom a_2 can trigger both transitions. We split the two transitions and obtain the automaton depicted on Fig. 4.5(c). Each transition is thus an explicit union of atoms, not the least upper bound.*

Moreover, using such explicit unions may be problematic to define minimization and canonical form. For instance, how to choose between the two automata of Fig. 4.6 ? The problem here is that there is no canonical representation for unions of convex polyhedra.

Example 4.4 *Figure 4.6 depicts two automata recognizing the same set of one-letter words $L = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(3,0),(3,1)\}$. The transitions of the two automata are however not comparable.*

4.1.3 Partitioned Lattice Automata

The solution we propose to fix these problems is to use a finite partition of the lattice Λ, which allows to decide when two transitions should be merged using the least upper

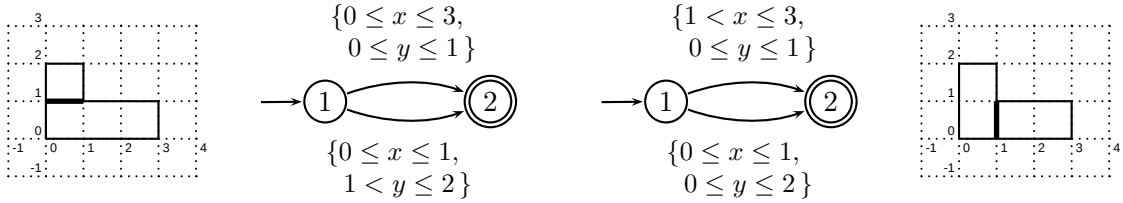


Figure 4.6: Two deterministic convex polyhedra automata that are equivalent

bound operator. The fusion of transitions will induce in general an over-approximation, controlled by the fineness of the partition. The gain is that the projection of labels onto their equivalence classes produces a finite automaton on which we can reuse classical notions.

Definition 4.5 (Partitioned lattice automaton (PLA)) *A* partitioned lattice automaton *(PLA) $\mathcal{A}$ is a lattice automaton $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ equipped with a partitioning function $\pi : \Sigma \to \Lambda$ such that Σ is a finite alphabet and the transition relation δ satisfies:*

$$\forall (q, \lambda, q') \in \delta,\ \exists \sigma \in \Sigma : \lambda \sqsubseteq \pi(\sigma)$$

Precision on the partitionning function. We consider n elements $\lambda_1, \ldots, \lambda_n$ of the atomic lattice Λ verifying:

1. if $i \neq j$, then $\lambda_i \sqcap \lambda_j = bot$;
2. for any atom $a \in At(\Lambda)$, there is one element $\lambda_i \sqsupseteq a$. The first property ensure that this element λ_i is unique.

So this set $\lambda_1, \ldots, \lambda_n$ defines a partition $P_1, \ldots, P_n$ of $At(\Lambda)$, with $P_i = \{a \in At(\Lambda) \mid a \sqsubseteq \lambda_i\}$. A function $\pi : \Sigma \to \Lambda$ is a partitionning function if the elements $\pi(\sigma_1), \ldots, \pi(\sigma_n)$ define such a partition. The definition of the PLA expresses that each label of a transition belongs to a single equivalence class of the partition.

A PLA is *merged*[2] if there is at most one transition per equivalence class betwwen two states, in other words if:

$$\forall q, q', \lambda_1, \lambda_2,\ \ (q, \lambda_1, q') \in \delta \wedge (q, \lambda_2, q') \in \delta \implies \pi^{-1}(\lambda_1) \cap \pi^{-1}(\lambda_2) = \emptyset$$

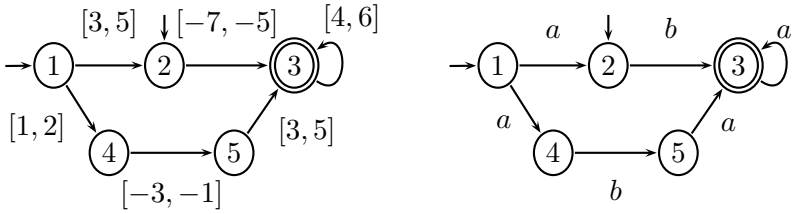
Definition 4.6 (Shape automaton and Shape equivalence) *Given a PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$, its shape automaton* shape($\mathcal{A}$) *is a finite automaton $(\Sigma, Q, Q_0, Q_f, \rightarrow)$ obtained by projecting the transition relation δ onto the equivalence classes:*

$$\frac{(q, \lambda, q') \in \delta}{(q, \pi^{-1}(\lambda), q') \in \rightarrow}$$

[2]The term "merged" comes from the merging operation needed to build a merged PLA from a given PLA.

Two PLAs are shape-equivalent *if their two shape automata recognize the same language*[3].

Example 4.5 *Figure 4.7 depicts an interval automaton $\mathcal{A}$, and its shape automaton* shape($\mathcal{A}$). *The partition is given by the function* $\pi : a \mapsto]-\infty, 0[\ ,\ b \mapsto [0, +\infty[$.



Lattice automaton $\mathcal{A}$ Finite automaton shape($\mathcal{A}$)

$$\text{Partition}: \quad \begin{matrix} \{\ ]-\infty, 0[& , & [0, +\infty[\ \} \\ \{b & , & a\} \end{matrix}$$

Figure 4.7: An interval automaton and its shape

Two transitions of a PLA labeled by elements belonging to different equivalence classes cannot be merged and are always kept separate, whereas they might be merged in the opposite case. Deterministic merged PLAs have the finite branching degree property: their states can have at most $|\Sigma|$ outgoing transitions.

From an expressiveness point of view, PLA are as expressive as lattice automata.

Proposition 4.1 (Equivalence between lattice automata and PLA) *Given a lattice automaton $\mathcal{A} = \langle \Lambda, Q, Q_0, Q_f, \delta \rangle$ and a partitioning function $\pi : \Sigma \to \Lambda$, there exists a partitioned lattice automaton $\mathcal{A}' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ recognizing the same language (this relies on the atomic lattice assumption).*

Proof: $\mathcal{A}'$ is obtained from $\mathcal{A}$ by replacing each transition (q, λ, q') of $\mathcal{A}$ by at most $|\Sigma|$ transitions (q, λ_i, q'), where $\lambda_i = \lambda \sqcap \pi(\sigma_i)$ if $\lambda_i \neq \bot$. □

However, for a given partition, *merged* PLAs are strictly less expressive, as shown on Fig. 4.6 and 4.8 with the trivial partition of size 1. Moreover, if one considers the lattice of affine equalities, which can be partitioned only with the trivial partition of size 1, the automaton of Fig. 4.4(a) shows that in general merged PLA are strictly less expressive than PLA.

Proposition 4.2 *Let $\mathcal{A}$ be a PLA. If* shape($\mathcal{A}$) *is deterministic, then $\mathcal{A}$ is deterministic.*

[3]In the following, shape automata are simply called "shapes".

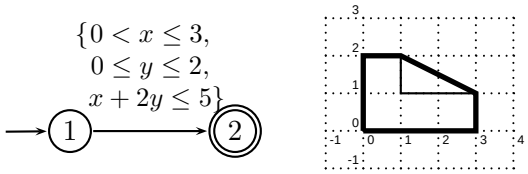


Figure 4.8: A merged PLA with the one-element partition

Proof: Let (q, λ_1, q_1) and (q, λ_2, q_2) be two transitions of $\mathcal{A}$, and (q, σ_1, q_1) and (q, σ_2, q_2) be the two corresponding transitions of shape($\mathcal{A}$). Since shape($\mathcal{A}$) is deterministic, $\pi^{-1}(\lambda_1) \cap \pi^{-1}(\lambda_2) = \emptyset$. Since π is a partitioning function, we have :

1. $\lambda_i \sqsubseteq \pi(\sigma_i)$
2. $\pi(\sigma_1) \sqcap \pi(\sigma_2) = \bot$

So $\lambda_1 \sqcap \lambda_2 = \bot$. And then $\mathcal{A}$ is deterministic. □

The converse is false in general. This property leads us to define a stronger notion of determinism :

Definition 4.7 (Strong determinism) *A PLA $\mathcal{A}$ is strongly deterministic if* shape($\mathcal{A}$) *is deterministic.*

This notion of determinism is useful since it is defined on the well-known finite automata, and is a guideline for the definition of a determinization algorithm. Therefore, in the sequel, "deterministic" means "strongly deterministic".

With all those definitions, we will be able to define a normalized form for lattice-based regular language L. This normalization will exploit the following lemma :

Lemma 4.1 (Testing language inclusion) *Let $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ and $\mathcal{A}' = \langle \Lambda, \pi, Q', Q'_0, Q'_f, \delta' \rangle$ be two merged PLAs with the same partitioning function*[4]*. Then $\mathcal{A} \sqsubseteq \mathcal{A}'$ iff there is a simulation relation $\mathcal{R} \subseteq \wp(Q) \times \wp(Q')$ verifying :*

1. $Q_0 \mathcal{R} Q'_0$
2. $\forall X \subseteq Q, \forall Y \subseteq Q' : \; X \mathcal{R} Y \implies \Big[(X \cap Q_f \neq \emptyset) \implies \Big(Y \cap Q'_f \neq \emptyset \Big) \Big]$
3. *Let $X \subseteq Q$ and $Y \subseteq Q'$ such that $X \mathcal{R} Y$. For any atom $a \in At(\Lambda)$, there are two sets $X_a \subseteq Q$ and $Y_a \subseteq Q'$ such that $X_a \mathcal{R} Y_a$ and:*

$$(\exists (q_x, \lambda_x, q'_x) \in \delta : q_x \in X \wedge a \sqsubseteq \lambda_x)$$
$$\Downarrow$$
$$\big(\exists (q_y, \lambda_y, q'_y) \in \delta' : q_y \in Y \wedge a \sqsubseteq \lambda_y \wedge (q'_x \in X_a \wedge q'_y \in Y_a)\big)$$

This lemma gives the proof of the inclusion test algorithm Figure 4.9: the loop invariant of the algorithm is, as long as *is_included* is true: if $(X, Y) \in$ *ToDo*, then $X \mathcal{R} Y$.

[4]We also assume that all states are useful, *i.e.* all states of the automata are reachable from an initial state and coreachable from a final state. This assumption holds for all the algorithms of this chapter.

Proof: On one hand, if there is a simulation like this one, it is obvious that $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$, because of the definition of a language recognized by a lattice automaton.

On the other hand, if $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$, we can build $\mathcal{R}$ starting from $Q_0 \mathcal{R} Q'_0$ and using the same construction as the one of Figure 4.9. Indeed, this algorithm tries to build $\mathcal{R}$ as long as *is_included* is true. The sets X_a and Y_a are set of states reachable from X and Y by a transition labeled by $\lambda \sqsupseteq a$.

So the loop invariant of our construction is if $(X, Y) \in ToDo$, then $X \mathcal{R} Y$. We suppose that we have $X \mathcal{R} Y$ and, there is one atom $a \in At(\Lambda)$ so that there is not any sets of states X_a and Y_a verifying the conditions imposed by $\mathcal{R}$. It means we have either:

- at least one transition $(q_x, \lambda_x, q'_x) \in \delta : q_x \in X \wedge a \sqsubseteq \lambda_x$ and no transition $(q_y, \lambda_y, q'_y) \in \delta' : q_y \in Y \wedge a \sqsubseteq \lambda_y$, or
- a state $q'_x \in Q_f$ and no state $q'_y \in Q'_f$.

In both case, we have at least a word recognized by $\mathcal{A}$ and not recognized by $\mathcal{A}'$, which violates the hypothesis $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$. The assumption that all states of the two automata are reachable from an initial state and co-reachable from a final state ensures that, in the first case, we will eventually find such a word. □

Note that the algorithm for inclusion test does not need any determinization step. This determinization is implicitly performed on the fly during the algorithm.

4.2 Normalization of PLAs

Normalization of partitioned lattice automata will be obtained by merging, determinizing and minimizing PLA. These operations provide a canonical form for PLA. Although this normalization induces an upper-approximation of the recognized language, this is a robust notion in the sense that the approximation is optimal.

4.2.1 Merging

Any PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ recognizing a language L can be transformed into a merged PLA $\mathcal{A}_m = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta_m \rangle \sqsupseteq \mathcal{A}$ recognizing a language $L_m \supseteq L$ by merging transitions as follows:

$$\frac{q, q' \in Q \quad \sigma \in \Sigma \quad \lambda_m = \bigsqcup \{ \lambda \sqcap \pi(\sigma) \mid (q, \lambda, q') \in \delta \}}{(q, \lambda_m, q') \in \delta_m}$$

The upper-approximation on the recognized language comes from the use of the *lub* operator. For instance, with a single equivalence class on $\mathbb{R}^2$, the merged PLA associated to any automaton depicted on Fig. 4.6 is depicted on Fig. 4.8.

Algorithm: Inclusion test for two PLAs
Input: two PLAs $\mathcal{A} = \langle \Lambda, \pi : \Sigma \to \Lambda, Q, Q_0, Q_f, \delta \rangle$ and $\mathcal{A}' = \langle \Lambda, \pi : \Sigma \to \Lambda, Q', Q'_0, Q'_f, \delta' \rangle$
Output: a boolean is_included
begin
  is_included := true ;
  $\mathcal{R} := \{(Q_0, Q'_0)\}$;
  $ToDo := \{(Q_0, Q'_0)\}$;
  while $ToDo \neq \emptyset \wedge$ is_included **do**
    $(X, Y) :=$ pickAndRemoveElement($ToDo$);
    for all $a \in At(\Lambda)$ **do** [1]
      $X_a := \emptyset$; $Y_a := \emptyset$;
      for all $(q_x, \lambda_x, q'_x) \in \delta$ such that $q_x \in X$ and $a \sqsubseteq \lambda_x$ **do**
        $X_a := X_a \cup \{q'_x\}$;
      endfor
      for all $(q_y, \lambda_y, q'_y) \in \delta'$ such that $q_y \in Y$ and $a \sqsubseteq \lambda_y$ **do**
        $Y_a := Y_a \cup \{q'_y\}$;
      endfor
      is_included := is_included $\wedge$ $(Y_a = \emptyset \implies X_a = \emptyset) \wedge \left(X_a \cap Q_f \neq \emptyset \implies Y_a \cap Q'_f \neq \emptyset\right)$;
      if $(X_a, Y_a) \notin \mathcal{R}$ **then**
        $\mathcal{R} := \mathcal{R} \cup (X_a, Y_a)$;
        $ToDo := ToDo \cup (X_a, Y_a)$;
      endif
    endfor
  endwhile
  return is_included ;
end

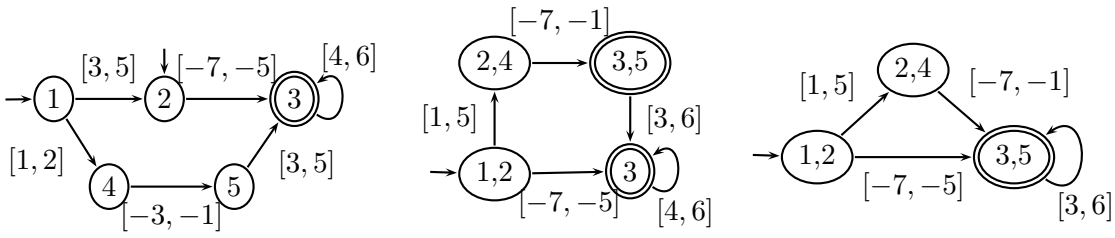
[1] The number of atoms may be infinite. So an implementation would manipulate elements $\lambda_x \sqcap \lambda_y$ for all outgoing transitions $(q_x, \lambda_x, q'_x) \in \delta, q_x \in X$ and $(q_y, \lambda_y, q'_y) \in \delta', q_y \in Y$.

Figure 4.9: Inclusion test algorithm for a merged PLA

4.2.2 Determinization

We give here an algorithm to obtain a deterministic automaton $\det(\mathcal{A})$ from a PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$. This algorithm mimics the determinization of the shape automaton using the subset construction on states and is illustrated on Fig. 4.10. The difference is that the transitions are merged in the course of the algorithm when they are labeled with values belonging to the same equivalence class. $\det(\mathcal{A}) = \langle \Lambda, \pi, \mathcal{X}, X_0, X_f, \Delta \rangle$ is characterized by:

- $\mathcal{X} = 2^Q$,
- $X_0 = \{Q_0\}$,
- $X_f = \{X \subseteq Q \mid X \cap Q_f \neq \emptyset\}$,
- Δ is the set of transitions (X_1, λ, X_2), defined for a couple $X_1, X_2 \subseteq Q$ and



(a) original automaton (b) deterministic automaton (c) minimal automaton

Figure 4.10: Determinization and minimization of an interval automaton with the partition $]-\infty, 0] \sqcup [0, +\infty[$

$\lambda \sqsubseteq \pi(\sigma)$ as:

$$\lambda = \sqcup\{\lambda \mid (q1, \lambda, q_2) \in \delta \wedge q_1 \in X_1 \wedge q_2 \in X_2 \wedge \lambda \sqsubseteq \pi(\sigma)\}$$

The full algorithm is given in Fig. 4.11. The resulting automaton $\det(\mathcal{A})$ is optimal in terms of language inclusion:

Proposition 4.3 (Determinizing PLA is a best upper-approximation) *Let $\mathcal{A}$ be a PLA and $\det(\mathcal{A})$ the PLA obtained with the algorithm of Fig. 4.11. Then $\det(\mathcal{A})$ is the best upper-approximation of $\mathcal{A}$ as a merged and deterministic*[5] *PLA:*

1. $\mathcal{A} \sqsubseteq \det(\mathcal{A})$*;*
2. *For any merged and deterministic PLA $\mathcal{A}'$ based on the same partition as $\mathcal{A}$, $\mathcal{A} \sqsubseteq \mathcal{A}' \implies \det(\mathcal{A}) \sqsubseteq \mathcal{A}'$.*

Proof: The result is a merged and deterministic PLA because the algorithm determinizes the shape and performs merging of new transitions. It is also clear that the resulting automaton recognizes a greater language than the original one. We must prove the third point of the proposition.

Let $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$ be the initial PLA, $\det(\mathcal{A}) = \langle \Lambda, \pi, \mathcal{X}, X_0, X_f, \Delta \rangle$ the result of our algorithm and $\mathcal{A}' = \langle \Lambda, \pi, Q', Q'_0 = \{q'_0\}, Q'_f, \delta' \rangle$ a merged and deterministic PLA such that $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$. As usual, we suppose that, in these automata, all states are reachable from the initial states and co-reachable form the final states.

We can build a simulation relation $\mathcal{R}$ on $\wp(Q) \times Q'$[6] defined as:

- At the beginning, $(Q_0, q'_0) \in \mathcal{R}$.
- If we have $(Q_x, q_x) \in \mathcal{R}$ then for all $\sigma \in \Sigma$, we consider $Q_y = \{q_y \in Q | \exists (q_x, \lambda, q_y) \in \delta \wedge q_x \in Q_x \wedge \lambda \sqsubseteq \pi(\sigma)\}$. Since $L_{\mathcal{A}} \subseteq L_{\mathcal{A}'}$ and $\mathcal{A}'$ is deterministic, there is a unique $q'_y \in Q'$ verifying $(q_x, \lambda', q'_y) \in \delta' \wedge \lambda' \sqsubseteq \pi(\sigma)$, thus we add $(Q_y, q'_y) \in \mathcal{R}$. By construction of $\det(\mathcal{A})$, there is also a transition $(Q_x, \lambda_{\det}, Q_y) \in \Delta$ with $\lambda_{\det} = \bigsqcup\{\lambda | \exists (q_x, \lambda, q_y) \in \delta \wedge q_x \in Q_x \wedge \lambda \sqsubseteq \pi(\sigma)\} \sqsubseteq \lambda'$.

[5] Remember that "deterministic" means strongly deterministic.

[6] We remind that $\mathcal{X} = \wp(Q)$.

Algorithm: Determinization of a PLA
Input: a PLA $\mathcal{A} = \langle \Lambda, \pi : \Sigma \to \Lambda, Q, Q_0, Q_f, \delta \rangle$ whose states are co-reachable
Output: a merged deterministic PLA $\mathcal{A}' = \langle \Lambda, \pi, X, X_0, X_f, \Delta \rangle$

```
begin
  X := {Q_0}; X_0 := {Q_0}; X_f := ∅;
  Δ := ∅;
  ToDo := {Q_0};
  while ToDo ≠ ∅ do
    x := pickAndRemoveElement(ToDo);
    for all σ ∈ Σ do
      λ_u := ⊥; x' := ∅;
      for all (q, λ, q') ∈ δ such that q ∈ x and λ ⊓ π(σ) ≠ ⊥ do
        λ_u := λ_u ⊔ (λ ⊓ π(σ)); x' := x' ∪ {q'};
      endfor
      if λ_u ≠ ⊥ then
        Δ := Δ ∪ {(x, λ_u, x')};
        if x' ∉ X then
          X := X ∪ {x'};
          if x' ∩ Q_f ≠ ∅ then X_f := X_f ∪ {x'};
          ToDo := ToDo ∪ {x'};
        endif
      endif
    endfor
  endwhile
end
```

Figure 4.11: Determinization algorithm for a merged PLA

We extend this simulation relation $\mathcal{R}$ to a simulation $\mathcal{R}' \subseteq \wp(\mathcal{X}) \times \wp(Q')$ satisfying the hypothesis of Lemma 4.1. Thus $L_{\det(\mathcal{A})} \subseteq \mathcal{A}'$ according to this lemma. □

Corollary 4.1 (The determinisation operation is an upper-closure operation) *The operation* det : PLA → PLA *is an upper-closure operation: it is* (*i*) *extensive:* $det(\mathcal{A}) \sqsupseteq \mathcal{A}$*;* (*ii*) *monotonic: for any* $\mathcal{A}$, $\mathcal{A}'$ *defined on the same partition,* $\mathcal{A} \sqsubseteq \mathcal{A}' \Rightarrow \det(\mathcal{A}) \sqsubseteq \det(\mathcal{A}')$*; and* (*iii*) *idempotent:* $\det(\det(\mathcal{A})) = \det(\mathcal{A})$.

