UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Peter Schrammel

Thèse dirigée par **Alain Girault**
et codirigée par **Bertrand Jeannet**

préparée au sein de l'**INRIA Grenoble – Rhône-Alpes**, du **Laboratoire d'Informatique de Grenoble** et de l'**École doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Logico-Numerical Verification Methods for Discrete and Hybrid Systems

Méthodes logico-numériques pour la vérification des systèmes discrets et hybrides

Thèse soutenue publiquement le **18 octobre 2012**,
devant le jury composé de :

M Marc Pouzet
Professeur ENS, LIENS Paris, Président
M Laurent Fribourg
Directeur de recherche CNRS, LSV Cachan, Rapporteur
M Éric Goubault
Directeur de recherche CEA, LIST Saclay, Rapporteur
Mme Laure Danthony-Gonnord
Maître de conférences, LIFL Lille, Examinatrice
M Jérôme Leroux
Chargé de recherche CNRS, LaBRI Bordeaux, Examinateur
M Bertrand Jeannet
Chargé de recherche INRIA, LIG Grenoble, Co-Directeur de thèse
M Alain Girault
Directeur de recherche INRIA, LIG Grenoble, Directeur de thèse

Acknowledgements

First of all, my thanks are directed to Bertrand Jeannet, who guided me with patience and consideration during the last three years, giving me the freedom to follow my own thoughts, but channelling the ideas towards the essential where necessary. Altogether, this resulted in smooth and productive collaborations, and consequently, in marvelous journeys to conferences around the globe.

I am extremely grateful to Alain Girault for sparking his enthusiasm for science, for his encouragements and his advice in any scientific or non-scientific issue, for carefully reviewing the manuscript, and inviting me to his "soirées de jeux".

I would like to thank the members of the jury, Marc Pouzet, Laurent Fribourg, Éric Goubault, Jérôme Leroux, and especially Laure Gonnord, whose work gave me inspiration for large parts of my thesis. I thank Thomas Gawlitza for the passionate discussions about the semantics of hybrid systems. He also stimulated the successful collaboration w.r.t. max-strategy iteration with Pavle Subotic, who helped me to tune the implementation and to review the resulting paper.

I have to thank the Inria laboratory in Montbonnot for providing me an office that offered me a magnificent view of the mountains everyday. Special thanks to the Inria large-scale initiative Synchronics, which financed my PhD position, and to the collegues involved, Gwenaël Delaval, Marc Pouzet, Timothy Bourke, Benoît Caillaud and many others, for infecting me with their synchronous spirit and pointing me to interesting research directions. I would like to thank the members of the Asopt and Vedecy projects, who helped me keeping up-to-date on abstract interpretation and hybrid system analysis issues and offered me a forum for discussing my work.

I am also grateful to Pascal Bertolino from IUT2 Grenoble for giving me the opportunity to teach undergraduates during the last two years.

Thanks to my office mates of the Pop Art team Marnes Hoff, Pascal Sotin, Henri-Charles Blondeel, and Sebti Mouelhi and the other current and former members of the "équipe", Pascal Fradet, Gregor Goessler, Gideon Smeding, Vagelis Babelis, Lăcrămioara Aştefănoaei, Petro Poplavko, Roopak Sinha, Avinash Malik, and many others, for the diverting as well as inspiring discussions during lunch, coffee, and other social activities. In particular, I would like to thank the project assistant Diane for all her patience in handling the administrative affairs.

I especially thank my flat mates and all my other friends in Grenoble for making the weekends and evenings an incredible success. Last, but not least, I am deeply grateful to my parents, my relatives and friends, who supported me from the distance and called on once in a while.

Contents

Résumé en français

0.1 Introduction

Motivation et contexte scientifique

Les deux dernières décennies ont été marquées par une explosion du nombre et de la complexité des *systèmes embarqués*. De plus en plus de tâches critiques dans les domaines de l'industrie, du transport, de la production d'énergie et de la médecine, par exemple, sont confiées à des systèmes informatiques : la sécurité des personnes et des biens dépend de leur bon fonctionnement. La validation de ces systèmes devient donc un enjeu considérable et les méthodes de vérification formelle automatique gagnent de l'importance.

Ces systèmes embarqués sont des systèmes de contrôle réactifs dans lesquels des programmes informatiques sont en interaction permanente avec leur environnement physique par l'intermédiaire des capteurs et des actionneurs. Nous considérons des programmes écrits dans les *langages synchrones* comme Lustre [CPHP87] qui sont conçus pour la programmation des systèmes temps-réel. Ces programmes impliquent des variables booléennes et numériques. Ainsi, nous appelons ces systèmes *logico-numériques*. Un contrôleur avec son environnement physique constitue un *système hybride*, *i.e.*, un système ayant des comportements discrets et continus par rapport au temps.

Notre objectif est de vérifier des *propriétés de sûreté* qui expriment que « quelque chose de mauvais n'arrive jamais ». Ces propriétés peuvent être vérifiées par le calcul de toutes les configurations possibles dans les exécutions du système.

Nos techniques de vérification s'inscrivent dans le cadre de l'*interprétation abstraite* [CC77], une méthode d'analyse algorithmique de programmes, qui permet une analyse de systèmes d'états infinis avec terminaison garantie. Toutefois, comme le problème de la vérification générale est indécidable pour les programmes logico-numériques, ces propriétés sont acquises au prix de sur-approximations, ce qui rend le problème semi-décidable : l'analyseur répond « oui, propriété prouvée » ou « je ne sais pas ».

Problématique et objectifs

Cette thèse aborde les aspects suivants de la vérification des systèmes embarqués :

Le problème d'explosion de l'espace d'états : les techniques d'interprétation abstraite classiques ne traitent que des variables numériques. Dans le cas de programmes logico-numériques elles ont recours à l'énumération des états booléens. Cette énumération devient inapplicable déjà pour les programmes petits et moyens.

Nous prenons en compte cette préoccupation en suivant l'approche de Jeannet [Jea00] qui a proposé une méthode d'interprétation abstraite basée sur un domaine ab-

strait qui gère symboliquement les variables booléennes et qui utilise le partitionnement de l'espace d'états pour améliorer le compromis entre la précision et l'efficacité.

La perte de précision due à l'extrapolation : l'interprétation abstraite numérique utilise un opérateur d'extrapolation (l'« élargissement ») pour forcer la convergence de l'analyse, souvent entraînant une perte considérable de précision. Des méthodes diverses ont été proposées pour enrayer ce problème. Dans cette thèse nous considérons l'accélération abstraite [GH06], qui exploite la régularité du comportement de certaines opérations numériques dans les boucles du programme pour effectuer une extrapolation précise dans le domaine abstrait de polyèdres convexes. Nous étudierons une deuxième méthode, l'itération de max-stratégies [GS07a], qui calcule la meilleure approximation de l'espace d'état atteignable dans le domaine abstrait des polyèdres gabarits à l'aide de la programmation mathématique.