Proof: The extensivity has already been shown. We have $\mathcal{A} \sqsubseteq \mathcal{A}' \sqsubseteq \det(\mathcal{A}')$. $\det(\mathcal{A})$ being the best approximation of $\mathcal{A}$ as a deterministic merged PLA, $\det(\mathcal{A}) \sqsubseteq \det(\mathcal{A}')$, which proves the monotonicity. The idempotence is a trivial consequence of Prop. 4.3. □

Corollary 4.2 *If* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *are two merged PLAs recognizing the same language* L*, then* $\det(\mathcal{A}_1)$ *and* $\det(\mathcal{A}_2)$.

Proof: $det(\mathcal{A}_2)$ is a merged and deterministic PLA and $L_{\mathcal{A}_1} = L \subseteq L_{det(\mathcal{A}_2)}$, so we have $L_{det(\mathcal{A}_1)} \subseteq L_{det(\mathcal{A}_2)}$. The same holds when switching $\mathcal{A}_1$ and $\mathcal{A}_2$, so $det(\mathcal{A}_1)$ and $det(\mathcal{A}_2)$ are two deterministic automata recognizing the same language. □

4.2.3 Minimization

We use for PLA a notion of minimization based on its shape automaton, in the same spirit as for the notion of determinism.

Definition 4.8 *A PLA is* minimal *or* normalized *if it is merged and if its shape automaton is minimal and deterministic. A normalized PLA will be also called a NLA (normalized lattice automaton).*

The algorithm to minimize a PLA consists in removing its unconnected states, in determinizing it according to the previous algorithm and in quotienting it w.r.t. the equivalence bisimulation relation, as defined on the states of its shape automaton (*cf.* Chapter 1). However, when quotienting the states of a PLA, transitions labeled with elements belonging to the same equivalence class are merged, which may induce an over-approximation of the recognized language, as shown on Fig. 4.10.

Definition 4.9 (Quotient PLA) *Given a merged PLA* $\langle \Lambda, \pi : \Sigma \to \Lambda, Q, Q_0, Q_f, \delta \rangle$ *and an equivalence relation* $\approx$ *on the set of states* Q*, the* quotient automaton $\mathcal{A}/\approx = \langle \Lambda, \pi, \widetilde{Q}, \widetilde{Q_0}, \widetilde{Q_f}, \widetilde{\delta} \rangle$ *is defined by:*

- $\widetilde{Q} = Q/\approx$*, the set of equivalence classes;*
- $\widetilde{Q_0} = \{\widetilde{q} | q \in Q_0\}$ *and* $\widetilde{Q_f} = \{\widetilde{q} | q \in Q_f\}$*;*
- $\widetilde{\delta}$ *is defined by the rule:*

$$\frac{\sigma \in \Sigma \quad \lambda_u = \bigsqcup\{\lambda \sqsubseteq \pi(\sigma) \mid \exists q_0 \in \widetilde{q}, \exists q_0' \in \widetilde{q'} : (q_0, \lambda, q_0') \in \delta\}}{(\widetilde{q}, \lambda_u, \widetilde{q'}) \in \widetilde{\delta}}$$

Note that the quotient automaton is a merged PLA.

Theorem 4.1 (Minimizing PLA is a best upper-approximation) *For any PLA* $\mathcal{A}$*, there is a unique (up to isomorphism) NLA* $\widehat{\mathcal{A}}$ *based on the same partition* π *such that*

1. $\mathcal{A} \sqsubseteq \widehat{\mathcal{A}}$*,*
2. *for any NLA* $\mathcal{A}'$ *based on the partition* π*,* $\mathcal{A} \sqsubseteq \mathcal{A}' \implies \widehat{\mathcal{A}} \sqsubseteq \mathcal{A}'$*.*

Proof: The minimization algorithm consists first in determinizing the automaton, and then, in quotienting the result by the largest bisimulation relation on its states induced by its shape automaton. The result is merged and deterministic and the quotient operation ensures that its shape is minimal and deterministic. Thus it is minimal according to Def. 4.8.

Then we show that if two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ recognize the same language L then $L_{\min(\mathcal{A}_1)} = L_{\min(\mathcal{A}_2)}$. According to Corollary 4.2, $\det(\mathcal{A}_1)$ and $\det(\mathcal{A}_2)$ recognize the same language, and so does $\text{shape}(\det(\mathcal{A}_1))$ and $\text{shape}(\det(\mathcal{A}_2))$.

Moreover $\text{shape}(\min(\mathcal{A}_1))$ and $\text{shape}(\min(\mathcal{A}_2))$ are isomorphic, because there is at most one minimal deterministic automaton and the minimization of the automaton is also a minimization of its shape.

We extend this isomorphism to $\min(\mathcal{A}_1)$ and $\min(\mathcal{A}_2)$, and, using the definition of the Myhill-Nerode equivalence relation and the fact that $L_{\det(\mathcal{A}_1)} = L_{\det(\mathcal{A}_2)}$, we prove the equality of languages. □

Definition 4.10 *Let us fix a partitioning function $\pi : \Sigma \to \Lambda$. For any language $L \in \text{Reg}(\Lambda)$ recognized by a lattice automaton $\mathcal{A}$, $\widehat{L}$ will denote the language recognized by the unique NLA $\widehat{A}$ verifying the properties of Thm. 4.1.*

Corollary 4.3 (The normalisation operation is an upper-closure operation) *The function $\widehat{\cdot} : \text{PLA} \to \text{NLA} \subseteq \text{PLA}$ is an upper-closure operator: it is extensive, monotonic (given a fixed partition) and idempotent.*

Proof: Based on the same principle as Corollary 4.1. □

Corollary 4.4 *For any languages L_1 and L_2,*

$$\widehat{L_1} \cup \widehat{L_2} \subseteq \widehat{L_1 \cup L_2} \tag{4.1}$$

$$\widehat{L_1 \cap L_2} \subseteq \widehat{L_1} \cap \widehat{L_2} \tag{4.2}$$

The inclusion is strict, because of the use of least upper bounds during normalization. To see this, consider the lattice of intervals on rationals, partitioned with the trivial partition of size 1. Take $L_1 = 0$, $L_2 = 2$. One has $\widehat{L_1} = L_1$, $\widehat{L_2} = L_2$, $\widehat{L_1} \cup \widehat{L_2} = 0+2$, but $\widehat{L_1 \cup L_2} = \sum_{x\in[0,2]} x$. Take now $L_1 = 0 + 2$ and $L_2 = 1 + 3$. One has $\widehat{L_1} = \sum_{x\in[0,2]} x$, $\widehat{L_2} = \sum_{x\in[1,3]} x$, $\widehat{L_1} \cap \widehat{L_2} = \sum_{x\in[1,2]} x$, but $\widehat{L_1 \cap L_2} = \emptyset$.

Complexity. Let $\mathcal{A}$ be a lattice automaton with n states, m transitions and a partition of size p (i.e. $|\Sigma| = p$). The complexity of the previous algorithms is given on Tab. 4.12, where the operations on Λ are considered as atomic.

Algorithm	Complexity (time)	Complexity (space)
inclusion test [1]	$O(2^{2n} \cdot m^2)$	$O(2^n)$
inclusion test [2]	$O(n \log n + m)$	$O(n \log n)$
merging	$O(m \cdot p)$	$O(n + m)$
determinization	$O(2^n \cdot m)$	$O(2^n)$
quotienting[3]	$O(n + m)$ + merging	$O(n + m)$ + merging
minimization[4]	$O(n \log n)$ + quotienting	$O(n \log n)$ + quotienting
normalization	determinization + minimization	determinization + minimization

Figure 4.12: Complexity of the basic operations on PLAs, where n is the number of states of the automaton an m the number of transitions

[1] Algorithm presented in Figure 4.9,
[2] Algorithm operating on minimal deterministic automata,
[3] If $\approx$ is known,
[4] The input automaton is assumed to be deterministic.

Proof: The inclusion algorithm described in Figure 4.9 operates on couples of sets of states, so the while loop is iterated at most $2^n \times 2^{n'}$ times, where n (resp. n') is the number of states of the first automaton (resp. the second automaton). Assuming $n > n'$, the memory needed is $O(2^n)$. In each loop, the actual algorithm considers all couples of transitions $(q_x, \lambda_x, q'_x) \in \delta$ and $(q_y, \lambda_y, q'_y) \in \delta'$. The worse case complexity is thus $O(2^{2n} \cdot m^2)$.

If the automata are minimal, we re-employ the inclusion test algorithm for finite automata ; one first check the inclusion of the shape automata and, if the answer is positive, one can check the matching transitions.

Merging is just splitting each transition according to the partition of size p.

The determinization algorithm is detailed in Figure 4.11. Like the algorithm for finite automata, the states of $\det(\mathcal{A})$ are sets of states of $\mathcal{A}$, thus it needs at most $O(2^n)$ memory space. There are at most $O(2^n)$ iterations of the while loops, and the for loop is iterated at most m times.

The quotient creates at most n new states and m new transitions and merge the new automaton. The minimization of a deterministic automaton is done by computing the classes of equivalence of the auto-bisimulation (complexity: $O(n \log n)$) and then quotienting the automaton. □

Thm. 4.1 defines a normalization for languages recognized by PLA. The set of NLA defined on Λ with the partition π will be denoted by $\mathrm{Reg}(\Lambda, \pi)$, which denotes also the corresponding set of recognized languages.

Having defined a normal form for PLA, we can now specify classical operations on languages recognized by NLA by defining them on their automata. But first, we study how to refine the partitioning function π.

4.2.4 Refinement of the Partitioning Function

In the previous paragraphs, the partitioning function $\pi : \Sigma \to \Lambda$ was fixed. The precision of the approximations made during the merging, determinization and minimization operations depends on the fineness of the partitioning function. For example, all outgoing transitions from a given state would be merged during the determinization algorithm employed with the trivial partition of size 1.

Definition 4.11 *A partitioning function* $\pi_2 : \Sigma_2 \to \Lambda$ refines *a partitioning function* $\pi_1 : \Sigma_1 \to \Lambda$ *if:*

$$\forall \sigma_2 \in \Sigma_2, \exists \sigma_1 \in \Sigma_1 : \pi_2(\sigma_2) \sqsubseteq \pi_1(\sigma_1)$$

Let $\mathcal{A}_1 = \langle \Lambda, \pi_1 : \Sigma_1 \to \Lambda, Q, Q_0, Q_f, \delta_1 \rangle$ *be a PLA. The automaton* $\mathcal{A}_2 = \langle \Lambda, \pi_2 : \Sigma_2 \to \Lambda, Q, Q_0, Q_f, \delta_2 \rangle$ refines $\mathcal{A}_1$ *if* π_2 *refines* π_1 *and the transitions of* δ_2 *are obtained by:*

$$\frac{(q, \lambda_1, q') \in \delta_1 \qquad \sigma_2 \in \Sigma_2 \qquad \lambda_2 = \lambda_1 \sqcap \pi_2(\sigma_2)}{(q, \lambda_2, q') \in \delta_2}$$

Refining an automaton does not modify immediately the recognized language, but leads to a more precise upper-approximation in merging, determinization and minimization operations.

Proposition 4.4 *Let* $\mathcal{A}_1$ *be a PLA and* $\mathcal{A}_2$ *a PLA refining* $\mathcal{A}_1$*. Then:*

1. $L_{\mathcal{A}_1} = L_{\mathcal{A}_2}$,
2. $merge(\mathcal{A}_1) \sqsupseteq merge(\mathcal{A}_2)$,
3. $det(\mathcal{A}_1) \sqsupseteq det(\mathcal{A}_2)$,
4. $\widehat{\mathcal{A}_1} \sqsupseteq \widehat{\mathcal{A}_2}$.

Proof:

1. Since $(\Lambda, \sqsubseteq)$ is an atomic lattice, we do not loose any atoms when splitting the transitions.
2. For two states $q, q' \in Q$ and for any $\sigma_1 \in \Sigma_1$, $\sigma_2 \in \Sigma_2$ such that $\pi_2(\sigma_2) \sqsubseteq \pi_1(\sigma_1)$, we have :

   $$\lambda_{m_2} = \bigsqcup \{ \lambda \sqcap \pi_2(\sigma_2) \mid (q, \lambda, q') \in \delta_1 \} \sqsubseteq \lambda_{m_1} = \bigsqcup \{ \lambda \sqcap \pi_1(\sigma_1) \mid (q, \lambda, q') \in \delta_1 \}$$

   where λ_{m_i} labels the transition obtained by merging the transitions between q_1, q_2 labeled with elements in $\pi_i(\sigma_i)$.
3. Idea of the proof : during each iteration of the **while** loop, the determinization of the automaton $\mathcal{A}_2$ merges less states and less transitions than the determinization of the automaton $\mathcal{A}_1$.

4. Idea of the proof : the equivalence class of the bisimulation relation (for the quotienting algorithms) are also more precise with π_2, so both determinization and quotienting merge less states and transitions of the original automaton.

□

Refining the partition π makes the class of normalized languages $\mathrm{Reg}(\Lambda, \pi)$ more expressive.

Corollary 4.5 *Let π_1 and π_2 be two partitioning functions for Λ, with π_2 refining π_1. Then any language recognized by a NLA built on the partition π_1 is also recognized by a NLA built on the partition π_2:* $\mathrm{Reg}(\Lambda, \pi_1) \subseteq \mathrm{Reg}(\Lambda, \pi_2)$.

Proof: Let $\mathcal{A}_1 \in \mathrm{Reg}(\Lambda, \pi_1)$ be a NLA. Its refinement $\mathcal{A}_2$ according to π_2 (*cf.* Def. 4.11) is normalized, and thus $\mathcal{A}_2 \in \mathrm{Reg}(\Lambda, \pi_2)$. □

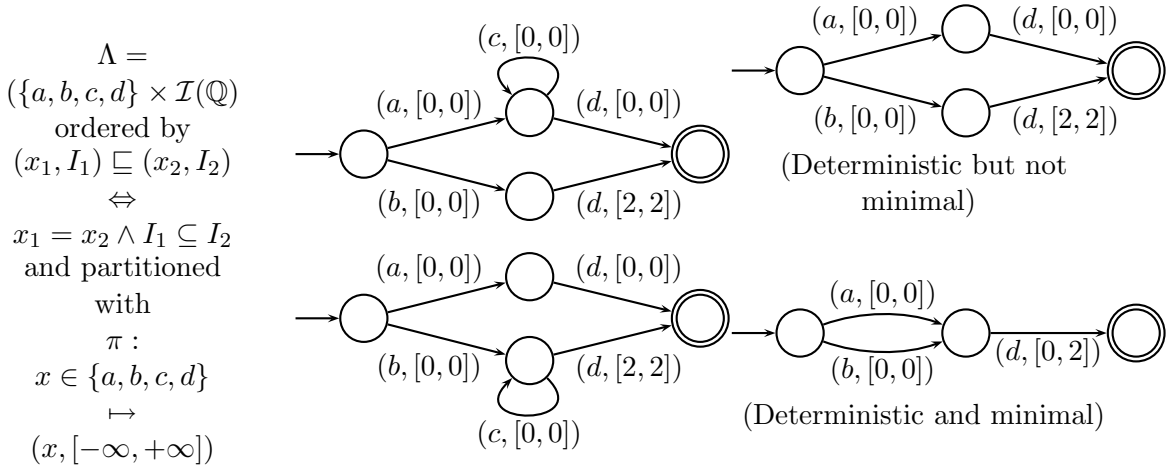
Choosing an adequate partitioning function is thus important. For the analysis of SCMs, where data messages are usually composed of a message type and some parameters, the type of the message defines a natural partition. When this standard partition is not sufficient for the analysis, it can be refined to a more adequate partition. In this sense, the abstraction refinement techniques based on partitioning (see for instance [Jea03, MR05]) are applicable to lattice automata.

4.3 Operations on PLAs

After the definition of a robust normalization concept, we can now define the classical operations on languages (union, intersection, ...) by defining them on lattice automata. The set operations (inclusion, union and intersection) and the normalization of their result define respectively the partial order, the least upper bound and the greatest lower bound of the lattice of NLAs. We also present operations like the concatenation or the left derivation ; those operations define the abstract semantics of the model presented in Chapter 5. Finally, we define a widening operator, which guarantees that the analysis presented in Chapter 5 terminates.

4.3.1 Set Operations

Inclusion test. Two PLAs that are not normalized can be compared for language inclusion using a simulation relation taking into account the partial order of the lattice, *cf.* lemma 4.1. If they are normalized, one can first compare for inclusion their shape automata, and in case of inclusion, one can compare the labels of matching transitions. If they are not normalized, the previous inclusion test algorithm works, but is more expensive.



(a) Lattice (b) The two minimized automata to be intersected (c) Intersection and its approximation

Figure 4.13: Intersection of normalized PLAs and its upper-approximation

Union and least upper bound. The exact union of two PLAs $\mathcal{A}_i = \langle \Lambda, \pi, Q^i, Q^i_0, Q^i_f, \delta^i \rangle$ can be computed very simply as the disjoint union of the two PLAs; this produces the PLA:

$$\mathcal{A} = \langle \Lambda, \pi, Q^1 \cup Q^2, Q^1_0 \cup Q^2_0, Q^1_f \cup Q^2_f, \delta^1 \cup \delta^2 \rangle$$

which is not deterministic.

Normalizing $\mathcal{A}$ transforms the exact union operation into an upper bound operator $\sqcup$ defined as follows: $\mathcal{A}_1 \sqcup \mathcal{A}_2 = \widehat{\mathcal{A}_1 \cup \mathcal{A}_2}$. As a corollary of Theorem 4.1, this upper bound operator is actually a *least upper bound operator* on the set of NLA ordered by language inclusion.

Intersection and its normalized upper-approximation. The exact intersection of two PLAs $\mathcal{A}_i = \langle \Lambda, \pi, Q^i, Q^i_0, Q^i_f, \delta^i \rangle$ can be computed as a product of the two PLAs; this produces a PLA $\mathcal{A} = \langle \Lambda, \pi, Q^1 \times Q^2, Q^1_0 \times Q^2_0, Q^1_f \times Q^2_f, \delta \rangle$ where δ is defined by:

$$\frac{(q_1, \lambda_1, q'_1) \in \delta_1 \quad (q_2, \lambda_2, q'_2) \in \delta_2 \quad \lambda_1 \sqcap \lambda_2 \neq \bot}{((q_1, q_2), \lambda_1 \sqcap \lambda_2, (q'_1, q'_2)) \in \delta}$$

We implicitly remove unconnected states in $\mathcal{A}$. If the input automata are normalized, $\mathcal{A}$ is merged and deterministic, but not necessarily minimal. Minimizing it may induce an upper-approximation, as shown by Fig. 4.13. Thus, NLAs are not closed under (exact) intersection.

These operations allow to equip the set of NLAs with a join semi-lattice structure.

Proposition 4.5 (NLAs as a join semilattice) *The set of NLA defined on an atomic lattice Λ with a fixed partition π, ordered by language inclusion, is a join semi-lattice: it has bottom and top elements, and a least upper bound operator.*
If the standard lattice operations on Λ are computable, so are the corresponding operations on the join semi-lattice of NLA.

Proof: It is clear that the bottom and top elements of this semi-lattice are respectively the empty automaton $\langle \Lambda, \pi, Q = \emptyset, Q_0 = \emptyset, Q_f = \emptyset, \delta = \emptyset \rangle$ recognizing no words, and the universal automaton $\langle \Lambda, \pi, Q = \{q\}, Q_0 = Q, Q_f = Q, \delta = \bigcup_{\sigma \in \Sigma} (q, \pi(\sigma), q) \rangle$ recognizing any word on the alphabet $At(\Lambda)$. The least upper bound operator is the operator $\sqcup$ on NLA defined above. Last, we have also given algorithms for testing inclusion and for computing the least upper bound. □

Remark 4.4 (Deterministic merged PLA as a full lattice) *If we consider the set of deterministic and merged (but not necessarily minimal) PLA, then this set partially ordered by language inclusion has a true lattice structure, instead of just the join semi-lattice structure of NLA. Indeed, Proposition 4.3 may be used to prove that the determinization of the exact union is a least upper bound operation, and as the exact intersection of two deterministic merged PLAs is a deterministic merged PLA, this intersection is trivially a greatest lower bound.*

One could manipulate deterministic merged PLA instead of NLA. But there may be several deterministic merged PLA recognizing the same language, hence the isomorphism between languages and automata is lost. The other drawbacks of such a choice is that testing inclusion is more expensive and that one cannot extend the widening operation on finite automata defined in Chapter 3.