Notre objectif est d'améliorer la précision des analyses logico-numériques en adaptant ces méthodes au cadre logico-numérique.

La vérification des langages de programmation de haut niveau pour les systèmes hybrides : le succès des outils industriels de modélisation et simulation comme Simulink montre la nécessité d'avoir des langages de haut niveau dans la conception de systèmes embarqués. Cependant, la vérification de systèmes hybrides se fonde sur l'*automate hybride* [ACH+95], un modèle de bas niveau.

Notre objectif est de rendre accessibles les méthodes d'interprétation abstraite à ces langages de simulation hybrides par une compilation vers les automates hybrides qui surmonte ce décalage conceptuel.

Développement d'outil : nous devons mettre en œuvre nos méthodes de vérification pour évaluer leur efficacité et leur précision, ce qui est difficile à faire par des moyens théoriques.

Notre objectif est de développer un outil extensible qui est capable de se connecter à des langages de programmation et d'intégrer des méthodes d'analyse diverses pour pouvoir automatiser des comparaisons expérimentales. Nous baserons notre outil sur la bibliothèque des domaines abstraits Apron [JM09].

Plan

La thèse se divise en trois parties :

La première partie présente des contributions par rapport à *la vérification des programmes numériques* qui comprennent des extensions de l'accélération abstraite.

La deuxième partie traite *la vérification des programmes logico-numériques* : nous présentons des méthodes qui généralisent l'accélération abstraite et l'itération de max-stratégies aux programmes logico-numériques.

La troisième partie porte sur *la modélisation et la vérification des systèmes hybrides* : nous décrivons une traduction d'un langage flot de données hybride en automates hybrides logico-numériques et nous montrons comment étendre les méthodes existantes d'analyse de systèmes hybrides à de tels automates.

Finalement, nous décrivons notre outil de vérification, ReaVer, qui implémente les méthodes proposées.

0.2 Spécification et vérification des systèmes critiques

Nous considérons des programmes écrits dans des langages synchrones comme LUSTRE par exemple. Nous modélisons ces programmes comme des *systèmes de transition discrets* de la forme $\begin{cases} \mathcal{I}(\boldsymbol{s}) \\ \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \end{cases}$ où $\mathcal{I}$ définit les valeurs initiales des variables d'états $\boldsymbol{s}$ et $\boldsymbol{f}$ est la fonction de transition calculant la valeur suivante de $\boldsymbol{s}$ en fonction de l'état actuel et des entrées $\boldsymbol{i}$. Une exécution d'un tel système est une suite (infinie) d'états $\boldsymbol{s}_0 \xrightarrow{\boldsymbol{i}_0} \boldsymbol{s}_1 \xrightarrow{\boldsymbol{i}_1} \dots \boldsymbol{s}_k \xrightarrow{\boldsymbol{i}_k} \dots$

Propriétés et observateurs

Dans le contexte de langages synchrones une propriété de sûreté peut être spécifiée à l'aide d'un *observateur synchrone* [HLR93], qui est un programme écrit dans le même langage, composé de façon synchrone en parallèle avec le programme à vérifier. L'observateur a une sortie booléenne qui est vraie si le préfixe courant de l'exécution satisfait la propriété. Un observateur synchrone permet d'exprimer toute propriété de sûreté par l'invariant « la sortie de l'observateur est toujours vraie ».

Un invariant peut être vérifié en calculant l'ensemble d'états accessibles. Pour cette analyse d'accessibilité nous utilisons l'interprétation abstraite.

Analyse d'accessibilité par interprétation abstraite

L'ensemble d'états accessibles est défini par le plus petit point fixe de l'équation $S = \mathcal{I} \cup \mathsf{post}(S)$, *i.e.*, un état est accessible s'il se trouve dans l'ensemble initial $\mathcal{I}$ ou s'il est accessible par un pas d'exécution $\mathsf{post}(S) = \{\boldsymbol{s}' \mid \exists \boldsymbol{s} \in S, \exists \boldsymbol{i} : \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i})\}$.

L'idée de l'interprétation abstraite est de calculer ce point fixe dans un espace de propriétés plus simple, le *domaine abstrait*, d'une façon qu'il soit garanti de sur-approximer l'ensemble d'états accessibles concrets.

Un exemple d'un domaine abstrait numérique est les *polyèdres convexes* qui sont des conjonctions de contraintes linéaires $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}$ (où $\mathbf{A}$ est une matrice constante, $\boldsymbol{x}$ est le vecteur des variables d'états et $\boldsymbol{b}$ est un vecteur constant). Les opérations sur les domaines abstraits incluent par exemple l'union $\sqcup$, l'intersection $\sqcap$ et la projection, *i.e.*, la quantification existentielle d'une variable.

La méthode classique du calcul de point fixe repose sur l'*itération de Kleene* et un *opérateur d'élargissement* qui garantit la terminaison dans le cas général.

Comme l'élargissement est souvent cause de perte importante de précision, nous considérons ci-après des méthodes d'accélération qui améliorent la précision de l'analyse.

0.3 Vérification de systèmes numériques

Les méthodes d'*accélération* [BW94, FO97, BFLP03] visent à calculer l'ensemble exact des états accessibles dans les systèmes de transition numériques. Contrairement à l'interprétation abstraite, qui surmonte le problème d'indécidabilité en calculant une approximation conservatrice, l'accélération identifie des classes de systèmes pour lesquelles l'accessibilité est décidable et peut être résolue exactement. L'idée est d'accélérer les

relations de transition τ associées aux cycles dans la structure de contrôle d'un programme en calculant l'effet de leur fermeture réflexive et transitive $\tau^* = \bigcup_{k\geqslant 0} \tau^k$ sur un ensemble d'états X.

Gonnord et al. [GH06, Gon07] ont proposé le concept de l'*accélération abstraite* qui intègre l'idée d'accélération dans le cadre d'interprétation abstraite avec des polyèdres convexes : les boucles simples (« self loops ») sont accélérées dans le domaine abstrait quand c'est possible, et dans les autres cas (boucles imbriquées, transitions trop expressives) on recourt à l'élargissement pour garantir la convergence du calcul de point fixe.

L'accélération abstraite considère des transitions sous forme de commandes gardées :

$$\tau : \underbrace{\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}}_{\text{garde}} \rightarrow \underbrace{\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}}_{\text{commande}}.$$

De telles transitions sont considérées accélérables si $\langle C^*, \cdot \rangle$ est un monoïde fini, *i.e.*, l'ensemble $C^* = \{\mathbf{C}^k \mid k \geqslant 0\}$ est fini.

Gonnord et al. fournissent des accélérations abstraites pour les cas de *translations* ($\mathbf{C} = \mathbf{I}$) et de *translations avec resets* ($\mathbf{C}$ est diagonale avec des coefficients $\in \{0, 1\}$). Les autres cas de transitions accélérables (« ultimement périodiques ») sont ramenés à ces deux cas par un changement de base.

0.3.1 Extensions de l'accélération abstraite

Nous étendons l'accélération abstraite aux systèmes avec des entrées numériques, et nous formulons aussi l'accélération abstraite en arrière.

Accélération abstraite avec des entrées numériques

Les programmes réactifs, tels que les programmes Lustre, interagissent avec leur environnement : à chaque pas de calcul ils prennent en compte les valeurs des variables d'entrée. Les variables d'entrée booléennes peuvent être codées dans la structure de contrôle par des choix non-déterministes finis. Les variables d'entrée numériques $\boldsymbol{\xi}$, par contre, exigent un traitement plus spécifique.

Nous considérons des transitions de la forme

$$\tau : \underbrace{\begin{pmatrix} \mathbf{A} & \mathbf{L} \\ \mathbf{0} & \mathbf{J} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\xi} \end{pmatrix} \leqslant \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{k} \end{pmatrix}}_{\mathbf{A}\boldsymbol{x}+\mathbf{L}\boldsymbol{\xi}\leqslant\boldsymbol{b} \,\wedge\, \mathbf{J}\boldsymbol{\xi}\leqslant\boldsymbol{k}} \rightarrow \boldsymbol{x}' = \underbrace{\begin{pmatrix} \mathbf{C} & \mathbf{T} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{\xi} \end{pmatrix} + \boldsymbol{u}}_{\mathbf{C}\boldsymbol{x}+\mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}}$$

Le défi posé par l'ajout des entrées se montre par le fait que toute transformation affine générale $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \rightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ peut être exprimée par un reset à une entrée $(\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{\xi} = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}) \rightarrow \boldsymbol{x}' = \boldsymbol{\xi}$. Comme l'accélération de transformations affines générales, même sans entrées, est hors de portée de l'état de l'art actuel, il n'y a aucun espoir d'obtenir une accélération précise pour ces transitions.

Néanmoins, nous pouvons accélérer les transitions avec des entrées si les variables d'état et les entrées ne sont pas liées dans la garde, *i.e.*, la garde est de la forme $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}$. Nous proposons une accélération abstraite dans le cas de ces gardes dites « simples ». Dans le cas de gardes « générales » nous relaxons, *i.e.*, sur-approximons, la garde afin d'obtenir des gardes simples : $\overline{G} = (\exists \boldsymbol{\xi} : G) \wedge (\exists \boldsymbol{x} : G)$.

Dans le cas de gardes simples nous réécrivons la translation τ avec des entrées par une translation $\tau : G \rightarrow \boldsymbol{x}' = \boldsymbol{x} + D$ sans entrées mais où D est un polyèdre. Cela

s'explique par le fait que à chaque pas d'exécution $\boldsymbol{x}$ est translaté par $\boldsymbol{T\xi} + \boldsymbol{u}$ où $\boldsymbol{\xi}$ est inclus dans le polyèdre $\mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}$ et donc $\boldsymbol{T\xi} + \boldsymbol{u}$ est un polyèdre également. Nous pouvons accélérer cette transition en calculant $\tau^{\otimes}(X) = X \sqcup \tau\big((X \sqcap G) \nearrow D\big)$, *i.e.*, nous intersectons l'ensemble de départ X avec la garde G, puis nous y additionnons D un nombre $\alpha \in \mathbb{R}^{\geqslant 0}$ de fois en utilisant l'opérateur de « passage de temps » [HPR97], ce qui est une opération triviale dans le domaine de polyèdres convexes ; et finalement, nous appliquons la transition τ une dernière fois et nous calculons l'union avec X.

L'accélération de translations avec resets suit la même idée mais avec la différence qu'il faut distinguer les variables translatées des variables réinitialisées dans le calcul.

Nous prouvons la correction de nos formules d'accélération et nous montrons la nature des approximations impliquées. En outre, nous comparons l'accélération abstraite à des autres approches basées sur l'interprétation abstraite.

Accélération abstraite en arrière

Nous fournissons une accélération abstraite en arrière pour les translations et les translations avec resets. Bien que l'inverse d'une translation soit une translation, la différence est que l'intersection avec la garde survient après la translation (inversée). Le cas des translations avec resets est plus compliqué que dans l'accélération en avant. La relaxation des gardes générales aux gardes simples s'applique de la même manière à l'accélération en arrière.

0.3.2 Généralisation de l'accélérabilité linéaire

Jusqu'à présent, nous n'avons considéré que les translations et les resets. Cependant, nos expériences ont montré que d'autres types de transitions sont aussi fréquents dans les programmes synchrones : les échanges de variables ($x'_1 = x_2$; $x'_2 = x_1$ par exemple) et les retards ($x'_1 = x_2$; $x'_2 = x_3$ par exemple) sont des transformations (ultimement) périodiques; par conséquent, nous savons comment les accélérer en suivant l'approche de Gonnord et al. Cependant, les dépendances des variables non-modifiées ($x'_1 = x_1 + x_2$; $x'_2 = x_2$ par exemple) ne sont pas considérés accélérables dans la théorie de l'accélération exacte. Néanmoins, intuitivement, nous devrions être capables de les accélérer parce que la variable x_2 est constante pendant les itérations de la boucle. Cette observation nous a conduit à réexaminer le concept de l'accélérabilité linéaire.

Une transition τ est *linéairement accélérable* si sa fermeture réflexive et transitive τ^* peut être réécrite en $\tau^* = \lambda X. \bigcup_i \{\mathbf{A}_i\boldsymbol{x} + k\boldsymbol{b}_i \mid k \geqslant 0, \boldsymbol{x} \in X\}$, *i.e.*, elle peut être représentée par une union finie de transformations linéaires et de translations.

Le critère classique d'accélération exacte d'une transformation affine $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ se fonde uniquement sur la matrice $\mathbf{C}$ ($\langle C^*, \cdot \rangle$ avec $C^* = \{\mathbf{C}^k \mid k \geqslant 0\}$ est un monoïde fini). Nous considérons ici un critère qui se base sur la forme homogène linéaire $\begin{pmatrix} \boldsymbol{x}' \\ \chi' \end{pmatrix} = \begin{pmatrix} \mathbf{C} & \boldsymbol{d} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \chi \end{pmatrix}$ (avec $\chi \equiv 1$) d'une transformation affine.