4.3.2 Other Language Operations

We define here some language operations that will be employed in the next chapter. They are similar to the ones defined for finite automata in Chapter 3.

Language concatenation. Language concatenation on deterministic PLA is performed exactly as for finite automata, by substituting to the final states of the first automaton a copy of the initial state of the second automaton. The obtained automaton is non-deterministic in general and requires normalization. Language concatenation can also be computed on non deterministic PLA, by using ϵ transitions and ϵ-closure to remove them.

Left derivation. In the finite case, the left derivation [Brz64] of a finite automaton is performed w.r.t. a letter. The corresponding operation would consist in deriving a PLA according to an equivalence class. However, the partition is a way to control the

precision of the approximations performed by the various operations on PLA, and it does not have a semantic meaning.

As a consequence, we define a more general left derivation L/λ operator as follows:

$$\begin{array}{rcccl} \cdot/\cdot & : & \mathrm{Reg}(\Lambda) \times \Lambda & \to & \mathrm{Reg}(\Lambda) \\ & & (L, \lambda) & \mapsto & \{\omega \in At(\Lambda)^* \mid \exists a \sqsubseteq \lambda : a \cdot \omega \in L\} \end{array}$$

We clearly have the identity $L/\lambda = \sum_{a \in At(\lambda)} L/a$, from which we can deduce the following proposition:

Proposition 4.6 *Let $\lambda \in \Lambda$ and $(\lambda_\sigma)_{\sigma \in \Sigma}$ be the projection of λ on the partition defined by $\pi : \Sigma \to \Lambda$. Then $L/\lambda = \sum_{\sigma \in \Sigma} L/\lambda_\sigma$.*

The left derivation can be implemented exactly on a deterministic PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0 = \{q_0\}, Q_f, \delta \rangle$ as follows:

$$\begin{array}{rcccl} \cdot/\cdot & : & \mathrm{PLA} \times \Lambda & \to & (\Sigma \to \mathrm{PLA}) \\ & & (L, \lambda) & \mapsto & (\ \sigma \mapsto \mathcal{A}/(\lambda \sqcap \pi(\sigma))\ ) \end{array}$$

where $\mathcal{A}/(\lambda \sqcap \pi(\sigma))$ is the empty automaton if $\lambda \sqcap \pi(\sigma) = \bot$, and the deterministic PLA $\langle \Lambda, \pi, Q, Q_0 = \{\delta(q_0, \sigma)\}, Q_f, \delta \rangle$ otherwise. One obtains a finite set of deterministic automata instead of a single one. This set can be of course upper-approximated by the least upper bound deterministic PLA, denoted by $\widehat{\mathcal{A}/\lambda} = \bigsqcup_{\sigma \in \Sigma} (\mathcal{A}/\lambda)(\sigma)$.

"First" and "Pop" operations. In addition to the left derivation, it may also be useful to extract the letters beginning the words of a language as follows:

$$\begin{array}{rcccl} \mathit{first} & : & \mathrm{Reg}(\Lambda) & \to & \wp(At(\Lambda)) \\ & & L & \mapsto & \{a \in At(\Lambda) \mid \exists w \in At(\Lambda)^*,\ a \cdot w \in L\} \end{array}$$

The equivalent algorithmic definition on a deterministic PLA $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0 = \{q_0\}, Q_f, \delta \rangle$ is:

$$\begin{array}{rcccl} \mathit{first} & : & \mathrm{PLA} & \to & (\Sigma \to \Lambda) \\ & & \mathcal{A} & \mapsto & (\sigma \mapsto \mathit{first}(\mathcal{A})(\sigma)) \end{array}$$

where for each $\sigma \in \Sigma$, $\mathit{first}(\mathcal{A})(\sigma)$ is the label of the unique transition $(q_0, \lambda, q) \in \delta$ with $\lambda \sqsubseteq \pi(\sigma)$, if it exists, and $\bot$ otherwise. It is clear that the set of atoms covered by $\mathit{first}(\mathcal{A})$ is exactly $\mathit{first}(L_\mathcal{A})$:

$$\bigcup_{\sigma \in \Sigma} \{a \in At(\Lambda) \mid a \sqsubseteq \mathit{first}(\mathcal{A})(\sigma)\} = \mathit{first}(L_\mathcal{A})$$

The *first* operation can be combined with the left derivation in order to implement the *pop* operation, which extracts the first letters and derives the corresponding "residue" automaton:

$$\begin{array}{rcccl} \mathit{pop} & : & \mathrm{PLA} \times \Lambda & \to & (\Sigma \to \Lambda \times \mathrm{PLA}) \\ & & (\mathcal{A}, \lambda) & \mapsto & \Big(\sigma \mapsto \big(\ \lambda \sqcap \mathit{first}(\mathcal{A})(\sigma)\ ,\ \mathcal{A}/(\lambda \sqcap \pi(\sigma))\ \big)\Big) \end{array}$$

The complexity of the previous operations is summarized on Tab. 4.1.

NLA operation	complexity	resulting PLA	exact operation ?
inclusion test	$O(n \cdot \log n)$	—	yes
union	$O(1)$	non-deterministic	yes
intersection	$O(n^2 + m^2)$	deterministic, non minimal	yes
language concatenation	$O(n \cdot p)$	non-deterministic	yes
letter right-concatenation	$O(n)$	non-deterministic	yes
letter left-concatenation	$O(1)$	normalized	yes
left derivation	$O(p)$	non-deterministic	yes
right derivation	$O(n \cdot p)$	deterministic, non minimal	yes
first	$O(p)$	—	yes
pop	$O(p)$	non-deterministic	yes
normalized union ($\sqcup$)	$O(2^n)$	normalized	no
normalized intersection ($\sqcap$)	$O(n^2 \log n)$	normalized	no
normalized pop	$O(p \cdot 2^n)$	normalized	no

Table 4.1: Complexity of various operations on a NLA. *We assume that the NLA has n states, m transitions and a partition of size p, and that the operations on Λ have a fixed cost. (i.e. $|\Sigma| = p$)*

4.3.3 Widening on NLAs

The widening on NLAs we define here combines the widening on finite automata of Chapter 3 with the standard widening operator $\nabla_\Lambda : \Lambda \times \Lambda \to \Lambda$ that we assume for the atomic lattice Λ. If a widening operator is not strictly required for Λ (because Λ satisfies the ascending chain condition), then ∇_Λ can be defined as the least upper bound $\sqcup$.

We first extend the ρ_k operator defined on finite automata in Chapter 3.

Definition 4.12 (Operator ρ_k on NLAs) *Let $\mathcal{A} = (\Lambda, \pi, Q, Q_0, Q_f, \Delta)$ be a NLA, $\text{shape}(\mathcal{A}) = (\Sigma, Q, Q_0, Q_f, \delta)$ its shape automaton. Let $k \geq 0$ be an integer and $\approx_k$ the k-depth bisimulation relation on Q induced by the partition of Q into $Q_0 \cap Q_f$, $Q_0 \setminus Q_f$, $Q_f \setminus Q_0$ and $Q \setminus (Q_0 \cup Q_f)$ and the transition relation $\delta \subseteq Q \times \Sigma \times Q$. We define $\rho_k(\mathcal{A})$ as the quotient PLA $\mathcal{A}/\approx_k$.*

The widening operator we suggest consists in applying the operator ρ_k when the two argument automata have a different shape automaton, and to apply the widening operator of the lattice Λ on their matching transitions when the two argument automata have the same shape automaton, as illustrated by Fig. 4.14.

Definition 4.13 (Widening on NLAs) *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two NLA defined on the same partition π with $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. The widening operator ∇_k is defined as :*

$$\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \begin{cases} \widehat{\rho_k(\mathcal{A}_2)} & \text{if } \text{shape}(\mathcal{A}_1) \neq \text{shape}(\rho_k(\mathcal{A}_2)) \\ \mathcal{A}_1 \nearrow \mathcal{A}_2 & \text{otherwise (which implies } \text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2)) \end{cases}$$

where $\mathcal{A}_1 \nearrow \mathcal{A}_2$ is the NLA $\mathcal{A}$ which has the same set of states as $\mathcal{A}_1$ and $\mathcal{A}_2$ and the

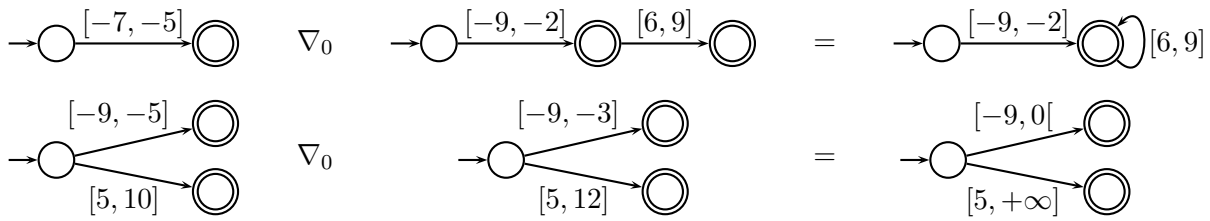


Figure 4.14: Widening on interval PLA, with a partition $[-\infty, 0[\sqcup [0, +\infty]$ and $k = 0$

set of transitions δ defined by the rule:

$$\frac{\sigma \in \Sigma \quad (q, \lambda_1, q') \in \delta_1 \quad (q, \lambda_2, q') \in \delta_2 \quad \lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)}{(q, (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma), q') \in \delta}$$

If $\mathcal{A}_1 \not\sqsubseteq \mathcal{A}_2$, then $\mathcal{A}_1 \nabla_k \mathcal{A}_2 \triangleq \mathcal{A}_1 \nabla_k (\mathcal{A}_1 \sqcup \mathcal{A}_2)$.

Notice that with the other hypothesis, the condition $\text{shape}(\mathcal{A}_1) \neq \text{shape}(\rho_k(\mathcal{A}_2))$ is equivalent to $\text{shape}(\mathcal{A}_1) \subsetneq \text{shape}(\rho_k(\mathcal{A}_2))$, and its negation $\text{shape}(\mathcal{A}_1) = \text{shape}(\rho_k(\mathcal{A}_2)) \supseteq \text{shape}(\mathcal{A}_2)$ implies $\text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2)$.

Theorem 4.2 *∇_k is a proper widening operator:*

1. *For any NLAs $\mathcal{A}1, \mathcal{A}_2$ defined on the same partition such that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1 \nabla_k \mathcal{A}_2$;*
2. *If there is an increasing chain of NLAs $\mathcal{A}_0 \sqsubseteq \mathcal{A}_1 \sqsubseteq \ldots \sqsubseteq \mathcal{A}_n \sqsubseteq \ldots$, the chain $\mathcal{A}'_0 \sqsubseteq \mathcal{A}'_1 \sqsubseteq \ldots \sqsubseteq \mathcal{A}'_n \sqsubseteq \ldots$ defined as $\mathcal{A}'_0 = \mathcal{A}_0$ and $\mathcal{A}'_{i+1} = \mathcal{A}'_i \nabla_k (\mathcal{A}'_i \sqcup \mathcal{A}_{i+1})$ is not strictly increasing.*

Proof:

1. We have $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ by hypothesis. If $\text{shape}(\mathcal{A}_1) \neq \rho_k(\text{shape}(\mathcal{A}_2)) = \text{shape}(\rho_k(\mathcal{A}_2))$, $\mathcal{A}_1 \nabla_k \mathcal{A}_2 = \widehat{\rho_k(\mathcal{A}_2)} \sqsupseteq \mathcal{A}_2$. If $\text{shape}(\mathcal{A}_1) = \text{shape}(\mathcal{A}_2) = \rho_k(\text{shape}(\mathcal{A}_2))$, $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1 \nearrow \mathcal{A}_2$ because for each pair of matching transitions (q, λ_1, q') and (q, λ_2, q') with $\lambda_1, \lambda_2 \sqsubseteq \pi(\sigma)$, we have $\lambda_2 \sqsubseteq (\lambda_1 \nabla_\Lambda \lambda_2) \sqcap \pi(\sigma)$ (as ∇_Λ is a widening operator on Λ).

2. $(\mathcal{A}'_i)_{i \geq 0}$ is an increasing chain of NLAs because of the first property. Thus, $(S'_i = \text{shape}(\mathcal{A}'_i))_{i \geq 0}$ is an increasing chain of finite automata. Moreover, by definition of ∇_k on NLAs, $S'_{i+1} = S'_i \nabla_k (S'_i \sqcup S_{i+1}) = \rho_k(S'_i \sqcup S_{i+1})$. As ∇_k as defined on finite automata is a widening operator, the chain $(S'_i)_{i \geq 0}$ becomes stationary at some rank N. The chain $(\mathcal{A}'_i)_{i \geq N}$ is thus an increasing sequence of NLAs with identical shape automaton S. For each transition $(q, \sigma, q') \in \delta^S$, one can extract the corresponding transition sequence $(q, \lambda'_i, q') \in \delta^{\mathcal{A}'_i}$ with $\lambda'_i \sqsubseteq \pi(\sigma)$. The sequence $(\lambda'_i)_{i \geq N}$ converges after a finite number of steps, because ∇_Λ is a widening operator. As there is only a finite number of transitions in the shape automaton S, the sequence $(\mathcal{A}'_i)_{i \geq N}$ also converges after a finite number of steps.

□

4.3.4 NLAs as an Abstract Domain

Normalized lattice automata allows to define an *abstract domain functor* which lifts abstract domains for some set to abstract domains for languages on this set. More precisely, given an *atomic* abstract lattice $\Lambda \xrightarrow{\gamma_\Lambda} \wp(S)$ for some set S with the concretization function γ_Λ, and a partitioning function π for Λ, $\mathrm{Reg}(\Lambda, \pi)$ can be viewed as an abstract domain for $\mathcal{L}(S) = \wp(S^*)$, the languages on elements of S, with the concretization function

$$\begin{array}{rccl} \gamma: & \mathrm{Reg}(\Lambda,\pi) & \to & \wp(S^*) \\ & \mathcal{A} & \mapsto & \{s_0 \dots s_n \in S^* \mid a_0 \dots a_n \in L_{\mathcal{A}} \wedge \forall i : s_i \in \gamma_\Lambda(a_i)\} \end{array}$$

and the widening operator of Def. 4.13. $\mathrm{Reg}(\Lambda, \pi)$ is a non-complete join semi-lattice of infinite height, so we do not have a Galois connection.

We have in addition the following monotonicity result for $\mathrm{Reg}(\Lambda, \pi)$ seen as a functor.

Theorem 4.3 *Let $\gamma_i : \Lambda_i \to \wp(S), i = 1, 2$ be two abstract domains for $\wp(S)$ such that Λ_2 refines Λ_1, with $\gamma_{12} : \Lambda_1 \to \Lambda_2$ such that $\gamma_1 = \gamma_2 \circ \gamma_{12}$. Let π_1 a partitioning function for Λ_1, and π_2 a partitioning function for Λ_2 refining $\gamma_{12} \circ \pi_1$. The abstract domain $\mathrm{Reg}(\Lambda_2, \pi_2)$ refines $\mathrm{Reg}(\Lambda_1, \pi_1)$.*

This theorem is illustrated by Figure 4.15.

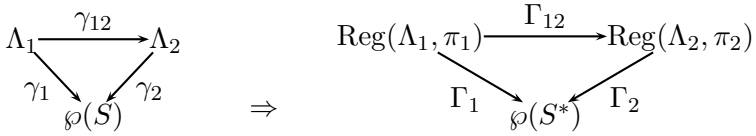


Figure 4.15: Refinement of abstract domains

Proof: Let $\Gamma_i : \mathrm{Reg}(\Lambda_i, \pi_i) \to \wp(S^*)$ be the two concretization functions induced by the functions γ_i. Let $\mathcal{A}_1 \in \mathrm{Reg}(\Lambda_1, \pi_1)$ be a NLA. We can build a NLA $\mathcal{A}_2 \in \mathrm{Reg}(\Lambda_2, \pi_2)$ such that $\gamma_2(\mathcal{A}_2) = \gamma_1(\mathcal{A}_1)$.

We first build and automaton $\mathcal{A}'_2$ which has the same states as $\mathcal{A}_1$, and a set of transitions defined by the rule:

$$\frac{(q, \lambda_1, q') \in \delta_1 \quad \lambda_2 = \gamma_{12}(\lambda_1)}{(q, \lambda_2, q') \in \delta'_2}$$

Since there is σ_1 such that $\lambda_1 \sqsubseteq \pi_1(\sigma_1)$, then $\lambda_2 \sqsubseteq \gamma_{12} \circ \pi_1(\sigma_1)$. The set of elements in S "covered" by the two labels is clearly the same. The resulting automaton $\mathcal{A}'_2$ thus satisfies $\Gamma_2(\mathcal{A}'_2) = \Gamma_1(\mathcal{A}_1)$. Moreover, $\mathcal{A}'_2$ is a NLA belonging to $\mathrm{Reg}(\Lambda_2, \gamma_{12} \circ \pi_2)$. We now define $\mathcal{A}_2$ as the refinement of $\mathcal{A}'_2$ w.r.t. the refined partition π_2, which recognizes the same language according to Prop. 4.4.

This transformation from $\mathcal{A}_1$ to $\mathcal{A}_2$ defines a function $\Gamma_{12} : \mathrm{Reg}(\Lambda_1, \pi_1) \to \mathrm{Reg}(\Lambda_2, \pi_2)$ such that $\Gamma_1 = \Gamma_2 \circ \Gamma_{12}$. □

Most language operations can be efficiently abstracted in $\mathrm{Reg}(\Lambda, \pi)$. This is in particular the case of language operations corresponding to the operations offered by the *FIFO queue* or *stack* abstract datatypes. Hence, $\mathrm{Reg}(\Lambda, \pi)$ is a suitable abstract domain for FIFO queues or stacks on elements of S, as it is shown in the following chapter.

4.4 Conclusion

The lattice automata, as defined in this chapter, are a suitable representation for regular languages over an infinite alphabet which is given a lattice structure. We explained the need for a given partition of the atoms, due to some annoying aspects of the basic definition of lattice automata. For example, "normal" lattice automata have an unbounded branching degree whereas *partitioned lattice automata* have a branching degree at most equals to $m \times n$, where m is the size of the partition and n the size of the automaton.

We defined a determinization algorithm that mimics the determinization of finite automata. The result of this algorithm is a deterministic merged PLA, which is optimal with regard to the partial order on lattice automata $\sqsubseteq$. The minimization algorithm works in the same way, and is also optimal for $\sqsubseteq$.

Note that all classical operations on lattice automata like merging, determinization, minimization, inclusion tests, etc are performed with almost the same complexity as their equivalent algorithms for finite automata, assuming the operations on the underlying lattice have a constant cost.

Moreover, if the underlying lattice has a widening operator, we can define a widening operator based on both :

- the k-bounded bisimulation,
- the widening operator ∇ of the underlying lattice.

In other words, we can see the lattice automata as a "functor", defining a new abstract lattice $(\mathrm{Reg}(\Lambda), \sqsubseteq, \nabla)$ from the underlying abstract lattice $(\Lambda, \sqsubseteq, \nabla)$.

A static analysis relying on the lattice $(\mathrm{Reg}(\Lambda), \sqsubseteq)$ (*cf.* Chapter 5) is thus modular, parametrized by $(\Lambda, \sqsubseteq)$, and we can easily change this abstract lattice without changing the analysis.

This is indeed one of the possibilities to improve the accuracy of a static analysis based on lattice automata. Other possibilities are:

- to increase the value of k: the higher k is, the more precise ∇_k is;
- to refine the partition $\pi : \Sigma \to \Lambda$;
- to replace the operator ρ_k by another widening operator on finite automate[7].

[7] We did not experiment this possibility.

This modularity is not the only advantage of lattice automata. The simplicity of their definition provides an effective representation of sets of words over an infinite alphabet. This is the reason why we tried to make the operations on lattice automata as general as possible, even if lattice automata were initially introduced to help the verification of communication protocols using integer parameters, which is the main application described in the following chapter.

Chapter 5

Applications : Verification of Symbolic Communicating Machines and Interprocedural Analysis

We illustrate in this chapter the application of the abstract domain $\mathrm{Reg}(\Lambda, \pi)$ defined in the previous chapter to the analysis of Symbolic Communicating Machines (SCMs). The SCM model is an extension of the CFSM model, with explicit variables. The values of those variables determine which transitions can be triggered. One can easily model communication protocols using integer variables in terms of SCMs, whereas CFSMs cannot describe this kind of protocols.

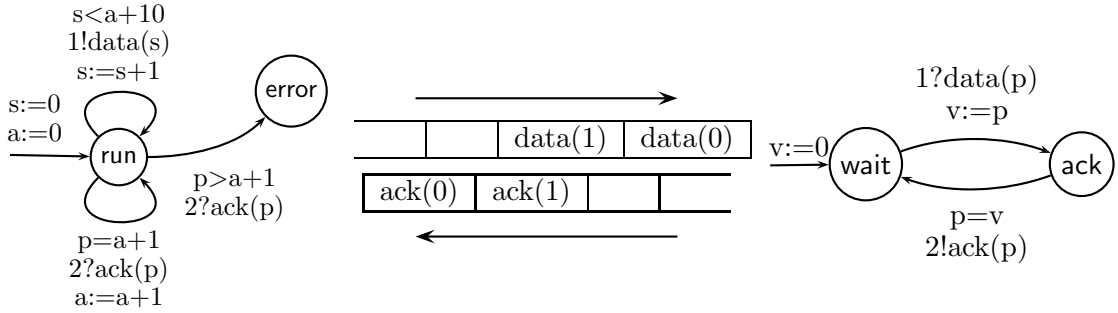
We define the SCM model and its straightforward abstract semantics, and give an example of analysis using the standard abstract semantics. This example highlights the need for a more sophisticated, non-standard abstract semantics. We define this new semantics and give some examples of analyses.

We also show how we can build an interprocedural analysis based on the abstract domain of lattice automata.

5.1 Verification of Symbolic Communicating Machines

5.1.1 Symbolic Communicating Machines

Symbolic Communicating Machines (SCMs) are Communicating Finite-State Machines extended with a finite set of variables V, the values of which can be sent into FIFO queues, *cf.* Figure 5.1. A transition is triggered when a condition ("guard") on the value of the variables is satisfied. In such a case, the machine can first send or receive values from a FIFO queue, then modify the values of variables, and last make the control jump to the destination location. This model is similar to other models like Extended Communicating Finite-State Machines [HSS+93] or Parametrized Communicating Extended



(a) Sender (b) Queues (c) Receiver

Figure 5.1: A simple sliding window protocol

Finite-State Machines [LRMS96].

Definition 5.1 *A SCM with N queues is defined by a tuple $\langle C, V, c_0, \Theta_0, P, \Delta \rangle$ where:*

- *C is a nonempty finite set of locations (control states).*
- *$V = \{v_1, \ldots, v_n\}$ is a finite set of variables. The domain of values of a variable v is denoted by $\mathcal{D}_v$, and the set of valuations of all variables in V by D_V.*
- *$c_0 \in C$ is the initial control state, and $\Theta_0 \subseteq \mathcal{D}_V$, a predicate on V, is the* initial condition.
- *$P = \{p_1, \ldots, p_l\}$ is a finite set of formal parameters that are used to send/receive values to/from FIFO queues. We assume that all queues use the same set of parameters $\mathcal{D}_P$.*
- *Δ is a finite set of* transitions. *A transition δ is either an* input $\langle c_1, G, i?\vec{p}, A, c_2 \rangle$ *or an* output $\langle c_1, G, i!\vec{p}, A, c_2 \rangle$ *where:*
  1. *c_1 and c_2 are resp. the origin and destination locations;*
  2. *$i \in [1..N]$ is a queue number;*
  3. *$\vec{p}$ is the vector of formal parameters, which holds the values sent or received to/from the queue i;*
  4. *$G(\vec{v}, \vec{p}) \subseteq \mathcal{D}_V \times \mathcal{D}_P$ is a predicate on the variables and the formal parameters (also called guard);*
  5. *A is an assignment of the form $\vec{v}' := A(\vec{v}, \vec{p})$, where $A : \mathcal{D}_V \times \mathcal{D}_P \to \mathcal{D}_V$, which defines the values of the variables after the transition.*

Compared to the definition of *Extended Communicating Finite-State Machines* [HSS+93], the values sent to queues may be of any type. Indeed, our model is closer to the Parametrized Communicating Extended Finite-State Machines model [LRMS96] (although there is no major difference between the three models).

In this model, the alphabet of "messages" is $\mathcal{D}_P$. We assume that this alphabet is not finite.

Standard Operational semantics. The standard operational semantics (SOS) of a SCM is defined in the same ways as the one of a CFSM, *i.e.* by an infinite transition system. The semantics of the SCM $\langle C, V, c_0, \Theta_0, P, \Delta\rangle$ is given as a labeled transition system $\langle Q, Q_0, \rightarrow\rangle$ where:

- $Q = C \times \mathcal{D}_V \times \left((\mathcal{D}_P)^*\right)^N$ is the set of states;
- $Q_0 = \{\langle c_0, \vec{v}, \varepsilon, \ldots, \varepsilon\rangle \mid \vec{v} \in \Theta_0\}$ is the set of initial states;
- $\rightarrow$ is defined by the two rules:

$$\frac{(c_1, G, i!\vec{p}, A, c_2) \in \Delta \qquad w_i' = w_i \cdot \vec{p} \qquad G(\vec{v}, \vec{p}) \qquad \vec{v'} = A(\vec{v}, \vec{p})}{\langle c_1, \vec{v}, w_1, \ldots, w_i, \ldots, w_N\rangle \rightarrow \langle c_2, \vec{v'}, w_1, \ldots, w_i', \ldots, w_N\rangle}$$

$$\frac{(c_1, G, i?\vec{p}, A, c_2) \in \Delta \qquad w_i = \vec{p}.w_i' \qquad G(\vec{v}, \vec{p}) \qquad \vec{v'} = A(\vec{v}, \vec{p})}{\langle c_1, \vec{v}, w_1, \ldots, w_i, \ldots, w_N\rangle \rightarrow \langle c_2, \vec{v'}, w_1, \ldots, w_i', \ldots, w_N\rangle}$$

A global state of a SCM is thus a tuple $\langle c, \vec{v}, w_1, \ldots, w_N\rangle \in C \times \mathcal{D}_V \times (\mathcal{D}_P)^* \times \cdots \times (\mathcal{D}_P)^*$ where c is a control state, $\vec{v}$ is the current value of the variables and w_i is a finite word on $\mathcal{D}_P$ representing the content of queue i.

The concrete collecting semantics of a SCM depicted on Fig. 5.3 is deduced from the operational semantics. It is a bit more complex because the model involves symbolic operations.

5.1.2 SCMs with a Single Queue: a Straightforward Approach

There are classical solutions to the abstraction of scalar variables. Consequently, the main issue of applying abstract interpretation techniques on this model is the abstraction of the queue contents. Whereas queue contents in the CFSM model are abstracted by regular languages, queue contents in the SCM model are abstracted using the abstract domain of lattice automata.

If there is a single queue, the concrete set of states associated to each control point $c \in C$ has the structure $\wp(\mathcal{D}_V \times (\mathcal{D}_P)^*)$: one associates to each control point the set of possible configurations for the variables of the communicating machine and its FIFO queue.

The concrete lattice $\wp(\mathcal{D}_V \times (\mathcal{D}_P)^*)$ can be abstracted with $\wp(\mathcal{D}_V) \times \mathcal{L}(\mathcal{D}_P)$ by projecting the two components. This allows to abstract $\wp(\mathcal{D}_V)$ using classical abstractions for variables of the environment, and to use lattice automata for abstracting the domain of queue contents $\mathcal{L}(\mathcal{D}_P)$.

For the sake of simplicity, and as an example, we will assume in the sequel that all variables and parameters are of rational type, and that sets of valuations are abstracted using the lattice of convex polyhedra $V^{(k)} = \mathsf{Pol}(\mathbb{Q}^k)$. We have the abstraction

$$\wp(\mathbb{Q}^n \times (\mathbb{Q}^p)^*) \;\leftrightarrows\; \wp(\mathbb{Q}^n) \times \mathcal{L}(\mathbb{Q}^p) \;\longleftarrow\; V^{(n)} \times \mathrm{Reg}(V^{(p)})$$

As the atoms of the lattice $V^{(p)} = \mathsf{Pol}(\mathbb{Q}^p)$ are precisely the elements of $\mathbb{Q}^p$, the concretization function of $\mathrm{Reg}(V^{(p)})$ is just the identity. The induced abstract semantics is given on Fig. 5.4.

Parameterized Domains	
Concrete domain	$C^{(n)} = \wp(\mathbb{Q}^n \times (\mathbb{Q}^p)^*)$
Abstract domain for variables	$V^{(n)} = \mathsf{Pol}(\mathbb{Q}^n)$
Abstract domain for one FIFO queue	$L = \mathrm{Reg}(V^{(p)})$
Abstract domain	$\Lambda^{(n)} = V^{(n)} \times L$

Figure 5.2: Semantic domains

Concrete collecting semantics

guard	$G(\vec{v},\vec{p})$	$[\![G]\!]$:	$\wp(\mathbb{Q}^n)$	$\to \wp(\mathbb{Q}^{n+p})$	by extension $[\![G]\!] : C^{(n)} \to C^{(n+p)}$
assignment	$A(\vec{v},\vec{p})$	$[\![A]\!]$:	$\wp(\mathbb{Q}^{n+p})$	$\to \wp(\mathbb{Q}^n)$	by extension $[\![A]\!] : C^{(n+p)} \to C^{(n)}$
output	$1!\vec{p}$	$[\![1!\vec{p}]\!]$:	$C^{(n+p)}$	$\to C^{(n+p)}$	
			X	$\mapsto \{(\vec{v},\vec{p},\ \omega\cdot\vec{p}) \mid (\vec{v},\vec{p},\ \omega) \in X\}$	
input	$1?\vec{p}$	$[\![1?\vec{p}]\!]$:	$C^{(n+p)}$	$\to C^{(n+p)}$	
			X	$\mapsto \{(\vec{v},\vec{p},\ \omega) \mid (\vec{v},\vec{p},\ \vec{p}\cdot\omega) \in X\}$	
transition	t	$[\![t]\!]$:	$C^{(n)}$	$\to C^{(n)}$	
			X	$\mapsto \begin{cases} [\![A]\!]\circ[\![G]\!] & \text{if } t=(G,--,A) \\ [\![A]\!]\circ[\![1!\vec{p}]\!]\circ[\![G]\!] & \text{if } t=(G,1!\vec{p},A) \\ [\![A]\!]\circ[\![1?\vec{p}]\!]\circ[\![G]\!] & \text{if } t=(G,1?\vec{p},A) \end{cases}$	

The semantics of a transition is defined as follows: first the queue variables $\vec{p}$ are introduced, and then constrained by the guard (semantics of guards). A push or pop operation is possibly performed. Last, the assignment updates the value of state variables. For push and pop, the value $\vec{p}$ is defined by a kind of unification between the guard and the queue content.

Figure 5.3: Concrete collecting semantics

Example 5.1 *We consider the protocol presented in the introduction (Figure 5.1). For the purpose of the analysis, we make the asynchronous product of the two automata. The result is a control structure with four states, labeled 00, 01, 10 and 11, with the following convention: the SCM is in the "xy" state if the sender is in the "x" state and the receiver is in the "y" one.*

The example was modeled by a system of equation and we solved the reachability analysis using a generic fix-point calculator. We used the polyhedra abstract lattice as implemented in the APRON library [APR]. No partitioning of the alphabet lattice was employed in this example. We obtained the result of Fig. 5.5.

The result of this analysis is disappointing: one cannot prove that the messages contained in the queues are indexed by integers that are lower than the variable s. This is not due to the queue abstraction, nor due to the variables abstraction, but to the coupling of the two abstractions. The following section gives a solution to this problem.

Standard abstract semantics		
guard	$G(\vec{v},\vec{p})$	$[\![G]\!]^\sharp : V^{(n)} \to V^{(n+p)}$ by extension $[\![G]\!]^\sharp : \Lambda^{(n)} \to \Lambda^{(n+p)}$
assignment	$A(\vec{v},\vec{p})$	$[\![A]\!]^\sharp : V^{(n+p)} \to V^{(n)}$ by extension $[\![A]\!]^\sharp : \Lambda^{(n+p)} \to \Lambda^{(n)}$
output	$1!\vec{p}$	$[\![1!\vec{p}]\!]^\sharp : V^{(n+p)} \times L \to V^{(n+p)} \times L$ $(Y,F) \mapsto (Y, F \cdot (\exists \vec{v} : Y))$
input	$1?\vec{p}$	$[\![1?\vec{p}]\!]^\sharp : V^{(n+p)} \times L \to V^{(n+p)} \times L$ $(Y,F) \mapsto \bigsqcup_\sigma (Y \sqcap \mathit{Embed} \circ \mathit{first}(F)(\sigma),\ (F/(\exists \vec{v} : Y))(\sigma))$ with $\mathit{Embed} : V^{(p)} \to V^{(n+p)}$ a canonical embedding function
transition	t	$[\![t]\!]^\sharp : \Lambda^{(n)} \to \Lambda^{(n)}$ $X \mapsto \begin{cases} [\![A]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, --, A) \\ [\![A]\!]^\sharp \circ [\![1!\vec{p}]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, 1!\vec{p}, A) \\ [\![A]\!]^\sharp \circ [\![1?\vec{p}]\!]^\sharp \circ [\![G]\!]^\sharp & \text{if } t = (G, 1?\vec{p}, A) \end{cases}$

Figure 5.4: Standard abstract semantics

5.1.3 SCMs with a Single Queue: Linking Messages and State Variables

The idea to improve on the previous abstraction is to use an augmented semantics to link the message variables contained in queues with the state variables of the machines.

The proposal of this section is to put into queues not only the queue content itself, but also (a subset of) the environment. This allows not only to establish relations between messages in queues and the current environment, but also to indirectly establish relations between the messages contained in different queues. For instance, the abstract value

$$\Big(s \in [8,9],\ \mathit{data}(\{s-p=3\}) \cdot \mathit{data}(\{s-p=2\}) \cdot \mathit{data}(\{s-p=1\})\Big)$$

represents the two concrete states:

$$(s=8, \mathit{data}(5) \cdot \mathit{data}(6) \cdot \mathit{data}(7))$$

and

$$(s=9, \mathit{data}(6) \cdot \mathit{data}(7) \cdot \mathit{data}(8))$$

This is to be compared with the standard abstraction of these two states:

$$\Big(s \in [8,9],\ \mathit{data}(p \in [5,6])) \cdot \mathit{data}(p \in [6,7]) \cdot \mathit{data}(p \in [7,8])\Big)$$

which represents $2^4 = 16$ concrete states.

Non-standard abstract semantics. The formalization of this technique requires slightly heavier notations. We assume that we put all the environment in the queue. The abstract lattice is then:

$$\Lambda^{(n)} = V^{(n)} \times L \quad \text{with} \quad L = \mathrm{Reg}(V^{(n+p)})$$

Control	Abstract Value
00	$[\|s \geq 0; a \geq 0\|]$ $(d, [\|p \geq 0\|])^*$ $(a, [\|p \geq 0\|])^*$
01	$[\|v \geq 0; s \geq 0; a \geq 0\|]$ $(d, [\|p \geq 0\|])^*$ $(a, [\|p \geq 0\|])^*$
10	$[\|s \geq 0; a - 2 \geq 0\|]$ $(d, [\|p \geq 0\|])^*$ $(a, [\|p \geq 0\|])^*$
11	$[\|v \geq 0; s \geq 0; a - 2 \geq 0\|]$ $(d, [\|p \geq 0\|])^*$ $(a, [\|p \geq 0\|])^*$

Figure 5.5: Analysis of the sliding window example with the standard approach

with the concretization function:

$$\gamma(Y, F) = \{(\vec{v}, \omega = \omega_0 \dots \omega_k) \in \mathbb{Q}^n \times (\mathbb{Q}^p)^* \mid \vec{v} \in Y \wedge (\vec{v}, \omega_0) \dots (\vec{v}, \omega_k) \in F\}$$

The partitioning function $\pi : \Sigma \to V^{(p)}$ used in PLA is implicitly extended to $\pi : \Sigma \to V^{(n+p)}$. $\Lambda^{(n)}$ may be seen as a reduced product of the two interacting components $V^{(n)}$ and $\mathrm{Reg}(V^{(n+p)})$.

Abstract operations. The abstract semantics of guards now involves both the abstract environment and the queue:

$$\begin{array}{rrcl} [\![G]\!]^r & : \quad V^{(n)} \times \mathrm{Reg}(V^{(n+p)}) & \to & V^{(n+p)} \times \mathrm{Reg}(V^{(n+p)}) \\ & (Y,\ F = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle) & \mapsto & ([\![G]\!]^\sharp(Y), F' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta' \rangle) \end{array}$$

with $\delta' = \{\ (q_1, \lambda', q_2) \mid (q_1, \lambda, q_2) \in \delta \wedge \lambda' = \lambda \sqcap (\exists \vec{p} : G)\}$.

The abstract semantics of assignments is more complex:

$$\begin{array}{rrcl} [\![A]\!]^r & : \quad V^{(n+p)} \times \mathrm{Reg}(V^{(n+p)}) & \to & V^{(n)} \times \mathrm{Reg}(V^{(n+p)}) \\ & (Y,\ F = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle) & \mapsto & (Y',\ F' = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta' \rangle) \end{array}$$

where $Y' = [\![A]\!]^\sharp(Y)$ and $\delta' = \{(q_1, \lambda', q_2) \mid (q_1, \lambda, q_2) \in \delta \wedge \lambda'$ defined as below $\}$. $\lambda'(\vec{v}, \vec{p_0})$ is obtained from $\lambda(\vec{v}, \vec{p_0})$ by the following operations:

1. We build $\phi(\vec{v}, \vec{p}, \vec{v'}, \vec{p_0}) = \lambda(\vec{v}, \vec{p_0}) \wedge \vec{v'} = A(\vec{v}, \vec{p}) \wedge Y(\vec{v}, \vec{p})$, where $\vec{v'}$ is the value of the state variables after the assignment, $\vec{v}$ and $\vec{p}$ are the current value of state and message variables, and $\vec{p_0}$ is the value of the message variable in the transition of the lattice automaton.

2. We then build $\phi'(\vec{v'}, \vec{p_0}) = \exists \vec{v} \exists \vec{p} : \phi$ by eliminating variables in the current environment.

Control	Abstract Value
00	$[\|0 \leq a \leq s \leq a+10\|]$ $(d, [\|0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\|])^*$ $(a, [\|0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\|])^*$
01	$[\|0 \leq a \leq s \leq a+10; 0 \leq v \leq s-1\|]$ $(d, [\|0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\|])^*$ $(a, [\|0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\|])^*$
10	$[\|0 \leq a \leq s-3 \leq a+7; 0 \leq v \leq s-1\|]$ $(d, [\|0 \leq a \leq s-1 \leq a+9; 0 \leq p \leq s-1\|])^*$ $(a, [\|0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\|])^*$
11	$[\|0 \leq a \leq s-3 \leq a+7; 0 \leq v \leq s-1\|]$ $(d, [\|0 \leq a \leq s-1 \leq a+10; 0 \leq p \leq s-1\|])^*$ $(a, [\|0 \leq a \leq s \leq a+10; 0 \leq p \leq s-1; 0 \leq v \leq s-1\|])^*$

Figure 5.6: Analysis of the sliding window example with the non-standard approach

3. Last we perform a renaming: $\lambda'(\vec{v}, \vec{p_0}) = \phi'(\vec{v'}, \vec{p_0})[\vec{v} \leftarrow \vec{v'}]$.

The application of the assignment to a NLA produces a NLA; in particular, as the partitioning function involves only the message parameters, and $[\![A]\!]^\sharp$ modifies only the state variables, the lattice automaton remains well-partitioned.

The semantics of message outputs and inputs are in some way simpler than with the previous abstract semantics:

$$[\![1!\vec{p}]\!]^r(Y, F) = (Y, F \cdot Y)$$

$$[\![1?\vec{p}]\!]^r(Y, F) = (Y \sqcap \mathit{first}(F), \widehat{F/Y})$$

The semantics of transitions remains similar to that of Fig. 5.4.

Example 5.2 *We performed a new analysis based on these non-standard semantics on the same example as before. A "widening threshold" operator was used instead of the standard widening on polyhedra [HPR97b]. We obtained the better result of Fig. 5.6.*

This analysis is quite accurate, since it shows that :

- $0 \leq a \leq s \leq a+10$, *i.e.* the sliding window property is satisfied,
- $0 \leq p \leq s-1$, *i.e.* the number of a message is strictly lesser than the number of the next message to be sent,
- $0 \leq v \leq s-1$, *i.e.* the number of the last received message is lesser than the number of of the next message to be sent.

Remark 5.1 (Efficiency of the approach.) *It should be noted that the abstraction of this section is much more expensive than the abstraction of Sect. 5.1.2, because guards and assignments do not apply only on the abstract value representing state variables,*

but also on the abstract values labeling the transitions of the automata. This suggests the use of a small partition π of the lattice Λ and the use of a small parameter k in the widening, in order to keep the lattice automata reasonably small.

In general, the most natural choice for the analysis of SCMs is to partition the alphabet according to the different kinds of messages exchanged (transmission, retransmission, acknowledgment, ...).

5.1.4 SCMs with Several Queues

As in Chapter 3, there are mainly two ways to analyze systems with several queues. One can follow:

- a non-relational, attribute-independent method, in which an abstract configuration is defined by a polyhedron representing the values of the state variables and N lattice automata, each representing a queue content,
- or a relational, attribute-dependent method, in which a single lattice automaton recognizes concatenated words $w = w_1 \sharp w_2 \sharp \ldots \sharp w_N$, where each subword w_i represents the content of the queue i and $\sharp$ is a special separating character [BG97].

The extension to lattice automata of these two variants discussed in Chapter 3 for finite automata is straightforward. In the previous subsections, the analyzed example, which has two FIFO queues, was implicitly analyzed with a non-relational approach. The implementation of lattice automata provides both representations (see Chapter 6).

5.1.5 Lossy Channels and Timers in the SCM Model

The opportunity of having variables in the SCM model makes the modelization of communication protocols with timers quite easy. Even when the messages do not carry any value, the method developed to verify SCMs helps the verification of communication protocols with timers.