Nous montrons par *décomposition en forme de Jordan* qu'une transformation linéaire est linéairement accélérable si

– les blocs de Jordan de taille 1 sont associés à des valeurs propres qui sont soit 0 soit des racines complexes de l'unité ;

– les blocs de Jordan de taille 2 sont associés à des valeurs propres qui sont soit 0 soit des racines complexes de l'unité et dans ce dernier cas la variable correspondant à la deuxième dimension d'un tel bloc a un nombre fini de valeurs dans l'ensemble initial (dans la base de Jordan) ; et
– les blocs de Jordan d'une taille arbitraire strictement supérieure à 2 sont associés à la valeur propre 0.

Ce critère est plus général que celui du monoïde fini, qui restreint les blocs de Jordan de taille 2 à des valeurs propres 0 et 1, et dans le dernier cas, la variable correspondant à la deuxième dimension d'un tel bloc doit valoir 1 dans l'ensemble initial.

Nous proposons des formules d'accélération abstraite pour les cas non-couverts par le critère du monoïde fini, qui correspondent aux transitions avec des dépendances des variables non-modifiées ou périodiques.

0.4 Vérification de systèmes logico-numériques

L'approche classique de l'application des méthodes numériques comme l'accélération abstraite aux programmes flot de données logico-numériques repose sur l'énumération de l'espace d'états booléen pour générer un graphe de contrôle purement numérique.

Par contraste, nous utilisons des analyses fondées sur des domaines abstraits logico-numériques qui permettent de traiter les variables booléennes ainsi que les variables numériques d'une manière symbolique. Ces domaines peuvent être construits en combinant des domaines abstraits Booléens et numériques afin de construire des domaines du type « produit » ($A \times B$) ou « puissance » ($A \to B$). Nous considérons les domaines $\wp(\mathbb{B}^m) \times \mathcal{N}(\mathbb{R}^n)$ [Jea00] et $\mathbb{B}^m \to \mathcal{N}(\mathbb{R}^n)$ [BCC+03, Jea] où la partie booléenne est représentée d'une manière exacte et le domaine $\mathcal{N}$ est par exemple les polyèdres convexes.

En plus, nous utilisons le partitionnement de l'espace d'états pour générer un graphe de contrôle, ce qui permet d'améliorer la précision en associant une valeur abstraite à chaque place du graphe. Le choix de la partition influence donc la précision et l'efficacité de l'analyse.

Nous dénotons $\boldsymbol{s} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{x} \end{pmatrix}$ resp. $\boldsymbol{i} = \begin{pmatrix} \boldsymbol{\beta} \\ \boldsymbol{\xi} \end{pmatrix}$ les vecteurs de variables d'état resp. d'entrées composés des sous-vecteurs booléen et numérique. Nous utilisons des représentations symboliques faisant usage de diagrammes de décisions binaires (Bdd) pour représenter les programmes logico-numériques et les domaines abstraits dédiés à leur analyse.

0.4.1 Accélération abstraite logico-numérique

Nous proposons une méthode pour analyser un programme logico-numérique à l'aide de l'accélération abstraite sans recourir à l'énumération d'états booléens, et une méthode de partitionnement efficace.

Accélération abstraite logico-numérique

L'accélération numérique peut être appliquée à des boucles où l'état numérique évolue tandis que l'état booléen reste constant, *i.e.*, la partie booléenne de la fonction de transition est l'identité. Cependant, il pourrait n'y avoir aucune boucle de cette sorte

dans le programme, tandis qu'en abstrayant légèrement le comportement des boucles nous pourrions bénéficier de techniques d'accélération précises plutôt que de compter sur l'élargissement qui risque de perdre beaucoup plus d'informations à la fin.

Nous considérons des boucles simples (« self-loops ») dont la transition est de la forme $\tau : g(\boldsymbol{s},\boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s},\boldsymbol{i}) \\ \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix}$ tel que $g(\boldsymbol{s},\boldsymbol{i}) = g^b(\boldsymbol{b},\boldsymbol{\beta}) \wedge g^x(\boldsymbol{x},\boldsymbol{\xi})$, et $\boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi})$ soit une transformation accélérable. Cette forme peut être obtenu par une factorisation de gardes des fonctions de transitions.

Notre accélération abstraite logico-numérique repose sur le découplage des parties booléennes et numériques de la fonction de transition :

$$\tau_b : g(\boldsymbol{s},\boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} f^b(\boldsymbol{s},\boldsymbol{i}) \\ \lambda(\boldsymbol{s},\boldsymbol{i}).\,\boldsymbol{x} \end{pmatrix} \text{ and } \tau_x : g(\boldsymbol{s},\boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s},\boldsymbol{i}).\,\boldsymbol{b} \\ \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix}.$$

Nous montrons que nous pouvons sur-approximer τ^* dans le domaine abstrait logico-numérique « produit » comme suit : nous accélérons d'abord la partie numérique τ_x en utilisant l'accélération abstraite numérique. Nous obtenons ainsi $X^{\otimes}$. Puis, nous itérons la partie booléenne partiellement évaluée sur $X^{\otimes}$ jusqu'à la convergence : $\tau^{\otimes}(B,X) = \left((\tau_b\,[X^{\otimes}])^*\,(B)\ ,\ X^{\otimes}\right)$.

Au premier coup d'œil les approximations induites par ce découplage partiel semblent plutôt grossières. Toutefois, cela n'est pas grave pour les raisons suivantes : Premièrement, les corrélations entre les variables booléennes et numériques qui sont perdues par notre méthode ne sont souvent pas représentable dans le domaine abstrait de toute façon, et deuxièmement, nous allons appliquer cette méthode à un graphe de contrôle obtenu par un partitionnement approprié, décrit ci-après.

Partitionnement par les modes numériques

L'idée est de regrouper dans une place ces états booléens qui impliquent le même comportement numérique (les « modes numériques ») et donc, le découplage des parties booléennes et numériques de la fonction de transition n'affectera probablement pas la précision.

Algorithmiquement, nous générons une telle partition efficacement en décomposant d'une manière adaptée les BDDs (diagrammes de décisions binaires) représentant les fonctions de transitions.

Résultats expérimentaux

Pour évaluer la précision et le passage à l'échelle de nos techniques nous avons effectué une comparaison avec les outils NBAC [Jea03] et ASPIC [Gon]. Notre outil REAVER est souvent en mesure de prouver les propriétés recherchées que les deux autres outils ne réussissent pas à montrer. Notre méthode de partitionnement génère des graphes de contrôle qui sont dix fois plus petits que ceux obtenus par l'énumération de l'espace d'états booléen (restreint aux états accessibles). Cela nous permet d'analyser de plus grands systèmes tout en étant suffisamment précis grâce à l'accélération abstraite logico-numérique.

L'accélération abstraite améliore la précision en réduisant le besoin de l'élargissement. Dans le chapitre suivant nous considérons les méthodes d'itération de strategies qui ne necéssitent pas d'opérateur de l'élargissement du tout.