We can model SCMs with lossy channels in the same way we modeled CFSM with lossy channel systems. The introduction of variables also adds the possibility to model channels that sometimes loose messages. For example, if we want to model a channel that looses at most one message out of ten, we simply add a variable $error$, which is increased each time a message is sent. When $error \geq 10$, the message that is about to be sent is discarded and $error$ is reset to 1.

Example. Here is another simplified version of the sliding window protocol. In this example, messages may be lost (one out of 4 data messages is lost, and one out of seven acknowledgment/error messages). The SCM is written in a language of description of SCMs, presented in Chapter 6.

Like the previous sliding window protocol, the sender can transmit some data, as long as there are no more than 10 messages sent but not acknowledged. The variable s is the number of new data to be sent, and a is the last acknowledgment received.

When an acknowledgment with a number greater than $a + 1$, or an error message, is received, it means there was a retransmission error. Then the variable s is reset to the last non-acknowledged message and the sender goes to a special state, meaning there was a transmission error. If there are two consecutive transmission errors, it goes to an error state and stops.

The receiver has variable v, which is the number of the next expected data. If a data message has a greater value, the message is discarded, and the receiver asks for a retransmission of the message. There is also a counter modeling a timer: when a wrong message is received, it is increased by one.

The variable er, in both automata, is the counter telling when a message is lost.

```
scm sliding_window_lossy :

nb_channels = 2 ;
parameters :
int data ;
int ack ;
int error ;

automaton sender :

int er = 1;
int s = 0 ;
int a = 0 ;

initial : 0

state 0 :
to 0 :  when s < a+10 and s == data and er <4, 0 ! data with s = s+1, er = er+1 ;
to 0 :  when s < a+10 and er == 4, with s = s+1, er = 1 ;
        /* the data message sent is lost */
to 0 :  when a == ack, 1 ? ack with a = a+1 ;
to 1 :  when a < ack, 1 ? ack with s = a ;
to 1 :  when true, 1 ? error with s = a ;

state 1 :
to 1 :  when s < a+10 and s == data and er <4, 0 ! data with s = s+1, er = er+1 ;
to 1 :  when s < a+10 and er == 4, with s = s+1, er = 1 ;
        /* the data message sent is lost */
to 0 :  when a == ack, 1 ? ack with a = a+1 ;
to 2 :  when a < ack, 1 ? ack ;
to 2 :  when true, 1 ? error ;
```

```
state 2 : /* error */


automaton receiver :

int v = 0 ;
int er = 1;
int t = 0 ;

initial : 0

state 0 :
to 0 :  when v != data and t <10, 0 ? data with t = t+1 ;
to 1 :  when v == data , 0 ? data ; /*transmision OK */
to 2 :  when t == 10 , with t=0;

state 1 : /* Ok */
to 0 :  when er <7 and ack == v, 1 ! ack with er = er+1 , v = v+1 ;
to 0 :  when er == 7 and ack == v , with er = 1 , v = v+1 ;


state 2 : /* transmission error */
to 0 :  when er <7 and err == v, 1 ! error with er = 1 ;
to 0 :  when er == 7 and err == v , with er = 1 ;
```

5.2 Application to Interprocedural Analysis

The analysis of symbolic communicating machines was the initial motivation for the study of lattice automata in Chapter 4. However, lattice automata can also be applied to precise interprocedural analysis, where one can use lattice automata for abstracting call-stacks. This allows both to simplify and to improve the abstraction proposed in [JS04]. One can also see this application as an extension to infinite state programs of [ES01a] which uses pushdown automata to model finite-state recursive programs and finite automata for representing (co-)reachable sets of configurations.

5.2.1 Program Model and Semantics

Program syntax. This program model is similar to the PDS model, presented in Section 2.2.1, extended with a set of variables.

A *procedure* $P_i = \langle \vec{fp}_i, \vec{fr}_i, \vec{loc}_i, G_i \rangle$ is defined by its input parameters $\vec{fp}_i$, its output parameters $\vec{fr}_i$, its local variables $\vec{loc}_i$, and G_i, its intraprocedural control flow graph (CFG). We also assume that input parameters are not modified during the execution of the procedure. This assumption is made solely for convenience, and involves no loss

of generality because it is always possible to copy input parameters to additional local variables.

The *intraprocedural CFG* G_i is a graph $G_i = \langle Ctrl_i, I_i \rangle$ where $Ctrl_i$ is the set of *control points* of P_i and $I_i : Ctrl_i \times Ctrl_i \to \mathsf{Inst}$ defines the instruction(s) lying between two control points between control points. G_i contains exactly one *start* point s_i and exactly one *exit* point e_i. We distinguish two kinds of instructions: (i) intraprocedural instructions, denoted as $\langle R \rangle$, that are specified as a relation $R \subseteq LEnv^2$ describing the transformation of the local environment; (ii) procedure calls $\langle \vec{y} := P_j(\vec{x}) \rangle$, where $\vec{x}$ and $\vec{y}$ are the vectors of actual input and output parameters. We require the graph G_i to be deterministic for procedure calls, *i.e.* if $I_i(c, c')$ is a call then there exists no c'' such that $I_i(c, c'')$ or $I_i(c'', c')$ is a call.

A *program* is defined by a set $(P_i)_{0 \le i \le p}$ of procedures. Since we specify the initial states of an analysis separately, there is no particular "main" procedure. The *interprocedural CFG* (G, I) of the program is the union of the intraprocedural CFG $(G_i)_{0 \le i \le p}$ of the different procedures, with the following modification: each procedure call edge linking a call-site point c and return-site point r, with $I_i(c, r) = \langle \vec{y} := P_j(\vec{x}) \rangle$ is replaced by

1. a *call-to-start* edge from c to the start point s_j of the called procedure, labeled by $\langle \texttt{call}\, \vec{y} := P_j(\vec{x}) \rangle$;
2. an *exit-to-return-site* edge from the exit point r_j of the called procedure to r, labeled by $\langle \texttt{ret}\, \vec{y} := P_j(\vec{x}) \rangle$;

Thus there are three kinds of instructions in interprocedural CFGs: intraprocedural instructions, procedure calls and procedure returns, see the factorial program beside Table 5.7. The functions call and ret record matching call and return-site nodes: $\mathsf{call}(r) = c$ and $\mathsf{ret}(c) = r$. We assume that a start point has no incoming edges except call-to-start edges.

$\vec{fp}_i$	: Formal input parameters of P_i
$\vec{fr}_i$	: Formal output parameters of P_i
$\vec{loc}_i$	: Local variables of procedure P_i
$LVar_i$	: $\vec{fp}_i \cup \vec{fr}_i \cup \vec{loc}_i$
$G_i = \langle Ctrl_i, I_i \rangle$	: Flow graph of P_i
$\mathsf{s}_i, e_i \in Ctrl_i$	: Entry and exit points of P_i
$G = \langle Ctrl, I \rangle$	: Flow graph of the program

Figure 5.7: Syntactic domains.

The syntactic domains are summarized on Figure 5.7.

Operational Semantics. The semantic domains are summarized in Table 5.9.

The standard operational semantics (SOS) is given by a transition system $(State, \to)$. *States* are stacks of the form $\Gamma = \langle c_0, \epsilon_0 \rangle \cdot \ldots \cdot \langle c_n, \epsilon_n \rangle$ of activation records (*i.e.* pairs of

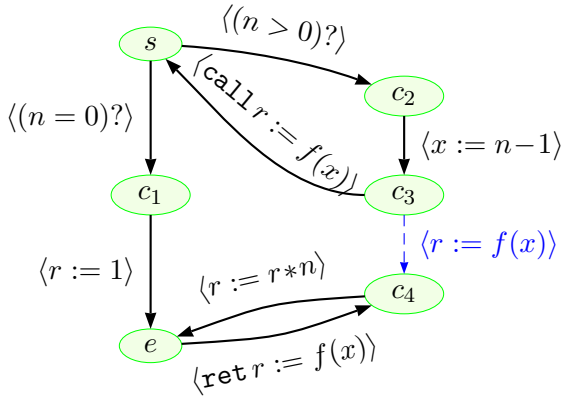


Figure 5.8: CFG of the Factorial Program

v	$\in \mathit{Value}$	: values
ϵ_i	$\in \mathit{LEnv}_i = \mathit{LVar}_i \to \mathit{Value}$	: local environments for P_i
ϵ	$\in \mathit{LEnv} = \bigcup_i \mathit{LEnv}_i$	: local environments
$\langle c, \epsilon \rangle$	$\in \mathit{Act} = \mathit{Ctrl} \times \mathit{LEnv}$	: activation record
Γ	$\in \mathit{State} = \mathit{Act}^+$	: stacks/program states

Figure 5.9: Semantic domains

a control point c_i and an environment ϵ_i). $\langle c_n, \epsilon_n \rangle$ is the *current activation record* or *top* of Γ; the *tail* of Γ is Γ without its top, *i.e.* $\langle c_0, \epsilon_0 \rangle \cdot \ldots \cdot \langle c_{n-1}, \epsilon_{n-1} \rangle$. Environments map variables to values; their update is written $\epsilon[x \mapsto v]$. The *transition relation* $\to \subseteq \mathit{State} \times \mathit{State}$ is defined (in SOS-style) by the rules in Figure 5.10. As long as a variable is not initialized, it holds nondeterministically any value in its domain, *cf.* rule (Call). As usual, $\to^*$ denotes the reflexive-transitive closure of $\to$.

$$\frac{I(c, c') = \langle R \rangle \quad R(\epsilon, \epsilon')}{\Gamma \cdot \langle c, \epsilon \rangle \to \Gamma \cdot \langle c', \epsilon' \rangle} \text{(Intra)}$$

$$\frac{I(c, \mathtt{s}_j) = \langle \mathtt{call}\ \vec{y} := P_j(\vec{x}) \rangle \quad \forall k : \epsilon_j(\vec{fp}_j^{(k)}) = \epsilon(\vec{x}^{(k)})}{\Gamma \cdot \langle c, \epsilon \rangle \to \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle \mathtt{s}_j, \epsilon_j \rangle} \text{(Call)}$$

$$\frac{I(e_j, c) = \langle \mathtt{ret}\ \vec{y} := P_j(\vec{x}) \rangle \quad \epsilon' = \epsilon[\vec{y}^{(k)} \mapsto \epsilon_j(\vec{fr}_j^{(k)})]}{\Gamma \cdot \langle \mathsf{call}(c), \epsilon \rangle \cdot \langle e_j, \epsilon_j \rangle \to \Gamma \cdot \langle c, \epsilon' \rangle} \text{(Return)}$$

Figure 5.10: SOS rules defining $\to$

Standard Collecting Semantics. The forward collecting semantics describes the set of reachable states of a program. It is the natural choice for expressing and verifying invariance properties and is derived from the operational semantics by collecting the states belonging to executions of the program. We define the function $\mathit{Reach} : \wp(\mathit{State}) \to \wp(\mathit{State})$ computing the states reachable from a set of *initial states* X_0 as:

$$\mathit{Reach}(X_0) \triangleq \{q \mid \exists q_0 \in X_0\ ,\ q_0 \to^* q\} = \mathrm{lfp}(F[X_0])$$

where lfp is the least fix-point operator, $F[X_0](X) \triangleq X_0 \cup post(X)$ and $post(X) = \{q' \mid \exists q \in X : q \rightarrow q'\}$ is the *forward transfer function*. Figure 5.11 gives the decomposition of Post according to the transitions of the CFG, *i.e.* $\text{Post}(X) = \bigcup_{(c,c') \in Ctrl \times Ctrl} \text{Post}(c \xrightarrow{I(c,c')} c')(X)$.

$$
\begin{array}{rl}
\text{Post}(c \xrightarrow{\langle R \rangle} c')(X) & = \left\{ \Gamma \cdot \langle c', \epsilon' \rangle \;\middle|\; \Gamma \cdot \langle c, \epsilon \rangle \in X \wedge R(\epsilon, \epsilon') \right\} \\
\text{Post}(c \xrightarrow{\langle \texttt{call}\, \vec{y} := P_j(\vec{x}) \rangle} \mathsf{s}_j)(X) & = \left\{ \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle \mathsf{s}_j, \epsilon_j \rangle \middle| \Gamma \cdot \langle c, \epsilon \rangle \in X \wedge \epsilon_j(\vec{fp}_j^{(k)}) = \epsilon(\vec{x}^{(k)}) \right\} \\
\text{Post}(e_j \xrightarrow{\langle \texttt{ret}\, \vec{y} := P_j(\vec{x}) \rangle} c)(X) & = \left\{ \Gamma \cdot \langle c, \epsilon' \rangle \;\middle|\; \begin{array}{l} \Gamma \cdot \langle \mathsf{call}(c), \epsilon \rangle \cdot \langle e_j, \epsilon_j \rangle \in X \\ \epsilon' = \epsilon \left[\vec{y}^{(k)} \mapsto \epsilon_j(\vec{fr}_j^{(k)}) \right] \end{array} \right\}
\end{array}
$$

Figure 5.11: Forward transfer function Post

Since $F[X_0]$ is monotonic and continuous, Kleene's fix-point theorem allows us to compute the forward collecting semantics by iterated application of $F[X_0]$ starting from $\emptyset$:

$$Reach(X_0) \;=\; \text{lfp}(F[X_0]) \;=\; \textstyle\bigcup_{n \geq 0} \; (F[X_0])^n(\emptyset)$$

The collecting semantics can also be considered backward, yielding the set of states X from which a given set of *final states* X_0 is reachable. In this case we call X the set of *co-reachable states* of X_0. We get the following definitions:

$$
\begin{array}{rcl}
\text{Pre}(X) & = & \{q \mid \exists q' \in X : q \rightarrow q'\} \\
\text{Coreach}(X_0) & = & \text{lfp}(G[X_0]) \;\; \text{with} \;\; G[X_0](X) = X_0 \cup \text{Pre}(X)
\end{array}
$$

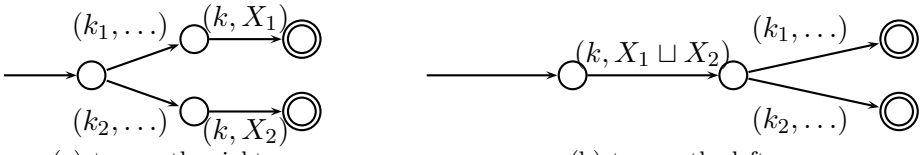
Properties of the stacks in the standard semantics. The assumption that formal input parameters are read-only variables induces strong properties on stacks which are the basis of our stack abstractions. A necessary condition for $q = \Gamma \cdot \langle c, \epsilon \rangle$ to lead to $q' = \Gamma \cdot \langle c, \epsilon \rangle \cdot \langle c', \epsilon' \rangle$ (where c is a call site to $P_{\mathsf{proc}(c')}$) is that the values of actual input parameters in ϵ have to match those of the formal input parameters in ϵ'. This is formalized by the following definition.

Definition 5.2 (valid calling activation record) *$\langle c, \epsilon \rangle$ is a* valid calling activation record *(or* valid*) for $\langle c', \epsilon' \rangle$ if*
(i) c is a call site for procedure P_j: $\exists j : c' = \mathsf{s}_j \;\wedge\; I(c, \mathsf{s}_j) = \langle \texttt{call}\, \vec{y} := P_j(\vec{x}) \rangle$;
(ii) actual and formal parameters are equal: $\forall k : \epsilon(\vec{x}^{(k)}) = \epsilon'(\vec{fp}_j^{(k)})$.

Extending Definition 5.2, we call a stack $\langle c_0, \epsilon_0 \rangle \ldots \langle c_n, \epsilon_n \rangle$ *consistent* if $\langle c_i, \epsilon_i \rangle$ is valid for $\langle c_{i+1}, \epsilon_{i+1} \rangle$ $(\forall 0 \leq i < n)$.

We define the function $\phi : \wp((K \times LEnv)^+) \rightarrow \wp((K \times LEnv)^+)$ as the function which extracts from its argument the subset of consistent stacks.

This notion of consistency guides the concretization of our abstractions of stacks, based on lattice automata.



(a) top on the right (b) top on the left

Figure 5.12: Top of the stack (control point c) on the right versus on the left

5.2.2 Abstracting Call-Stacks with Lattice Automata

According to Figure 5.9, the concrete lattice of properties of the imperative programs considered is $\wp((K \times LEnv)^+)$. We assume that we have an atomic abstract lattice $LEnv^\sharp$ for environments, with a concretization function $\gamma_e : LEnv^\sharp \to \wp(LEnv)$. By noticing the isomorphism $\wp(K \times LEnv) \simeq K \to \wp(LEnv)$, we extend this abstraction to activation records:

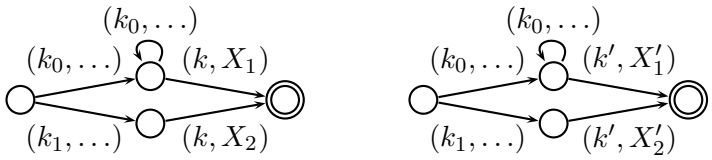
$$\begin{array}{rccc} \gamma_e : & (K \to LEnv^\sharp) & \longrightarrow & (K \to \wp(LEnv)) \simeq \wp(K \times LEnv) \\ & e^\sharp & \longmapsto & \gamma_e \circ e^\sharp \end{array} \tag{5.1}$$

We now instantiate Eqn (5.1) by taking $S = (K \times LEnv)^+$ and $A = K \to LEnv^\sharp$. We thus abstract sets of call-stacks with $\mathrm{Reg}((K \to LEnv^\sharp), \pi)$. The canonical partitioning function π we will use is the function separating abstract environments according to their attached control point: $\pi : K \to LEnv^\sharp$ defined by $\pi(k) = \top_{LEnv^\sharp}$.

A first important point is to decide whether the top of the stack is on the right (initial states) or on the left (final states). This choice is not neutral, as determinism in automata is oriented to the left (and normalization of automata induces a loss of precision). If we want to perform polyvariant analyses, where several abstract environments may be associated to the same control point depending on the calling context, as in Figure 5.12.(a), we should put the top of the stack on the right. Indeed, if we reverse the deterministic automaton of Figure 5.12.(a) and if we normalize it, Figure 5.12.(b), then X_1 and X_2 will be merged, which is not the desired behavior. As a consequence, we adopt the convention where the top of the stack is on the right, which is moreover in accordance to the notation used in the concrete and abstracts semantics of programs described in Section 5.2.1.

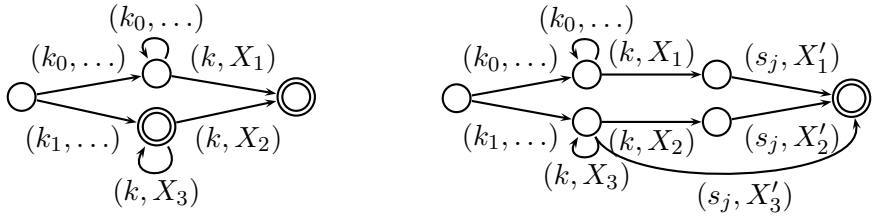
A second point is to exploit the fact that call-stacks should be consistent (*cf.* Section 5.2.1). Thus the concretization function γ_{seq} of $\mathrm{Reg}((K \to LEnv^\sharp), \pi)$ will be combined with a function ϕ filtering out non-consistent stack, resulting in the concretization function $\gamma_{cs} = \phi \circ \gamma_{seq}$. This will be exploited in abstract transfer functions, which should approximate correctly $\alpha_{cs} \circ F \circ \gamma_{cs}$ where F is a concrete transfer function.

Abstract transfer functions. We need now to define the abstract postcondition on $\mathrm{Reg}((K \to LEnv^\sharp), \pi)$. A procedure call (corresponding to a push) involves parameter passing between the current activation record and the newly added activation record. A procedure return (corresponding to a pop) also involves a combination of the first two activation records on the top of stack.



(a) Before the instruction (b) After the intruction

Figure 5.13: Abstract transfer function for an intraprocedural instruction $\tau = k \xrightarrow{\langle R \rangle} k'$, with $X'_i = post^\sharp_f(\tau)(X_i)$



(a) Before the call (b) After the call

Figure 5.14: Abstract transfer function for a procedure call $\tau = k \xrightarrow{\langle \texttt{call}\ \vec{y} := P_j(\vec{x}) \rangle} s_j$, where $X'_i = post^\sharp_f(\tau)(X_i)$

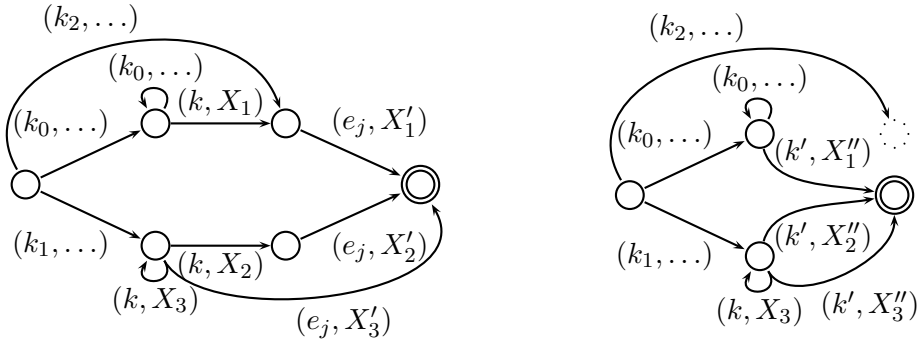
We have detailed on Figure 5.11 the concrete forward transfer function. We describe here how we defined and implement their abstract counterpart on lattice automata. For all type of instructions, we consider the PLA $\mathcal{A}(k)$ representing the reachable stacks associated to the control point k, in which all the transitions reaching a final states are labeled by abstract activation records of the form (k, X), with $X \in A$ is an abstract environment.[1]

The case of an intraprocedural instruction $\tau = k \xrightarrow{\langle R \rangle} k'$ is the easiest and is illustrated on Figure 5.13. In the PLA $\mathcal{A}(k)$, no final state can have an outgoing transition, because k cannot be a call-site. This means that the transitions reaching a final state represent necessarily the top of the (abstract) stack. In this situation, the image of $\mathcal{A}$ can be obtained by replacing transition $l \xrightarrow{(k,X)} l'$ where l' is a final state by transitions $l \xrightarrow{(k', post^\sharp_f(\tau)(X))} l'$. This may result in a non-deterministic PLA, for instance if $k' = k_0$ on Figure 5.13.(b).

The case of a procedure-call instruction $\tau = k \xrightarrow{\langle \texttt{call}\ \vec{y} := P_j(\vec{x}) \rangle} s_j$ is also easy and is illustrated on Figure 5.14. For each transition of $\mathcal{A}(k)$ of the form $l \xrightarrow{(k,X)} l'$, where l' is a final state, we add a transition $l' \xrightarrow{(s_j, post^\sharp_f(\tau)(X_i))} l_f$, with l_f the new and unique final state.

We consider now the case of a procedure-return instruction $\tau = e_j \xrightarrow{\langle \texttt{ret}\ \vec{y} := P_j(\vec{x}) \rangle} k'$,

[1] as we never merge final states with non-final ones in widening, this invariant will be preserved.



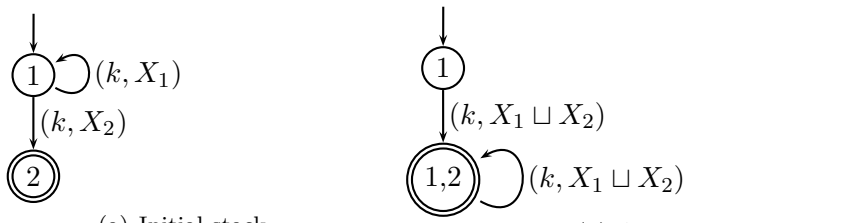
(a) Before the return

(b) After the return

Figure 5.15: Abstract transfer function for a procedure return $\tau = e_j \xrightarrow{\langle \texttt{ret}\ \vec{y} := P_j(\vec{x}) \rangle} k'$, with $k = \mathsf{call}(k')$ and $X''_i = post^\sharp_f(\tau)(X_i, X'_i)$

illustrated on Figure 5.15. We note $k = \mathsf{call}(k')$ the call-site corresponding to the return-site k'. Notice that in the PLA $\mathcal{A}(e_j)$, no final state can have an outgoing transition, because e_j cannot be a call-site. The operation proceeds as follows: for any sequence of transitions in $\mathcal{A}(e_j)$ of the form $l \xrightarrow{(k,X)} l' \xrightarrow{(e_j,X')} l''$, where l'' is a final state, we add a transition $l \xrightarrow{(k', post^\sharp_f(\tau)(X,X'))} l_f$, with l_f the new and unique final state.

Normalization and widening. In Chapter 4, we defined a normal form for lattice automata. However, this normalization induces a loss of information, which happens to be sometimes too strong in the context of interprocedural analysis. We will give experimental evidence of this problem in the following section. Figure 5.16 illustrates the problem with a recursive procedure, and normalization at the call-site. Figure 5.16.(a) gives the possible structure of the postcondition of an intraprocedural instruction leading to call-site. After determinization, which is the first step of normalization, we obtain the automaton of Figure 5.16.(b), where the two different environments X_1, X_2 have been merged.



(a) Initial stack

(b) After determinization

Figure 5.16: Loss of information at the call site of a recursive call

As a consequence, we will mostly manipulate non-normalized lattice automata, and we will rely solely on widening for termination. We illustrate this issue with two examples of interprocedural analysis, and introduce the *polyvariant analysis*.

5.2.3 An Interprocedural Analyzer

5.2.3.1 Implementation in an interprocedural analyzer

We implemented the abstract postcondition on call-stacks represented with lattice automata defined in Chapters 4, and we have connected this abstract lattice to the academic INTERPROC interprocedural analyzer [int]. We also implemented the abstract precondition which is based on similar principles.

INTERPROC is an interprocedural analyzer for a small imperative language with recursive procedure calls [int]. It infers invariants on the numerical variables of analyzed program, by forward analysis. It can also infer necessary conditions to reach a given control point (by backward analysis), and it can combine both kind of analysis iteratively. It exploits the APRON library for abstracting numerical variables [APR] and the FIXPOINT library as a generic fix-point equation solver [Fix].

INTERPROC has been recently improved by adding Boolean and enumerated variables to numerical variables (integer, real, or floating-point), and by replacing the APRON numerical abstract domain by the BDD+APRON domain. This last domain abstracts environments belonging to $LEnv = V \to \mathbb{B} \cup \mathbb{R} \simeq \mathbb{B}^m \cup \mathbb{R}^n$ with the lattice $LEnv^\sharp = \mathbb{B}^m \to A[n]$, where $A[n]$ is one of the abstract lattice for $\wp(\mathbb{R}^n)$ provided by the APRON library. In this lattice, finite-state variables are thus not abstracted, whereas numerical variables are abstracted as before. Elements of $LEnv^\sharp$ are actually represented with MTBDDs (Multi-Terminal Binary Decision Diagrams), hence the name BDD+APRON.

All the following experiments have been performed by using convex polyhedra as the abstract domain for numerical variables.

5.2.3.2 Two toy examples

Polyvariant analysis. Our stack abstraction allows us to implement polyvariant analysis very easily. In a polyvariant analysis, a function may be analyzed separately according to calling-context. A very simple example where a polyvariant analysis is useful is given on Figure 5.17.(a). On Figure 5.17.(b), the result of a standard interprocedural analysis with functional abstraction is given in the comments. The analysis has not captured properly the effect of the `abs` function, because of the approximation induced by the use of convex polyhedra: `abs` is called both with negative and positive arguments, so the analysis just infers $r \geq -n \wedge r \geq n \wedge n + r \geq 0$ at the exit point of `abs`. At the first call-site, we obtain the exact invariant thanks to the relation $n + r \geq 0$, but at the second call-site in the loop, we do not obtain $y = x$. On Figure 5.17.(c), we use the stack abstraction, using a bounded backward bisimulation of depth 1 for normalization, and we give in comments the join of the tops of the reachable stacks. Because of the join performed on the top of reachable stacks, the invariants of function `abs` are not improved, however the invariants of function `main` are exact, because internally the full abstract stacks are used to compute the effect of function calls. Figure 5.17.(d) depicts the reachable stacks associated to the exit point of procedure `abs`.

```
proc abs(n:int) returns
        (r:int)
begin
  if n>=0 then
    r = n;
  else
    r = -n;
  endif;
end

var x:int,y:int;
begin
  x = -3;
  y = abs(x);
  x = 3;
  while x>=0 do
    y = abs(x);
    x = x-1;
  done;
end
```

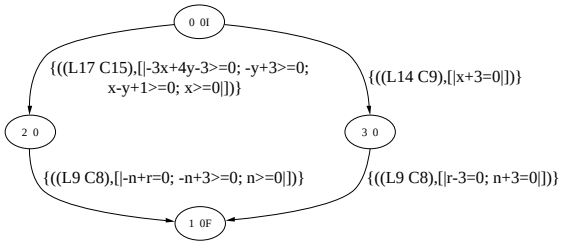
(a) Program

```
proc abs (n : int) returns (r : int)
begin
  /* (L4 C5) [|-n+3>=0; n+3>=0|] */
  if n >= 0 then
     /* (L5 C14) [|-n+3>=0; n>=0|] */
     r = n; /* (L6 C10) [|-n+r=0; -n+3>=0; n>=0|] */
  else
    /* (L7 C6) [|-n-1>=0; n+3>=0|] */
    r = - (n); /* (L8 C11) [|n+r=0; -n-1>=0; n+3>=0|] */
  endif; /* (L9 C8) [|-n+r>=0; -r+3>=0; n+r>=0|] */
end

var x : int, y : int;
begin
  /* (L13 C5) top */
  x = - (3); /* (L14 C9) [|x+3=0|] */
  y = abs(x); /* (L15 C13) [|y-3=0; x+3=0|] */
  x = 3; /* (L16 C8) [|-3x+4y-3>=0; -y+3>=0; x+y+1>=0|] */
  while x >= 0 do
    /* (L17 C15) [|-3x+4y-3>=0; -y+3>=0; x>=0|] */
    y = abs(x); /* (L18 C15) [|-x+y>=0; -y+3>=0; x+y>=0|] */
    x = x - 1; /* (L19 C12) [|-x+y-1>=0; -y+3>=0; x+y+1>=0|] */
  done; /* (L20 C7) [|-x-1>=0; -y+3>=0; x+y+1>=0|] */
end
```

(b) Forward analysis with functional abstraction

```
proc abs (n : int) returns (r : int) var ;
begin
  /* (L4 C5) [|-n+3>=0; n+3>=0|] */
  if n >= 0 then
     /* (L5 C14) [|-n+3>=0; n>=0|] */
     r = n; /* (L6 C10) [|-n+r=0; -n+3>=0; n>=0|] */
  else
    /* (L7 C6) [|n+3=0|] */
    r = - (n); /* (L8 C11) [|r-3=0; n+3=0|] */
  endif; /* (L9 C8) [|-n+r>=0; -r+3>=0; n+r>=0|] */
end

var x : int, y : int
begin
  /* (L13 C5) top */
  x = - (3); /* (L14 C9) [|x+3=0|] */
  y = abs(x); /* (L15 C13) [|y-3=0; x+3=0|] */
  x = 3; /* (L16 C8) [|-3x+4y-3>=0; -y+3>=0; x-y+1>=0|] */
  while x >= 0 do
    /* (L17 C15) [|-3x+4y-3>=0; -y+3>=0; x-y+1>=0; x>=0|] */
    y = abs(x); /* (L18 C15) [|-x+y=0; -x+3>=0; x>=0|] */
    x = x - 1; /* (L19 C12) [|-x+y-1=0; -x+2>=0; x+1>=0|] */
  done; /* (L20 C7) [|y=0; x+1=0|] */
end
```

(c) Forward analysis with stack abstraction, bounded backward bisimulation of depth 1

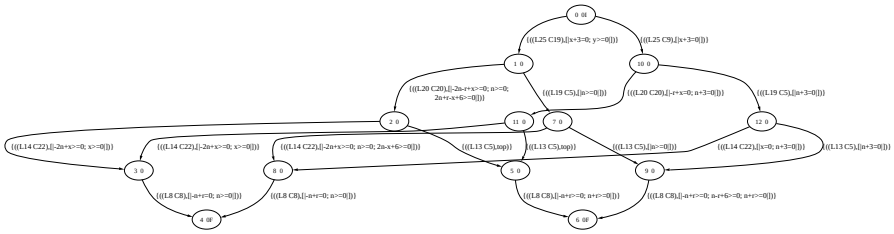


(d) Abstract stack at exit point of function `abs` with stack abstraction, bounded backward bisimulation of depth 1

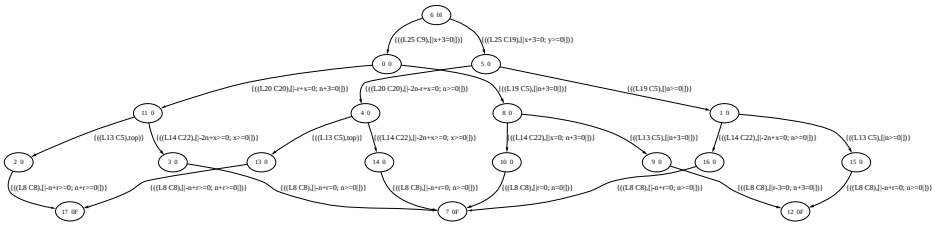
Figure 5.17: Example of polyvariant (forward) analysis

```
proc abs(n:int) returns
        (r:int)
begin
  if n>=0 then
    r = n;
  else
    r = -n;
  endif;
end

proc f(n:int) returns (r:int)
var x : int;
begin
  x = abs(n); x = x+n; r = abs(x);
end

proc g(n:int) returns (r:int)
var x : int;
begin
  r = f(n); r = x-r; r = f(r);
end

var x:int,y:int;
begin
  x = -3; y = g(x); x = g(y);
end
```

(a) Program example



(b) Abstract stack at exit point of `abs`, with bounded backward bisimulation of depth 1



(b) Abstract stack at exit point of `abs`, with bounded backward bisimulation of depth 2

Figure 5.18: Stack abstraction and program inlining

Polyvariant analysis and inlining. An alternative to polyvariant analysis is to first inline the initial program, and then to perform an ordinary interprocedural analysis. However, the inlining approach has three drawbacks:

1. As it relies on a combination of program transformation and analysis, it does not allow us to formalize easily the approximation performed in the analysis, whereas our approach is in some way an integrated one;

2. It works only for non-recursive programs, whereas our approach works for recursive programs, as illustrated by Section 5.2.3.3. In such cases, the analysis performs implicitly a bounded inlining of the program.

3. It is not equivalent to the stack abstraction, thanks to the use of bounded bisimulation, as illustrated by the example of Figure 5.18.(a), where the main procedure calls twice `g`, calling itself twice `f`, which last calls twice `abs`. Figure 5.18.(b) depicts the abstract stack at exit point of `abs`, where several non-final states have several incoming edges, which means that the automaton does not correspond to a full tree. To obtain a full tree, we need to use a bounded backward bisimulation of depth 2, see Figure 5.18.(c), where only final states have several incoming edges, which does not impact the precision).

5.2.3.3 The MacCarthy91 function

We then experimented the analyzer to the well-known MacCarthy 91 function, depicted on Figure 5.19. The exact semantics of `MC` is:

$$\forall n \in \mathbb{Z} : \texttt{MC}(n) = \begin{cases} 91 & \texttt{if } n \leq 100 \\ n - 10 & \texttt{if } n \geq 101 \end{cases}$$

Applying a standard analysis (based on the functional abstraction $\alpha_e \circ \alpha_f$ of Section 5.2.1), using polyhedra [Min01], yields to the invariants given in the comments on Figure 5.20, that do not capture the exact semantics, but its best approximation with convex polyhedra. On the other hand, if we manually partition the input parameter with the condition `n>=100`, we obtain the program of Figure 5.21, on which the functional abstraction delivers exact invariants.

We will show that the gain of stack abstraction over the functional abstraction in term of precision, in the case where we do not partition the values of variable `n`. In Tabs.5.1–5.3, "+det" means that determinization is applied, "-det" that it is not, "-backward n" means that we use bounded backward bisimulation of depth "n" (default 0), and "forward n" means that we use bounded backward bisimulation of depth "n" (default 0).

Experiment 1. We first try to infer that if we assume `x<=100` at beginning of the main function, we have at the end `y=91`. The results are depicted on Tab. 5.1. The functional analysis did not succeed, neither the stack-based analysis with the option "-backward 0", which is essentially equivalent to functional analysis, as normalized

Type of analysis and normalization	$y = 91$ proved ?
functional	no
stack -det -backward 0	no
stack -det -backward 1	no
stack -det -backward 2	yes
stack -det -backward 3	yes
stack +det -backward 3	no
stack -det -forward 3	no

Table 5.1: Experiment 1

Type of analysis and normalization	property on x inferred ?
functional	$\top$
stack -det -backward 1	$x \geq 102$
stack +det -backward 1	$\top$
stack -det -forward 1	$\top$
stack -det -forward 2	$\top$
stack -det -forward 2	$\top$

Table 5.2: Experiment 2: backward analysis

Type of analysis and normalization	property on x inferred ?
functional	$\top$
stack -det -backward 1	$x \geq 102$
stack +det -backward 1	$x \geq 102$
stack -det -forward 1	$\top$
stack -det -forward 2	$\top$
stack +det -forward 2	$\top$

Table 5.3: Experiment 2: backward analysis after forward analysis

automata have at most 2 states, a final and a non-final one, see Figure 5.22.(a). We need a depth of 2 or more to prove the property. This means that in this case the stack abstraction allows to recover the loss of information due to the use of convex polyhedra. Notice that in this case, the lattice automaton attached to the return point of function `MC` is rather complex, see Figure 5.22.(c). Now, if we apply determinization, the analysis fails, even with a depth of 3. The analysis also fails when using forward bisimulation.

Experiment 2. We now assume any input in the function `main` when calling the function `MC`, and we try to infer a necessary condition on `x` for having `y>91` at the end of `main`. This can be done by considering the function `main` besides, in which the instruction `fail` defines a final control point for backward analysis. The results are depicted on Tab. 5.2. Here, a depth of 1 only in the bounded backward bisimulation allows to infer the exact condition. If we perform a forward analysis to select reachable states before performing a backward analysis on the restricted state-space, we obtained the results of Tab. 5.3. In this case we succeed to infer the right condition bounded backward bisimulation of depth 1, even with determinization, thanks to the gain in precision due to the intersection of the two analyses. Figure 5.23 depicts the abstract call-stack at the end of function `MC`, after the forward analysis followed by the backward analysis. One can observe that with a depth of 1, the call-stack is still quite complex, whereas with a depth 2 the intersection is much more precise.

```
var x:int, y:int,b:bool;
begin
  y = MC(x);
  if y>91 then
    fail;
  endif;
end
```

As for experiment 1, the use of bounded forward bisimulation does not allow to obtain the expected results.

5.2.3.4 Discussion

We did not experiment non-toy programs, as we need for this a connection to an existing programming language, which is not done yet. These preliminary experiments allow already to draw some conclusions.

First of all, the experiments on the MacCarthy 91 function showed than the normalization of lattice automata in this context should be based on a backward bisimulation relation, and that it is preferable to work with non-deterministic automata. Second, these experiments illustrate the fact that lattice automata allow to a implement polyvariant analysis that performs a partial inlining and works with recursive programs; the degree of partial inlining is moreover controlled by the depth of the bounded bisimulation relation used for normalization. They also show that the use of abstract stacks allows to recover partially the loss of information due to the abstraction of the environments. We last point out the fact that backward analysis is easily implemented, as well as its intersection with forward analysis during the fix-point computations.

These experiments also showed that applying lattice automata to interprocedural analysis of non-toy programs requires a deeper study. For instance, the depth of the

bounded bisimulation is maybe too rough to tune finely the size of lattice automata. The implementation could also be more clever, in particular in combined forward/backward analysis where the present implementation perform a full lattice automata intersection at each control point.

5.2.4 Related Work

As explained in the introduction, one can distinguish two main approaches to interprocedural static analysis, namely the *functional* and the *operational*. The functional approach of [CC77b] has been used for instance to analyze the access to arrays in interprocedural Fortran programs [CI96], using the abstract domain of convex polyhedra [CH78]. [RHS95] can be seen as an algorithmic implementation of [KS92] using graph reachability techniques, which can be used with finite lattices and distributive data flow functions. This technique can be applied to all *bit-vector* analyses and the Bebop tool [BR00] is based on it. An extension [SRH96] allows to tackle some finite-height infinite lattices, like (linear) constant propagation.

In the operational approach, [Bou90] considers more complex Pascal programs with reference parameter passing, which introduces aliasing on the stack (*i.e.* several variables may refer to the same location in the stack), and nested procedure definitions. Unsurprisingly, the devised solution is quite complex. It has been implemented using the interval domain for integers [CC77a]. Stacks are collapsed more severely than in our model. The proposal of [EK99], implemented in Moped [ES01b] and applied to concurrent programs in [BET03], relies on the result that the set of reachable stacks of a pushdown automata is a regular language, that can be represented by finite-state automata. The analyzed program is converted to a pushdown automaton, and is thus restricted to programs manipulating finite-state variables/properties, or requires the finite abstraction of data/properties prior the analysis. A recent extension allows the use of some infinite finite-height lattices [RSJ03] and represents a very interesting mix of the two approaches: pushdown automata are here extended by associating transformers to transition rules. This allows to encode the control part of the program and properties belonging to a finite lattice in the pushdown automata, whereas properties belonging to a finite-height lattice can be handled by the transformers attached to the transitions. This approach can be applied to infinite-height lattice by patching it with the use of widening (as done in the experiments of [GR07]), but the elegant theoretical results are lost. It can also be used for backward analysis and its intersection with forward analysis [LKRT07]. However, in this case the intersection is performed after the two analyses has been performed independently, which can lead to less precise results than performing the second analysis on the restricted state-space computed by the first analysis. A distinguish feature of our approach is to ability to tune the abstraction of stacks, whereas it is fixed in the analysis of weighted pushdown systems, and defined by control flow graph of the encoded programs, and possibly by the finite-state variables that are encoded in the control rather than with the weights attached to transitions.

5.3 Conclusion

We presented in this chapter the model of Symbolic Communicating Machines (SCMs). The main interest of this model is to easily describe real communication protocols, which often manipulate boolean and integer variables.

We illustrated the use of lattice automata for the verification of SCMs, and we showed the need for a non-standard semantics to couple the abstraction of the state variables of the machines with the contents of the FIFO queues. To our knowledge, this is the first verification technique able to deal with message carrying any kind of values without manual transformation of the model. We also explore the applicability of lattice automata to interprocedural analysis and compare this solution to related work.