0.4.2 Itération de max-stratégies logico-numérique

Les méthodes d'itération de stratégie résolvent les (in)équations de point fixe associées à l'analyse d'accessibilité par une séquence de calculs de point fixe sur des systèmes plus simples. Elles peuvent être appliquées à des domaines gabarits (templates) avec des formes de contraintes finies et a priori fixées, comme par exemple les *polyèdres gabarits*, *i.e.*, des contraintes de la forme $\mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d}$. Les bornes constantes $\boldsymbol{d}$ sont déterminées lors de l'analyse à l'aide de la programmation linéaire.

Itération de max-stratégies numérique

Nous considérons l'itération de max-stratégies [GS07a] pour les programmes affines, qui est une méthode de résolution de systèmes de point fixe d'inéquations sémantiques (sur les variables de bornes de polyèdres gabarits). Une stratégie contient exactement une inéquation par variable. Le point fixe du système est approché par les points fixes des stratégies successivement améliorées (en remplaçant une inéquation par une autre) jusqu'à ce que le point fixe du système soit atteint. Il est garanti que ce point fixe est le plus petit.

Itération de max-stratégies logico-numérique

Nous proposons un algorithme qui permet d'appliquer l'itération de stratégie numérique à des programmes logico-numériques, mais en comparaison avec une tentative précédente [SJVG11] qui necéssite de l'élargissement, notre méthode calcule toujours le plus petit point fixe dans le domaine logico-numérique.

Notre analyse se base sur l'alternance (1) d'une itération de Kleene tronquée dans un domaine abstrait logico-numérique avec des polyèdres gabarits et (2) l'itération de max-stratégies numériques.

L'itération de Kleene tronquée (propagation) explore le système jusqu'à ce que, pour tous les point de contrôle, l'ensemble des états accessibles booléens ne change pas quelle que soit la transition que nous prenons. L'idée sous-jacente est de découvrir un sous-système, dans lequel les variables booléennes sont stables pendant plusieurs itérations. Dans un tel sous-système, les variables numériques peuvent évoluer pendant que les transitions booléennes restent à l'intérieur du système, *i.e.*, elles ne « découvrent » pas de nouveaux états booléens, jusqu'à ce que des conditions numériques soient remplies qui fassent que les variables booléennes « quittent » le sous-système.

On utilise l'itération de max-stratégies (phase (2)) pour calculer le point fixe pour les variables numériques pour un tel sous-système. Puis, l'itération de Kleene (phase (1)) continue à explorer un autre sous-système temporairement stable. L'algorithme se termine en un nombre fini d'étapes dès que l'itération de Kleene de la phase (1) a atteint un point fixe.

Résultats expérimentaux

Pour évaluer la précision et l'efficacité de nos techniques nous avons effectué une comparaison entre l'itération de max-stratégies numériques basée sur l'énumération des booléens et notre méthode logico-numérique. Les résultats montrent que, grâce à l'approche logico-numérique, nous gagnons un ordre de grandeur en temps de calcul tout en étant aussi précis.

En comparaison avec l'accélération abstraite il faut remarquer que bien que l'itération de max-strategies soit capable de calculer le plus petit point fixe, elle ne le peut que sur des formes de polyèdres donnés, tandis que l'accélération abstraite est en mesure de « découvrir » des nouvelles contraintes d'invariants.

Jusqu'à présent nous n'avons considére que des systèmes discrets. Dans la suite nous allons étendre les concepts logico-numériques aux contrôleurs discrets en interaction avec leur environnement physique continu, *i.e.*, les systèmes hybrides.

0.5 Modélisation et vérification de systèmes hybrides

Le modèle classique pour la vérification de systèmes hybrides est l'automate hybride [ACH+95], qui combine un système de transition discret avec des équations différentielles.

En revanche, pour la modélisation en industrie, les outils et langages de simulation numériques comme Simulink [Sim] sont le plus largement utilisés. Pourtant, leur sémantiques ont quelques particularités.

Le langage de programmation hybride synchrone Zelus [BBCP11b] aborde ces questions. Sa sémantique est basée sur l'analyse non-standard [Rob96, Lin88], ce qui permet de spécifier une sémantique déterministe du système hybride indépendamment des questions d'intégration numérique. De plus, elle s'intègre bien avec les langages synchrones.

0.5.1 Traduction des langages flot de données hybrides vers les automates hybrides logico-numériques

Notre objectif est de vérifier les systèmes hybrides écrits dans des langages de simulation comme Simulink ou Zelus. Cependant, il y a un décalage conceptuel entre ces langages de haut niveau et les automates hybrides, la représentation adaptée pour la vérification : dans les premiers, les transitions discrètes sont déclenchées par des zéro-crossings – des événements survenant lorsqu'une fonction passe du négatif au positif – tandis que dans les derniers les comportements discrets sont dirigés par des combinaisons d'invariants de place et les gardes des transitions discrètes – dans un automate hybride les variables continues peuvent évoluer aussi longtemps que l'invariant de place est satisfait et une transition discrète peut être prise pendant que la garde est satisfaite.

Notre objectif est de formaliser la traduction d'un *formalisme flot de données hybride*, qui nous sert comme une base commune afin d'abstraire des constructions de haut niveau de langages comme Simulink ou Zelus, vers les *automates hybrides logico-numériques*, qui permettent une représentation compacte sans exiger l'énumération de l'espace d'états discret.

Formalisme flot de données hybride

Ce formalisme étend les systèmes de transition discrets par des équations différentielles ordinaires (EDO) et des zéro-crossings: $\begin{cases} \mathcal{I}(\boldsymbol{s}) \\ \begin{cases} \dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s}, \boldsymbol{i}) \end{cases} \end{cases}$

où $\boldsymbol{s} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{x} \end{pmatrix}$, et $\dot{\boldsymbol{x}}$ est la dérivée de $\boldsymbol{x}$ par rapport au temps. Les modes continus

(dans $\boldsymbol{f}^c$) sont gardés par des variables discrètes $\left(\text{par exemple, } \dot{x} = \begin{cases} x & \text{if } b \\ -x & \text{if } \neg b \end{cases}\right)$, et les transitions discrètes (dans $\boldsymbol{f}^d$) sont gardées par des zéro-crossings (par example, $x' = y$ if $up(z)$, $up(z)$ étant la condition de franchissement du 0).

La sémantique est (comme celle de ZELUS) inspirée par le mode de fonctionnement d'un simulateur numérique, qui, après l'initialisation, intègre les EDOs jusqu'à ce qu'un zéro-crossing ait été activé. Puis les transitions discrètes sont effectuées, avant de reprendre l'intégration.

Automates hybrides logico-numériques

Nous étendons les automates hybrides par des variables discrètes booléennes (et numériques). Les relations de transitions continues, qui définissent les comportements continus et qui sont associées à des places de l'automate, ont donc la forme $V(\boldsymbol{b}, \boldsymbol{x}, \dot{\boldsymbol{x}})$. Les relations associées aux transitions discrètes ont la forme $R(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{b'}, \boldsymbol{x'})$.

Si nous éliminons toutes les variables booléennes en codant leurs valeurs en des places, la sémantique d'un tel automate est égale à celle d'un automate hybride standard qui ne traite que des variables numériques.

Traduction

La question principale dans la traduction est la question des zéro-crossings : la différence fondamentale entre le concept de zéro-crossings utilisé dans le langage d'entrée et la combinaison d'invariants de place et de gardes dans le langage de sortie est que l'activation d'un zéro-crossing dépend de l'histoire (c'est-à-dire une partie de la trajectoire antérieure) tandis que la satisfaction d'invariants de place et de gardes ne dépend que de l'état actuel.

Donc, le principe de notre traduction consiste à ajouter des états discrets pour mémoriser l'historique de l'évolution continue. Nous discutons trois choix naturels pour la sémantique de zéro-crossing, et nous montrons comment traduire des zéro-crossings sans et avec des entrées, des combinaisons logiques de zéro-crossings et les zéro-crossings déclenchés par des transitions discretes.

Dans tous les cas, en raison des limitations du modèle d'automates hybrides, la traduction ajoutera des comportements qui ne sont pas présents dans le programme original. Par conséquent, la traduction entraîne une sur-approximation en termes de traces et d'états accessibles. Nous discutons la nature et l'ampleur de ces approximations.

La traduction vers les automates hybrides nous permet d'utiliser les outils actuels de vérification hybrides tels que HYTECH [HHWT97], PHAVER [Fre05] et SPACEEX [FGD+11]. Cependant, ces outils nécessitent de coder les états booléens explicitement en des places, ce qui peut causer une explosion de la taille de l'automate hybride. Pour éviter ce problème, nous appliquons le concept d'analyse logico-numérique aux méthodes d'analyse hybride dans le chapitre suivant.

0.5.2 Analyse des automates hybrides logico-numériques

En plus du calcul des transitions discrètes, l'analyse d'accessibilité des automates hybrides logico-numériques doit tenir compte des transitions continues. Le calcul de

cette évolution continue ressemble à l'accélération de boucles dans les systèmes discrets. Toutefois, par rapport à l'accélération abstraite logico-numérique, il est en fait plus simple, parce que seulement les variables continues (numériques) évoluent dans une transition continue, tandis que les variables discrètes (booléennes et numériques) restent constantes. Par conséquent, aucun découplage des variables booléennes et numériques n'est nécessaire comme nous avions dû le faire dans le cadre de l'accélération abstraite logico-numérique.

Analyse

Nous proposons une technique qui calcule à la volée des relations de transitions continues, convexes et spécialisées à l'état courant qui peuvent être traitées par des méthodes existantes d'accessibilité continue numérique.

Nous montrons comment appliquer cette technique à trois méthodes d'analyse hybrides d'horizon de temps non-borné : le passage de temps polyédrique [HPR97], l'itération de max-stratégies pour les systèmes hybrides affines [DG11a] et les abstractions relationnelles [ST11], une méthode qui abstrait les transitions continues par des boucles discrètes.

Partitionnement par les modes continus

Les relations continues des automates hybrides logico-numériques peuvent regrouper de nombreux comportements continus distincts, et donc l'analyse ne serait pas très précise.

Nous proposons deux méthodes de partitionnement pour découvrir des modes continus : pour détecter les modes définis par les états booléens nous appliquons la technique proposée dans le contexte de l'accélération abstraite logico-numérique aux relations de transitions continues. La deuxieme méthode se fonde sur une analyse disjonctive pour trouver les modes continus caractérisés par les états numériques discrets.

0.6 Implémentation

Nous avons mis en œuvre les méthodes de vérification développées tout au long de cette thèse dans un outil appelé ReaVer, REActive system VERifier.

Afin de rendre l'outil le plus générique et extensible possible, nous avons séparé l'implémentation en (1) un « framework » qui définit des concepts génériques (graphe de contrôle, domaine abstrait, analyse, etc) et fournit des fonctions auxiliaires, et (2) les implémentations de ces concepts qui composent l'outil. Le framework se base sur la librarie de domaines abstraites BddApron [Jea]. Les implémentations peuvent en outre appeler d'autres bibliothèques, comme des solveurs LP et SMT.

L'outil accepte les formats d'entrée suivants:

- Le format NBac [Jea00], qui est une description textuelle d'un système dynamique discret avec la spécification d'une propriété. Nous l'avons étendu au format Hybrid NBac en ajoutant des équations différentielles et un opérateur de zéro-crossing.
- Le langage Lustre, qui peut être analysé après transformation au format NBac à l'aide de l'outil Lus2NBac [Jea00].
- Un sous-ensemble de Zelus (resp., Lucid Synchrone), en se basant sur la partie amont du compilateur Zelus [BBCP11a].

Les programmes hybrides sont traduits vers les automates hybrides. Puis, les programmes sont partitionnés selon les modes numériques par exemple. Parmi les méthodes d'analyse disponibles, il y a l'analyse standard sans ou avec accélération abstraite, l'itération des max-stratégies (basée sur le solveur de [DG11a]), ainsi que leurs versions logico-numériques et hybrides.

0.7 Conclusions et perspectives

Dans le contexte de la vérification de propriétés de sûreté des systèmes embarqués, nous avons poursuivi l'objectif d'améliorer la précision des propriétés inférées tout en améliorant le passage à l'échelle vis-à-vis de la combinatoire booléenne. En plus, nous avons considéré l'analyse de systèmes embarqués modélisés dans les langages de simulation hybrides.

Notre approche se fonde sur des méthodes d'analyse logico-numériques, c'est-à-dire des méthodes d'interprétation abstraite qui sont capables de manipuler des variables booléennes de manière implicite à l'aide de domaines abstraits logico-numériques, et sur des méthodes de partitionnement de l'espace d'états.

Dans ce cadre, nous avons adapté des méthodes permettant d'améliorer la précision du contexte numérique au contexte logico-numérique.

En ce qui concerne les langages de simulation hybrides, nous avons proposé une traduction vers les automates hybrides logico-numériques, et nous avons montré que notre approche d'analyse logico-numérique s'applique également à de tels systèmes.

Nos expériences ont démontré que ces méthodes permettent d'améliorer l'efficacité des analyses par au moins un ordre de grandeur en comparaison aux approches purement numériques, tout en améliorant la précision des invariants calculés.

Perspectives. Nous pensons que le potentiel d'accélération abstraite numérique est loin d'être entièrement exploité. Ce travail est un premier pas vers la généralisation de ce concept. Des recherches plus poussées seront nécessaires afin de permettre le calcul d'approximations polyédriques qui sont en mesure de découvrir des invariants complexes, par exemple pour les transformations linéaires générales.