We discussed the application of lattice automata to interprocedural analysis. With the abstraction of the stack contents provided by lattice automata, we defined a polyvariant analysis, where several abstract environments may be associated to the same control point depending on the calling context. We had to adapt the operations like "push" and "pop" so they become context-sensitive. We experimented this kind of analysis and showed that we had to define ρ_k operator by a backward bisimulation instead of a forward one. We also need to do as few normalization operations as possible. We experimented our analyzer on some toy examples, justifying the choice we maid, and we plan to apply it on more significant examples.

Future work also includes a deeper study of the experimental relevance of lattice automata to the analysis of SCM. The challenge we would like to take up is the verification of the SSCOP communication protocol, which is a sliding window protocol from which our running example is extracted. Previous verification attempts that we are aware of are either based on enumerated state-space exploration techniques [BFG$^+$00b], or on the partial use of theorem proving [Rus03]. It would be interesting to study the application of lattice automata to shape analysis, in the spirit of [BHRV06].

```
proc MC(n:int) returns
       (r:int)
var t1:int, t2:int;
begin
  if n>100 then
     r = n-10;
  else
     t1 = n + 11;
     t2 = MC(t1);
     r = MC(t2);
  endif;
end

var x:int, y:int,b:bool;
begin
  y = MC(x);
end
```

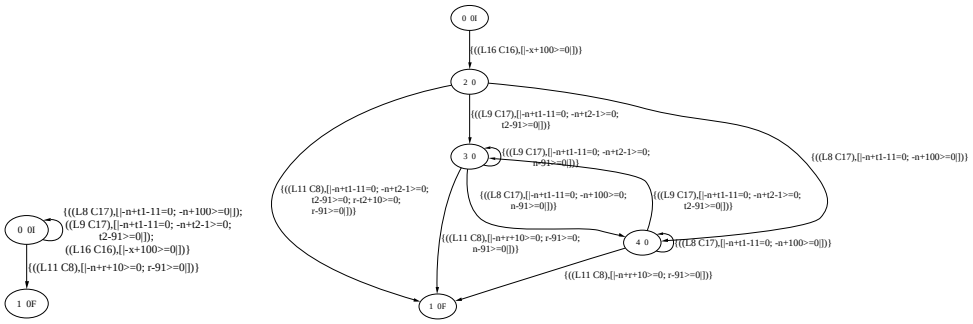
Figure 5.19: MacCarthy91 function

```
proc MC (n : int) returns
        (r : int)
var t1 : int, t2 : int;
begin
  /* (L4 C5) top */
  if n > 100 then
     /* (L5 C15) [|n-101>=0|] */
     r = n - 10; /* (L6 C14)
                    [|-n+r+10=0; n-101>=0|] */
  else
    /* (L7 C6) [|-n+100>=0|] */
    t1 = n + 11; /* (L8 C17)
                    [|-n+t1-11=0; -n+100>=0|] */
    t2 = MC(t1); /* (L9 C17)
                    [|-n+t1-11=0; -n+100>=0;
                      -n+t2-1>=0; t2-91>=0|] */
    r = MC(t2); /* (L10 C16)
                  [|-n+t1-11=0; -n+100>=0;
                    -n+t2-1>=0; t2-91>=0;
                    r-t2+10>=0; r-91>=0|] */
  endif; /* (L11 C8) [|-n+r+10>=0; r-91>=0|] */
end

var x : int, y : int, b : bool
begin
  /* (L15 C5) top */
  y = MC(x); /* (L16 C12) [|-x+y+10>=0; y-91>=0|] */
end
```

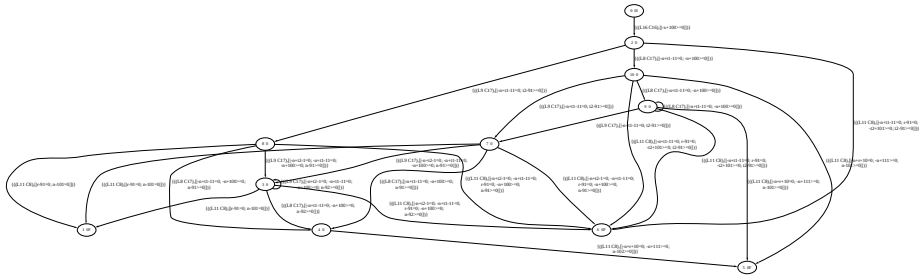
Figure 5.20: Reachability analysis

```
proc MC (b : bool, n : int) returns (r : int)
var t1 : int, t2 : int, b1 : bool;
begin
  /* (L6 C5) { [|n-101>=0|] IF b,
              [|-n+100>=0|] IF not b } */
  if b != n > 100 then
    /* (L7 C24) bottom */
    fail; /* (L7 C30) bottom */
  endif; /* (L7 C37) { [|n-101>=0|] IF b,
                      [|-n+100>=0|] IF not b } */
  if b then
     /* (L8 C11) { [|n-101>=0|] IF b,
                   bottom OTHERWISE } */
     r = n - 10; /* (L9 C14) { [|-n+r+10=0; n-101>=0|] IF b,
                               bottom OTHERWISE } */
  else
    /* (L10 C6) { [|-n+100>=0|] IF not b,
                  bottom OTHERWISE } */
    t1 = n + 11; /* (L11 C17) { [|-n+t1-11=0; -n+100>=0|] IF not b,
                                 bottom OTHERWISE } */
    b1 = t1 > 100; /* (L12 C17)
                      { [|-n+t1-11=0; -n+100>=0; n-90>=0|] IF not b and b1,
                        [|-n+t1-11=0; -n+89>=0|] IF not b and not b1,
                        bottom OTHERWISE } */
    t2 = MC(b1, t1); /* (L13 C20)
                        { [|-n+t2-1=0; -n+t1-11=0; -n+100>=0; n-90>=0|] IF
                          not b and b1,
                          [|-n+t1-11=0; t2-91=0; -n+89>=0|] IF not b and not b1,
                          bottom OTHERWISE } */
    b1 = t2 > 100; /* (L14 C17)
                      { [|t2-101=0; t1-111=0; n-100=0|] IF not b and b1,
                        [|-n+t1-11=0; -n+t2-1>=0; -t2+100>=0; t2-91>=0|] IF
                        not b and not b1,
                        bottom OTHERWISE } */
    r = MC(b1, t2); /* (L15 C19)
                       { [|t2-101=0; t1-111=0; r-91=0; n-100=0|] IF not b and b1,
                         [|-n+t1-11=0; r-91=0; -n+t2-1>=0; -t2+100>=0; t2-91>=0|] IF
                         not b and not b1,
                         bottom OTHERWISE } */
  endif; /* (L16 C8)
            { [|-n+r+10=0; n-101>=0|] IF b,
              [|t2-101=0; t1-111=0; r-91=0; n-100=0|] IF not b and b1,
              [|-n+t1-11=0; r-91=0; -n+t2-1>=0; -t2+100>=0; t2-91>=0|] IF
              not b and not b1 } */
end

var x : int, y : int, b : bool
begin
  /* (L20 C5) top */
  b = x > 100; /* (L21 C12) { [|x-101>=0|] IF b,
                             [|-x+100>=0|] IF not b } */
  y = MC(b, x); /* (L22 C14) { [|-x+y+10=0; x-101>=0|] IF b,
                               [|y-91=0; -x+100>=0|] IF not b } */
end
```

Figure 5.21: Reachability analysis on a partitioned version

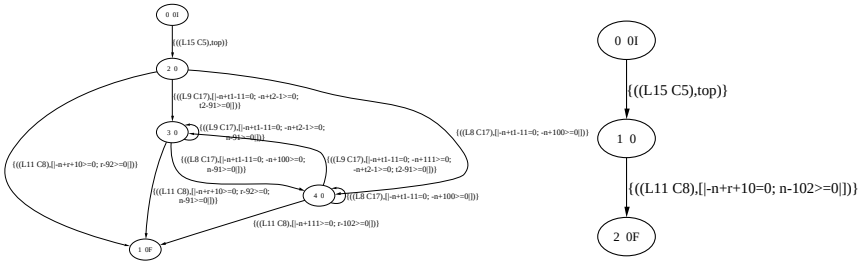


(a) "-backward 0" (b) "-backward 1"



(c) "-backward 2"

Figure 5.22: Experiment 1: abstract call-stack at the end of function `MC`, with option "stack -det"



(a) "-backward 1" (b) "-backward 2"

Figure 5.23: Experiment 2: abstract call-stack at the end of function `MC`, with option "stack -det" and combined forward and backward analysis

Chapter 6

Implementation

This chapter describes how we implemented the lattice automata and the algorithms presented in Chapters 3, 4 and 5. Technically, the implementation of lattice automata is an Objective CAML library. The CFSMs and SCMs analyzer is a tool also written in Objective CAML [OCa], using:

- the lattice automata library,
- the APRON (numerical abstract domains) libraries [APR],
- a generic fix-point calculator [Fix].

The APRON library aims at providing several libraries implementing numerical abstract domains like intervals [CC77a], convex polyhedra [CH78] or octagons [Min01]. Those libraries have a common interface, so we can easily switch between abstract domains.

We try to make the implementation as modular as possible. So the APRON libraries are not specifically needed by the SCM analyzer, any abstract domain with an Objective CAML interface can replace APRON. The experiments, however, were made using only the APRON libraries.

In the following, we briefly describe the modules of the lattice automata library and the input language of the analyzer. We also discuss the difference between the theory and the implementation, and give some examples of analyses.

We also implemented our interprocedural analysis by improving an existing tool, Interproc, developed by Bertrand Jeannet. The results of the experimentation are given in Section 5.2.3.

6.1 The Lattice Automata Library

The lattice automata library implements the operations on lattice automata presented in Chapter 4.

6.1.1 Description of the Library

The lattice automata library uses some modules of the CamlLib library [CLi], in particular the representation of graphs. The lattice automata library is composed of the following modules:

- **PLattice:** module for partitioned lattice;
- **Regexp:** module for regular expressions;
- **LAutomaton:** module for lattice automata, implementing the operations described in the previous chapters;
- **StackQueueN:** this module provides the functions needed for the definition of the semantics of SCMs;
- **StackQueueNRel:** same module as the previous one, except that queues (or stacks) are represented in a single QDD-like automaton, instead of several automata.

Remark 6.1 *Note that we only implemented the partitioned lattice automata, not the general lattice automata.*

The two last modules have a common interface. So one can easily switch between a relational analysis and a non-relational one (cf. Chapter 3). They provide functions operating on stacks and queues.

We now give some details of the implementation, highlighting the differences between the implementation and the theory.

6.1.2 The Representation of the Partitioned Lattice

Representation of PLAs. A partitioned lattice automaton $\mathcal{A} = \langle \Lambda, \pi, Q, Q_0, Q_f, \delta \rangle$, as defined in Chapter 4, is represented by a graph with additional informations like the sets of initial states and the set of final states. The partition $\pi : \Sigma \to \Lambda$ is given by an explicit map, associating each letter σ of the alphabet to the element $\pi(\sigma)$. The module **PLattice** implements the operations on the partitioned lattice. So the user must specify, when building an automaton, the whole partition even when some elements are not employed to label transitions of the automaton.

Remark 6.2 *When the lattice Λ is actually of the form $\Lambda = \Sigma \times \Lambda'$, and the partition given by the function $\pi(\sigma) = (\sigma, \top')$ where $\top'$ is the greatest element of the lattice Λ', the implementation is simply a map $\Sigma \to \Lambda'$, without the letters. This case happens when we verify SCMs with a partition only based on the type of messages sent or received.*

Manager. The module **PLattice** is technically a functor, the input of which is a module implementing the alphabet Σ. The standard operations on the lattice Λ are provided by an external manager. The type of a partitioned lattice is an open type, parametrized by the type of the elements of Λ. This manager also requests an element `sep`, the value of which does not matter ; this element is employed to label transitions separating states of different queues, for QDD-like automata.

Functions. The functions provided by this module are:

- operations on the partitioned lattice: meet, join, widening, etc ;
- some functions to change the partition.

The second kind of functions take a function redefining the partition as an argument, and applies it to the map representing the partition.

6.1.3 Lattice Automata Modules

Representation of lattice automata. As mentioned previously, a lattice automaton is represented by a graph with additional informations. Our implementation of lattice automata, like QDDs, represent all the queue contents in a single automaton. The automata presented in Chapter 4 are indeed the particular case where there is a single queue. The *dimension* $\dim(\mathcal{A})$ of an automaton is the number of queues.

We identify some sub-automata, each sub-automaton representing the contents of the queue i (between 0 and $\dim(\mathcal{A}) - 1$). So each vertex of the graph carries a queue number, and each edge of the graph is either:

- a set of classical transitions, between two vertices of the same queue i;
- a separation transition, between a vertex of the queue i and a vertex of the queue $i + 1$.

Edges are labeled by a map $\Sigma \to \Lambda$ defining the transitions of a merged PLA; for example, if there is an edge labeled my the map $a \mapsto [1, 2], b \mapsto [-3, -1]$, it means that we have two transitions, one labeled by $[1, 2]$ and belonging to the equivalence class a, the other labeled by $[-3, -1]$ and belonging to the equivalence class b.

Each queue has its own set of initial and final states, stored in two arrays of length $\dim(\mathcal{A})$. Other important informations attached to the graph are the partition and the lattice[1].

Functions of the module LAutomaton. This module implements the algorithms described in Chapter 4, like inclusion tests, determinization and normalization. It also implements the language operations like the union and the intersection of two automata. There are also functions for the concatenation of a letter and the left and right derivation. Other noticeable functions are:

[1]There are also some useful informations for the algorithms, like boolean values being true if the lattice automaton is deterministic or minimal.

- the operator ρ_k, described in Chapter 3;
- the widening operator, described in Chapter 4.

6.1.4 Interfaces for Queues and Stacks

The modules **StackQueueN** and **StackQueueNRel** are the modules called by the analyzer. They may be seen as an interface layer, because they essentially call functions of the modules **LAutomaton** and **LAutomatonRep**. Their role is:

- to rename some functions like the left-derivation or the right-concatenation. The new names (pop/push) are more adapted to the queues or stack terminology, so the user does not need to remember what language operation corresponds to a push.
- to give the choice between a relational analysis and a non relational one. Since we have a QDD-like representation of lattice automata, we need a new layer to implement a non-relational lattice, where N queue-contents are represented by an array of N lattice automata of dimension 1.

6.2 The SCM Analyzer

The analyzer performs a reachability analysis of a SCM (forward analysis). The analyzer reads the description of a SCM, written in a small input language. It then generates a system of equations and calls a fix-point calculator[Fix]. At the end of the computation, we obtain an over-approximation of the reachability set.

6.2.1 Input Language

The input language is a textual description of a SCM. Here is the example of the sliding window protocol.

```
scm sliding_window :

nb_channels = 2 ;
parameters :
int data ;
int ack ;

automaton sender :

int s = 0 ;
int a = 0 ;

initial : 0
```

```
state 0 :
to 0 :  when s < a+10 and s == data , 0 ! data with s = s+1 ;
to 0 :  when ack == a+1 , 1 ?ack with a = a+1 ;
to 1 :  when ack > a+1 , 1 ?ack ;

state 1 :



automaton receiver :

real v = 0 ;


initial : 0

state 0 :
to 1 :  when true , 0 ? data with v = data ;

state 1 :
to 0 :  when ack == v , 1 ! ack ;
```

Grammar. We remind that <> denotes a non-terminal symbol, [ ] denotes something optional and the bold words are the key words of the language.

<scm> ::= **scm** <ident> :
<channels>
[<lossy>]
<parameters>
<channels>
{<automaton>}$^+$

<channels> ::= **nb_channels** = <integer> ;
<lossy> ::= **lossy** : <integer> {,<integer>}*
<parameters> ::= **parameters** : {<declaration>}*

Each SCM has a name, a number of channels (0,1,..., nb_channels-1), some parameters (being the messages sent in the channels) and at least one automaton. One may also indicate which channels are lossy. Other channels are considered as perfect FIFO queues.

<automaton> ::= **automaton** <ident> :
{<declaration>}*
initial : <integer> {, <integer> }*
{<state>}$^+$
<declaration> ::= <type_def> <ident> [= <iexpr>];
<type_def> ::= **int** | **real**

Each automaton has a name, at least one initial state and may have some local variables. The local variables may have an initial value, given by an expression.

<state> ::= **state** <integer> : {<transition>}*
<transition> ::= **to** <integer> : **when** <bexpr>, <action> <assign_rule> ;
<assign_rule> ::= [**with** <assignment>{, <assignment>}*]
<action> ::= [<integer> { ! | ? } <ident>]
<assignment> ::= <ident> = <iexpr>

Each state is identified by an integer. A transition leads to a state, identified by its integer identifier, when the guard is satisfied. It sends or receives a message, or does nothing, and may modify the values of variables.

<bexpr> ::= **true** | **false** | (<bexpr>)
| <bexpr> **and** <bexpr> | <bexpr> **or** <bexpr> | **not** <bexpr>
| <iexpr> <const> <iexpr>

<const> ::= == | != | < | <= | > | >=

<iexpr> ::= <float> | <integer> | <ident> | (<iexpr>)
| <iexpr> + <iexpr> | <iexpr> - <iexpr>
| <iexpr> * <iexpr> | <iexpr> / <iexpr>
| <iexpr> % <iexpr> | - <iexpr>
| **sqrt** <iexpr>

The syntax of the expressions is rather classical. % is the modulo and **sqrt** is the square root.

Observers. The user can specify an observer of the property, using the same input language. This observer can read the values of variables of the observed SCM, but shall not modify them. It has its own set of variables.

6.2.2 Description of the Software

The analyzer is written in Objective CAML and his currently available via INRIA GForge [GFo]. This Gforge project provides the source files of the analyzer and the source files of all required libraries (called CamlLib, Fixpoint and Apron). See also the APRON project page (`http://apron.cri.ensmp.fr/`) for more details.

Use of the APRON library. In the current version of the analyzer, one can only choose one of the APRON abstract domains as the lattice abstracting the values of the variables. We are thus limited to integer and floating-point variables. The architecture

of the software, however, only refers to this library in the file `scm_manager.ml`. In other words, if we want to change the abstract domain in the future, we just have to rewrite this file.

Options of the analyzer. The analyzer is called by the command `./analyzer`. The main options offered to the user are:

- **-v** : displays the version of the analyzer.
- **-threshold** : the analyzer use the widening with thresholds instead of the standard widening.
- **-apron** *name* : uses one of APRON abstract domains. Without this option, the analyzer performs an analysis with finite automata instead of lattice automata. The available lattices are "polka" (convex polyhedra), "box" (intervals) and "oct" (octagons).
- **-forward** k : sets the parameter k for the widening. Default is 0.
- **-backward** k : sets the parameter k for the widening (backward bisimulation). Default is 0.
- **-obs** *filename* : reads an observer, which is a SCM described in the file *filename*. The result is the reachability analysis of the synchronous product

The SCM is read from the standard input, and the result of the analysis is printed onto the standard output.

6.2.3 Examples of analyses

We illustrate on some examples the effect of the options of the analyzer.

widening with thresholds. When the option **-threshold** is active, the analyzer employs the widening with thresholds [HPR97a], as defined in the APRON library, instead of the standard widening. We explain the difference between the two widenings operating on the lattice of convex polyhedra. A convex polyhedron is defined by a conjunction of linear constraints $\bigwedge_{1\leq i\leq m} a_{i,1}x_1 + \ldots a_{i,n}x_n \leq c_i$. If we have two convex polyhedra $P_1 \sqsubseteq P_2$, then all the constraints satisfied by P_2 are also satisfied by P_1, but the contrary is not true.

Let $P_1 = y \geq 0 \wedge x \leq y \wedge x \leq 4$ and $P_2 = y \geq 0 \wedge x \leq y \wedge x \leq 6$ be two convex polyhedra such that $P_1 \sqsubseteq P_2$.

- The standard widening consists in removing from P_1 the constraints not satisfied by P_2. In this example, $P_1 \nabla P_2 = y \geq 0 \wedge x \leq y$.

- The widening with thresholds is parametrized by a finite set of linear constraints Thresholds $= \{a_{i,1}x_1 + \ldots a_{i,n}x_n \leq c_i \,|\, 1 \leq i \leq p\}$. We remove the constraints of P_1 not satisfied by P_2, but also add the constraints of Thresholds satisfied by both polyhedra. In this example, if Thresholds $= \{x \leq 5, 2y + 7 \geq x\}$, then $P_1 \nabla P_2 = y \geq 0 \wedge x \leq y \wedge 2y + 7 \geq x$.

The widening with thresholds is thus more precise than the standard widening, but we have to guess what are the useful constraints. This set depends on the SCM we analyzed. In general, the useful constraints are the postcondition of the transitions of the SCM. When the option **-threshold** is active, the analyzer generate a set of constraints based on the postconditions of all transitions of the SCM. The user may also specify other constraints that will be add to this set.