Notre itération de max-stratégies logico-numérique est basée sur un algorithme simple et générique. Il serait souhaitable de développer un solveur de max-stratégies logico-numérique plus intégrée afin d'améliorer les performances de cette approche.

Nous n'avons pas encore effectué une évaluation approfondie expérimentale de nos méthodes d'analyse de systèmes hybrides. À notre connaissance, il n'y a pas d'autres outils de vérification avec horizon de temps non-borné pour les systèmes hybrides logico-numériques. Cependant, nous pourrions comparer la version numérique avec la version logico-numérique des méthodes d'analyse hybride que nous avons choisies.

La version actuelle de notre outil ne prend qu'un sous-ensemble restreint du langage Zelus comme entrée. Nous devrions pousser notre développement vers une prise en compte plus complète. En outre, une intégration plus étroite avec la partie amont sera nécessaire pour mettre en œuvre une approche plus pratique de retour au programme source.

Finalement, il serait intéressant de considérer des applications de nos méthodes, par exemple dans l'analyse de systèmes échantillonnés, la synthèse de contrôleurs et paramètres, la génération de cas de test et le débogage.

Chapter 1

Introduction

1.1 Motivation

The last two decades were shaped by an explosion in number and complexity of computerized systems. Computers get embedded almost everywhere and integrate a multitude of functions: home appliances and entertainment, mobile phones, portable computers, MP3 players, digital cameras, navigation assistants, electronic books, wearable fitness devices, electronic identification, ticketing and payment devices, a.s.o. This development of *embedded systems* is even more remarkable in applications where the presence of computers is less perceived by the users: a middle-class car for instance contains nowadays around fifty computers! In these applications more and more safety-critical tasks like braking, steering and engine control are automated and entrusted to computer systems: the safety of life and property depends on their correct operation. *Safety-critical systems* are mainly encountered in the following areas:

- In *industry*, manufacturing lines and process control, *e.g.*, in chemical plants, are traditionally subject to automation.
- In *transport systems*, mechanics and humans are more and more replaced by computers: avionics, automotive, automatic urban trains, railway and road signaling.
- *Energy production* systems such as nuclear power plants and the control of energy distribution networks involve high environmental and financial risks.
- In *military* systems they are omnipresent *e.g.*, in transport and weapon systems, combat jets, cruise missiles, missile defense and unmanned aerial vehicles.
- *Space* applications have extreme requirements on fault tolerance: spacecrafts, satellites, autonomous space vehicles and life support systems for spacemen.
- *Building automation* comprises applications like heating and air-conditioning, elevators, and security and safety systems (*e.g.*, access control and fire detectors).
- *Medicine* is an expanding field for safety-critical applications like medical imaging, intensive care, radiotherapy, automated medication dispensing devices, implants (*e.g.*, pacemakers), prostheses, and remote surgery.

All these embedded systems are reactive control systems (Fig. 1.1) in which computer systems ("controllers") are in permanent interaction (feedback loops) with their physical environment ("plant") via sensors and actuators. Physical systems comprise, *inter alia*, electronics, mechanics, and hydraulics. These controllers are subject to real-time constraints and resource constraints (energy, weight, memory, processing power). The resource consumption and reaction time must be bounded statically (at compile

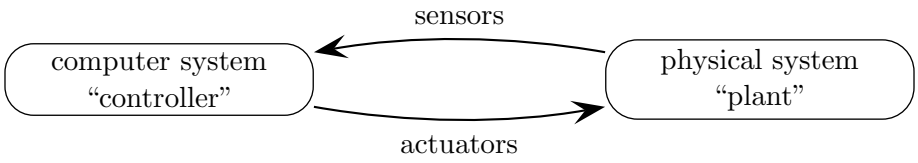


Figure 1.1: Embedded system

time). Systematic engineering processes (*e.g.*, V-model) support the development of such systems in order to detect conceptual errors as early as possible. In several application areas there are norms on the quality assurance regarding the system and software development and maintenance process, *e.g.*, DO-178B for avionics, and ISO/IEC-61508 for critical systems. Often certification by an authority is required before commissioning.

The *validation* process, i.e., checking compliance between intended and actual behavior, becomes increasingly expensive and a huge challenge for industry. There are two general approaches to validation: testing and formal verification.

Testing can detect errors and increase the level of confidence in software, but it is not exhaustive in general and therefore it cannot guarantee the absence of errors.

Formal verification mathematically proves the absence of errors. Formal verification methods come in two flavors: *Theorem proving* is a semi-automatic approach which requires a human to interact with a proof assistant tool. *Algorithmic methods* are implemented in fully automatic tools. They are widespread in industry, *e.g.*, in circuit verification, but they are still limited regarding software verification. The industrial need for automatic verification is confirmed by the success of tools like PolySpace[1] and Astrée [CCF+05], which can prove the absence of runtime errors in large embedded C programs.

Model-based design methods iterate refinement-validation phases in a top-down manner. *Software synthesis* enables to automate such refinements and, thus, systems are correct by construction. Actually, the methods used for synthesis and verification are closely related and face similar difficulties, because they are somehow inverse operations: verification checks the compliance of a behavior with a specification, whereas synthesis determines the behavior compliant with a specification.

In this thesis we will concentrate on algorithmic verification methods for embedded systems.

1.2 Scientific Context

Embedded systems are modeled as discrete or hybrid, *i.e.*, discrete and continuous, dynamical systems involving Boolean and numerical variables. Thus, we call such systems *logico-numerical*. Our goal is to check so-called *safety properties* which express that "something bad never happens". Such properties can be checked by computing the possible configurations of the system for any execution.

Synchronous languages. Synchronous languages were designed for programming reactive control systems, *i.e.*, embedded controllers. Examples for such languages are Lustre [CPHP87], Esterel [BC85] and Signal [BGJ91]. Their development was motivated by the fact that reactive systems are parallel systems by nature, but languages,

[1] `http://www.mathworks.fr/products/polyspace`

like assembler or C, are not adapted to dealing with the complexity of parallel systems. Hence, programming in such low-level languages is error-prone and programmers have to struggle with problems of non-determinism, race conditions and deadlocks. Moreover, these issues make the program analysis difficult.

In synchronous languages, parallel computations execute in lockstep in a sequence of synchronized *reactions* (synchronous paradigm). Hence, determinism and deadlock-freedom are guaranteed by construction. They abstract away all architectural and hardware issues of distributed, embedded systems so that the programmer can concentrate on the functionality. The link to real time and the execution platform is established by a worst-case reaction time (WCRT) analysis, which guarantees that the compiled code is schedulable under given application constraints.

Synchronous languages have been successfully used in avionics, railway and space applications, notably using the SCADE [Sca] tool from Esterel Technologies which provides a DO-178B level A certified compiler.