Example 6.1 *Figure 6.1 is taken from the analysis of the sliding window protocol, presented in Chapter 5. In this case, the transition* $p == s, !p, s = s + 1$ *generates two thresholds* $p \leq s - 1$ *and* $p \geq s - 1$*. Since the first constraint is satisfied by both polyhedra, the widening with thresholds adds it to the resulting polyhedron.*

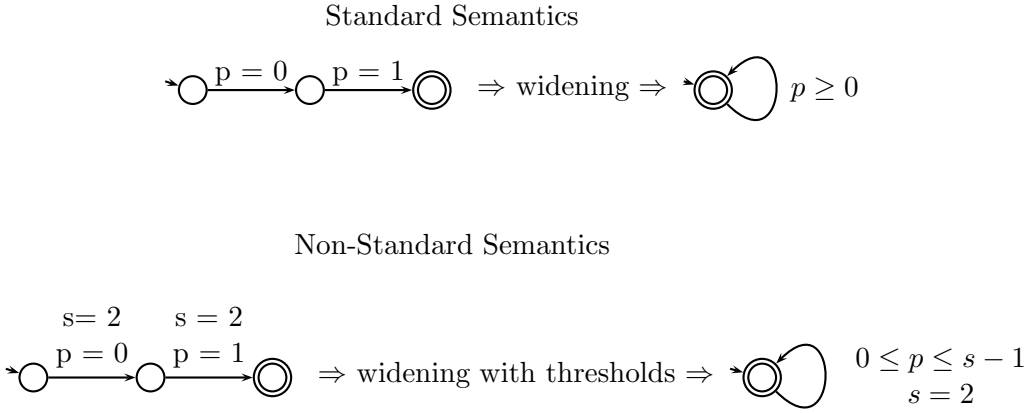
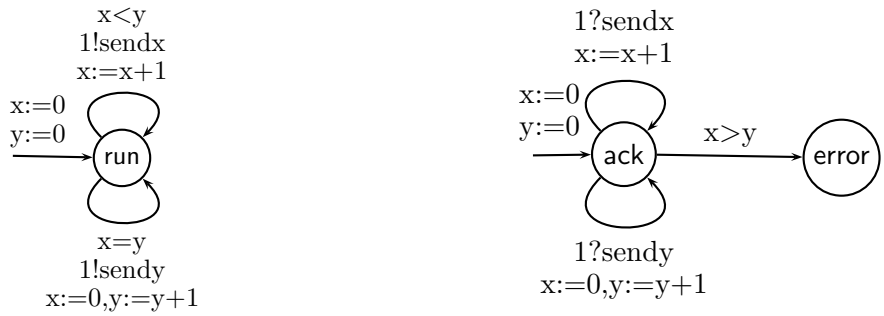


Figure 6.1: Non-standard semantics: the widening with thresholds

Choice of the lattice. The choice of the abstract lattice determines both the precision of the analysis and its complexity. The current architecture of the analyzer offers three possible lattices as an abstraction of the values of the variables $x_1, \ldots x_n$:

- the lattice of intervals ("boxes") abstracts the values of the variables by n interval constraints : $\bigwedge 1 \leq i \leq n a_i \leq x_i \leq b_i$;
- the lattice of convex polyhedra employs a conjunction of linear constraints: $\bigwedge_{1 \leq i \leq m} a_{i,1}x_1 + \ldots a_{i,n}x_n \leq c_i$;
- the lattice of octagons is a kind of convex polyhedra, where the coefficients in the linear constraints are restricted to the set $\{-1, 1\}$.



(a) Sender (c) Receiver

Figure 6.2: A toy example of SCM

This choice is again a choice between a relational lattice (the convex polyhedra) which is precise but costly, and a non-relational lattice (the boxes) which is less costly but also far less precise. The lattice of octagons is a compromise between the two.

Example 6.2 *Figure 6.2 presents a simple SCM with a guard $x < y$. We prove, using the convex polyhedra, that the state "error" cannot be reached, whereas the same analysis with intervals is inconclusive.*

Conclusion

This chapter describes the current implementation of the lattice automata library and of the SCM analyzer. While the first one is "complete", in other words it implements all the functions we need, the SCM analyzer is still under development.

We plan to improve, in the following months, the interface of our software. We hope it will eventually take standard formats, like IF [BFG+00a] or SDL [Tur93] files as input, and give a graphical representation of the output.

Another improvement would be the automatic discovery of abstract counter-examples and the refinement of the partition when the result of our analysis is that the "bad configurations" may be reachable.

We also plan to perform backward analyses as well as forward analyses. The operations needed for a backward analysis are already programmed in the lattice automata library.

Finally, we want to base our analysis on abstract domains more sophisticated than the numerical abstract domains of APRON. In particular, we will combine a numerical abstract domain and a boolean abstract domain in order to verify communication protocols with integer and boolean variables, like the bounded retransmission protocol (BRP).

We hope that, with these improvements, the analyzer will become a useful tool for the analysis of communication protocols and other FIFO systems.

Conclusion

When we started this work, our first goal was to employ the abstract interpretation framework to verify FIFO channel systems. The main scientific issues were:

- to define some non-numerical abstract domains adapted to data structures like queues or stacks. Even if there are a great number of abstract domains, few domains abstract a sequence of values, and none were conceived specifically for the analysis of FIFO channel systems.
- to show the advantages of using the abstract interpretation framework for the verification of FIFO channel systems, compared to acceleration techniques.

Contributions

These two scientific concerns lead us to the definition of an abstract domain based on regular languages, presented in Chapter 3. Our main contribution was to define a suitable widening operator ∇_k, based on the equivalence relation $\approx_k$ (bisimulation of depth k). Its principle is to merge the states of the automaton belonging to the same equivalence class of $\approx_k$.

From a practical point of view, a static analyzer using ∇_k has some advantages compared to other softwares conceived for the verification of Communicating Finite State Machines (CFSMs):

- ∇_k guarantees that we obtain an over-approximation of the least fix-point of the function Post; the computation usually terminates in only a few steps.
- The over-approximation is a regular language. In other approximate algorithms, the approximations are less precise: they just keep the number of messages, or their type, loosing the FIFO ordering. Our analysis is arguably precise enough to obtain the exact reachability set (when possible) for the communication protocols we studied.
- The parameter k determines the precision of the analysis; the higher k is, the more precise analysis we get. We also have the choice between a relational analysis, which is precise but may be expensive and a non-relational one, which is less precise but also less expensive.

Since the results, presented in [LGJJ06], were satisfying, we extended this method to the verification of Symbolic Communicating Machines (SCMs). The SCM model is basically the CFSM model extended with local variables and messages with parameters. Since the alphabet of messages is infinite in the SCM model, we represent the queues contents by lattice automata instead of finite automata.

The definition and the main properties of lattice automata are discussed in Chapter 4. We remind here the main results of our work on lattice automata:

- We defined the notion of lattice automata, and discussed the need for a partition of the underlying lattice Λ. With this partition, we can adapt some algorithms operating on finite automata to lattice automata.
- We defined and proved the algorithms for the inclusion test, the determinization and the minimization. Those algorithms lead to a robust notion of normalization of lattice automata, which enjoys nice properties.
- We studied the possibility to refine the partition of Λ. We demonstrated that this refinement improves the quality of the abstractions.
- We defined a widening operator, based on both the widening operator of the lattice Λ and the the widening operator of Chapter 3.
- We also defined the language operations we needed for the analysis of systems with stacks and queues.

With those algorithms, we analyzed SCMs in the same way we analyzed CFSMs (*cf.* Chapter 5). This task was however a bit more complex than the analysis of CFSMs, since we must have a symbolic representation of the possible values of the variables as well as a symbolic representation of the queue contents. We first defined a straightforward semantics, in which an abstract value is a couple (Y, F), where Y is an abstraction of the values of the variables and F an abstraction of the queues contents. The queues-contents are lattice automata, the underlying lattice of which being an abstraction of the values of the parameters.

In our experiments, we adopted some well-known abstract domains, like the convex polyhedra, as an abstraction of the values of both the variables and the parameters. While experimenting this kind of analysis, we concluded that a good analysis must keep the link between the values of the parameters and the values of the variables.

We thus introduced a more sophisticated, non-standard semantics that keeps this link. The abstract values are still couples of the form (Y, F), but this time, the base lattice of the lattice automata representing the queues contents is an abstraction of the values of both the parameters and the variables. We obtain better results with this new semantics, presented in [LGJ07].

We also studied the adaptation of this method to interprocedural analysis. In this context, we have a set of procedures, each one having a set of local variables, that can call each others. Our idea was to abstract all possible values of stack contents by a lattice automaton.

We had to do some minor adjustments in order to obtain a useful analysis of stack contents: the bisimulation is computed backward, the "push" and "pop" operations are defined to make them context-sensitive , and we tried to avoid normalization as often as possible. With these adjustments, our experiments show that this interprocedural analysis is quite precise.

We have implemented the algorithms on lattice automata in an Objective CAML library. This library provides all the functions needed for the analysis of systems with queues and stacks. We have also implemented the SCM analyzer described in Chapter 5. We are still improving this analyzer and making it more user-friendly. We also modified an existing interprocedural analyzer and we experimented it on some toy examples.

Perspectives

The current version of the SCM analyzer only manipulates numerical abstract domains. The definition of lattice automata is however not restricted to this particular case. We plan to use a combination of boolean and numerical abstract domains, like in the interprocedural analyzer, so that the SCM analyzer can deal with protocols having boolean variables. Later, we hope to fully exploit the aspect of "abstract lattice functor" of the lattice automata, and to use any abstract lattice, including the lattice automata themselves.

As mentioned in Chapters 3 and 5, the SCM analyzer cannot ensure that we get the exact reachability set. In order to prove that a property is not satisfied, we must have, in addition to our analyzer, a semi-algorithm able to compute the exact reachability set. We already discussed the possibility of combining our analysis with the QDDs acceleration techniques. We can also combine it with an approach based on the counter-examples guided abstraction refinement [CGJ$^+$00].

This approach consists in considering a finite abstraction of an infinite state-space, based on a finite set of predicates, and then in employing model checking techniques. The result is either a proof that the system satisfies a given safety property, or an abstract counter-example. We then check whether this counter-example is a real one, and if it is not a real one, we refine the abstraction by adding new predicates. The idea is to reuse the operator $\rho_k : \mathrm{Reg}(\Sigma) \to \mathrm{Reg}(\Sigma)$, defined in Chapter 3, to generate new predicates.

Another future research topic is to look at other systems where the configurations are represented by finite words. An example of such systems was presented in Chapter 2, when we introduced the transducers: the configurations of some identical processes sharing a token are represented by finite words over the alphabet $\Sigma = \{0, 1\}$. If the processes have integer or real variables instead of being finite automata, a configuration of the system is a finite word over an infinite alphabet. We expect to reuse lattice automata, or transducers based on the same principle as lattice automata, to analyze this kind of systems.

We are also studying how to manipulate tree automata containing some lattice elements, in the same way as we defined the lattice automata of Chapter 4. One possibility

is to consider the elements of a lattice Λ as ground terms[2] of the terms recognized by a tree automaton [CDG+02, GT95]. In this context, we must define a widening operator on those tree automata by combining the widening operator ∇_Λ with a widening operator on tree automata. One of the interests of such tree automata is to give an effective representation of terms containing integers, instead of representing those integers by terms of the form $s(s(s(\dots s(0)\dots)))$. This representation will simplify some methods of verification of cryptographic protocols, like the one of [GK00].

Other ongoing works include the supervisory control of FIFO systems. While verification techniques can prove that a system satisfies a property, the goal of supervisory control is to "repair" a faulty system by adding a supervisor. This supervisor can disable some events, thus restricting the behavior of the system. There are well-know algorithms to synthesize the supervisor when the considered system is finite [WR87, RW89, CL99]. The events of the system are either controllable or uncontrollable. One of the major difficulties of this synthesis is to compute the set of states B that violate a given property, and then the states that can lead to B by a sequence of uncontrollable events. Thus, it is basically a reachability problem. When the reachability problem is undecidable, classical algorithms cannot be applied directly.

In [LGJM05, LGJM06], we suggested to use abstract interpretation to overcome the undecidability issue. Since the synthesis involves a fix-point computation, we compute an over-approximation of this fix-point thanks to abstract interpretation methods. We then obtain a valid supervisor, ensuring the considered property, but also forbidding some non-faulty behaviors. When we proposed this method, we lacked a lattice abstracting sets of queue contents.

The abstract lattice proposed in this thesis allows us to compute the supervisor of a FIFO channel system. The computation of a centralized supervisor that can observe everything is quite easy, but not realistic. When a system is composed of several units communicating via FIFO channels, we have to solve two problems. First, some parts of the system (*e.g.* the FIFO queues) are not observable; so we must study how to define, within the abstract interpretation framework, the supervisory control of systems under partial observation. We then expect to have a small supervisor for each unit instead of a big one for the global system, in other words we want to decentralize the supervisor.

[2] Assuming we have a set of symbols $\mathcal{F}$, the terms are recursively defined by $t = f(t_1, \dots, t_n)$ if the arity of $f \in \mathcal{F}$ is n and $t_1, \dots t_n$ are terms already defined. A ground term is a symbol of arity 0.

Bibliography

[AAB99] Parosh Aziz Abdulla, Aurore Annichini, and Ahmed Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 208–222. Springer-Verlag, 1999.

[ABJ98] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 305–318, London, UK, 1998. Springer-Verlag.

[APR] The APRON library. `http://apron.cri.ensmp.fr/`.

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model checking. In *Int. Conf. on Concurrency Theory, CONCUR'97*, volume 1243 of *LNCS*, 1997.

[BET03] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL'03*, pages 62–73. ACM, 2003.

[BFG+00a] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: A validation environment for timed asynchronous systems. In *Computer Aided Verification*, pages 543–547, 2000.

[BFG+00b] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and test generation for the SSCOP protocol. *Scientific Computer Programming*, 36(1), 2000.

[BG97] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. *Form. Methods Syst. Des.*, 14(3):237–255, 1997.

[BGWW97] Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. The power of qdds. In *SAS '97: Proceedings of the 4th International Sym-*

posium on Static Analysis, pages 172–186, London, UK, 1997. Springer-Verlag.

[BH99] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. *Theor. Comput. Sci.*, 221(1-2):211–250, 1999.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Thomas Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *Static Analysis Symposium, SAS'06*, volume 4218 of *LNCS*, 2006.

[BHV04] Ahmed Bouajjani, Peter Habermehl, and Tomás Vojnar. Abstract regular model checking. In *CAV*, pages 372–386, 2004.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Computer Aided Verification*, pages 403–418, 2000.

[BLW03] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In *CAV*, pages 223–235, 2003.

[BMS+06] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *Symp. on Logic in Computer Science,LICS '06*, 2006.

[Bou90] François Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *PLILP'90*, LNCS 456, pages 307–323. Springer, 1990.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, Ecole Polytechnique, 1992.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop*, LNCS 1885, pages 113–130. Springer, 2000.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4), 1964.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[CC77a] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Description of Programming Concepts*, pages 237–277. North Holland, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2002.

[CFI96] Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 1996.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM Press.

[CI96] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6), 1996.

[CL99] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.

[CLi] The CamlLib Library
. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/camllib/`.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.

[EK99] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Int. Conf. on Foundations of Software Science and Computation Structure (FoSSaCS '99)*, volume 1578 of *LNCS*, March 1999.

[ES01a] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification, CAV'01*, volume 2102 of *LNCS*, 2001.

[ES01b] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *CAV'01*, LNCS 2102, pages 324–336. Springer, 2001.

[Fer01] Jérôme Feret. Abstract interpretation-based static analysis of mobile ambients. In *Eighth International Static Analysis Symposium (SAS'01)*, number 2126 in LNCS. Springer-Verlag, 2001. © Springer-Verlag.

[FIS03] Alain Finkel, S. Purushothaman Iyer, and Grégoire Sutre. Well-abstracted transition systems: application to fifo automata. *Information and Computation*, 181(1):1–31, 2003.

[Fix] Fixpoint: an OCaml library implementing a generic fix-point engine . `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/`.

[FM82] Alain Finkel and Gérard Memmi. Fifo nets: a new model of parallel computation. In *Proceedings of the 6th GI-Conference on Theoretical Computer Science*, pages 111–121, London, UK, 1982. Springer-Verlag.

[GFo] INRIA GForge website, Bertrand Jeannet's project. `http://gforge.inria.fr/scm/?group_id=841`.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Conference on Automated Deduction*, pages 271–290, 2000.

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, August 2007.

[GT95] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. 24:157–175, 1995.

[HJU79] Hopcroft, J.E., and J.D. Ullman. *introduction to automata theory languages and computation*. Addison-Wesley, 1979.

[HPR97a] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[HPR97b] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[HSS+93] Masahiro Higuchi, Osamu Shirakawa, Hiroyuki Seki, Mamoru Fujii, and Tadao Kasami. A verification procedure via invariant for extended communicating finite-state machines. In *Computer Aided Verification,CAV '92*, volume 663 of *LNCS*, 1993.

[int] Interproc: interprocedural analyzer for an imperative language with (recursive) procedure calls. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`.

[ITU99] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, 1999.

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *LNCS*, Venezia (Italy), Septembe 1999.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 220–234, 2000.

[JR87] C. Jard and M. Raynal. Specification of properties is required to verify distributed algorithms. In *Colloque AFCET sur les langages de spécification*, février 1987.

[JS04] Bertrand Jeannet and Wedelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Int. Conf. on Algebraic Methodology and Software Technology, AMAST'04*, volume 3116 of *LNCS*, July 2004.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6, 1976.

[KF94] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2), 1994.

[KS92] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC'92*, LNCS 641, pages 125–140. Springer, 1992.

[LGJ07] T. Le Gall and B. Jeannet. Lattice automata: a representation of languages over an infinite alphabet, and some applications to verification. In *The 14th International Static Analysis Symposium, SAS 2007*, number 4634 in LNCS, pages 52–68, Kongens Lyngby, Denmark, August 2007.

[LGJJ06] T. Le Gall, B. Jeannet, and T. Jéron. Verification of communication protocols using abstract interpretation of fifo queues. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology, AMAST '06, Kuressaare, Estonia*, LNCS. Springer-Verlag, July 2006.

[LGJM05] T. Le Gall, B. Jeannet, and H. Marchand. Supervisory control of infinite symbolic systems using abstract interpretation. In *44nd IEEE Conference on Decision and Control (CDC'05) and Control and European Control Conference ECC 2005*, pages 31–35, Seville (Spain), December 2005.

[LGJM06] T. Le Gall, B. Jeannet, and H. Marchand. Contrôle de systèmes symboliques, discrets ou hybrides. *Technique et Science Informatiques (TSI)*, 25:293–319, 2006.

[LHR97] David Lesens, Nicolas Halbwachs, and Pascal Raymond. Automatic verification of parameterized linear networks of processes. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 346–357. ACM Press, 1997.

[LKRT07] Akash Lal, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Abstract error projection. In *Static Analysis Symposium, SAS'07*, volume 4634 of *LNCS*, August 2007.

[LRMS96] David Lee, K. K. Ramakrishnan, W. Melody Moh, and Udaya Shankar. Protocol specification using parameterized communicating extended finite state machines. In *Int. Conf. on Network Protocols, ICNP'96*, 1996.

[Min01] A. Miné. The octagon abstract domain. In *Proc. of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, IEEE, pages 310–319, Stuttgart, Gernamy, October 2001. IEEE CS Press.

[MO83] Jose S. Metos and John V. Oldfield. Binary decision diagrams: From abstract representations to physical implementations. In *Proceedings of the 20th conference on Design automation*, pages 567–570. IEEE Press, 1983.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming, ESOP'05*, volume 3444 of *LNCS*, 2005.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. In *Symp. on Principles of Database Systems*, 2000.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part 1. *Theoretical Computer Science*, 13, 1981.

[NSV01] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In *Mathematical Foundations of Computer Science, MFCS'01*, volume 2136 of *LNCS*, 2001.

[OCa] The programming language Objective CAML. `http://caml.inria.fr/`.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[RHS95] Tom Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM, 1995.

[RSJ03] T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *SAS'03*, LNCS 2694. Springer, 2003.

[Rus03] V. Rusu. Combining formal verification and conformance testing for validating reactive systems. *Journal of Software Testing, Verification, and Reliability*, 13(3), September 2003.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[SRH96] Mooly Sagiv, Tom Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *TCS*, 167(1–2):131–170, 1996.

[Tur93] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[WB95] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 21–32, London, UK, 1995. Springer-Verlag.

[WR87] W. Wonham and P. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

List of Figures

Résumé

L'analyse des systèmes communiquant par file a fait l'objet d'études scientifiques nombreuses mais qui n'ont pu offrir des solutions totalement satisfaisantes. Nous proposons d'aborder ce problème dans le cadre de l'interprétation abstraite, en définissant des treillis abstraits adaptés à ce type de systèmes. Nous considérons d'abord le cas où les messages échangés sont assimilables à un alphabet fini, puis nous nous attaquons au cas, plus difficile, des messages portant des valeurs entières ou réelles. Ce problème nous amène à définir et à étudier un nouveau type d'automate, les automates de treillis. Ces automates de treillis peuvent également être utilisés pour l'analyses des programmes utilisant une pile d'appels.

Abstract

Many scientific studies analysed the FIFO channel systems, but none offered a fully satisfying solution. We propose to tackle this problem within the abstract interpretation framework, by defining some abstract lattices adapted to this kind of systems. We first consider systems with a finite alphabet of messages, then we consider more complex systems, with an infinite alphabet of messages. This leads us to define and to study a new kind of automata: the lattice automata. Those automata are also useful for the analysis of programs with a call stack.