Hybrid systems. A computer system together with its physical environment makes up a hybrid system: The controller is formalized as a discrete-time transition system and the plant is modeled using (continuous-time) differential equations. *Hybrid automata* [Hen96] are a classical formal model for hybrid systems that enable the modeling of the controller, the plant and their interactions in a single formalism. Simulation tools for hybrid systems, like SIMULINK/STATEFLOW [Sim], are widely used within a model-based design flow, for example, in automotive applications.

While control theorists design continuous-time controllers that perform the desired control task, we have to deal with event- or time-triggered discrete implementations of these controllers. Our focus is clearly on the discrete part of the system: we will target systems where the complexity of the discrete part largely exceeds the complexity of the continuous environment.

Static analysis by abstract interpretation. Abstract interpretation [CC77] is an algorithmic program analysis method, which enables unbounded (time) analysis of infinite state systems with guaranteed termination.

However, since the verification problem is undecidable for general logico-numerical programs, these properties are purchased for the price of over-approximations that make the problem semi-decidable: the analyzer either replies "yes, property proved" or "don't know". Hence, such algorithms cannot find counter-examples and falsify properties.

Numerous academic and industrial tools have been developed within the abstract interpretation framework, *inter alia*, the above cited ASTRÉE and POLYSPACE analyzers.

1.3 Problems and Objectives

This thesis addresses the following aspects of the verification of embedded systems:

(1) *State space explosion problem*: Classical abstract interpretation techniques deal only with numerical variables. In the case of logico-numerical programs, they enumerate first the Boolean states and encode them into program control points. It is well-known from model checking of Boolean systems that explicitly enumerating states becomes intractable already for medium-sized programs.

(2) *Precision loss due to extrapolation*: Numerical abstract interpretation uses an extrapolation operator ("widening") in order to force convergence of the analysis. How-

ever, termination is dearly bought by a hard-to-predict loss of precision.

(3) *Verification of high-level programming languages*: Verification methods are generally based on automata-based formal models. Yet, it is cumbersome to program real systems in such low-level formalisms lacking any programming language concepts.

We detail these three aspects:

State space explosion. We will take into account this challenge by following the approach of Jeannet et al. [JHR99, Jea00, Jea03], who proposed an abstract interpretation method for logico-numerical programs based on an abstract domain that handles symbolically both Boolean and numerical variables, and state space partitioning which enables trading precision for efficiency.

Precision loss due to extrapolation. Various methods have been proposed to stem the precision loss by extrapolation operators in (numerical) abstract interpretation. In this thesis we consider *abstract acceleration*, introduced by Gonnord et al. [GH06, Gon07], which exploits the regularity of the behavior of certain numerical operations in loops of the program to perform a *precise* extrapolation in the abstract domain of convex polyhedra. A second method that we consider is *max-strategy iteration*, proposed by Gawlitza et al. [GS07a, GS07b, Gaw09], which computes the best approximation of the reachable state space in the abstract domain of template polyhedra with the help of mathematical programming.

Our goal is to improve the precision of logico-numerical analyses by adapting the above methods from the numerical to the logico-numerical setting.

Verification of high-level hybrid programming languages. The success of industrial modeling and simulation tools like Simulink [Sim] makes apparent that there is a need for high-level languages in the design of embedded systems, which serve as a common basis for simulation, code generation and verification. However, hybrid system verification is based mostly on the low-level representation of *hybrid automata* [ACH+95].

Our goal is to make abstract interpretation methods amenable to hybrid programming languages by a compilation into hybrid automata that copes with these conceptual discrepancies.

The efficiency and precision of verification methods is hard to evaluate by theoretical means. Therefore, the proposed techniques need to be implemented in order to enable experimental evaluation and comparison:

Tool development. Our goal is to develop a tool that, apart from implementing our methods, enables the integration of available and future abstract interpretation methods and the connection to actual programming languages. This should make it easier to automatize experimental comparisons: for some of the numerous abstract interpretation methods proposed every year, academic prototypes are available, however, it is laborious to compare the efficiency and precision because often benchmarks have to be transformed in the proprietary input formats, and the tool output must be re-parsed in order to automatize comparisons.

1.4 Contributions

The first objective of this thesis is to provide methods that allow us to improve the precision while dealing with the state space explosion problem. To this end, we want to make numerical analysis methods, which are known to improve the precision, applicable in the logico-numerical context. We will consider two such methods: abstract acceleration and max-strategy iteration.

Verification of numerical programs. Before tackling logico-numerical programs we have to settle a minor deficiency of abstract acceleration: it is not yet ready to handle numerical input variables, which are, *e.g.*, used to model inputs from sensors. Thus, we *extend abstract acceleration to numerical inputs* (published in [SJ10, SJ12a]).

Moreover, we provide an *extension of abstract acceleration to backward analysis*, *i.e.*, co-reachability analysis [SJ12a]. As a side effect, these contributions include detailed proofs for the abstract acceleration methods originally proposed by Gonnord et al.

Furthermore, we *revisit the notion of linear accelerability* and provide a more general characterization of linearly accelerable transitions.

Verification of logico-numerical programs. Now, we are prepared to *apply abstract acceleration to logico-numerical programs* [SJ11], a method based on decoupling Boolean and numerical transition functions and a partitioning technique for detecting "numerical modes", *i.e.*, states with the same numerical behavior. Moreover, we provide experimental results that demonstrate the efficiency and precision of the approach.

As a second method, we present an algorithm for *applying max-strategy iteration to logico-numerical programs* [SS13]. Our experiments show that this is a promising approach.

The remaining contributions to theory concern the translation of high-level hybrid languages and their verification:

Modeling and verification of hybrid systems. We want to verify programs written in high-level programming languages for hybrid systems, *e.g.*, Zelus [BBCP11a, BBCP11b]. To this end, we present a *translation of a hybrid data-flow language to logico-numerical hybrid automata* [SJ12b], which addresses the problem of zero-crossings – an issue left open in existing translation attempts which raises tricky semantical issues. Moreover, the target formalism of the translation, logico-numerical hybrid automata, takes into account the state space explosion problem.

At last, we describe how to *extend hybrid system analysis methods to logico-numerical hybrid automata*. We present partitioning methods for detecting continuous modes and propose principles for adapting existing numerical hybrid system analysis methods to logico-numerical hybrid systems.

Implementation. On the practical side, we have developed *a verification tool*, ReaVer[2] (§14) based on the Apron library [JM09], a widely accepted API for numerical abstract domains. Besides the methods we propose, it implements several existing techniques and it is easily extensible. Besides a low-level input format it supports a subset of the Zelus language.

[2] `http://pop-art.inrialpes.fr/people/schramme/reaver/`

1.5 Outline

The organisation of the thesis is illustrated in Fig. 1.2.

We start with an introduction to formal program models and analysis methods: §2 explains how we formally represent discrete *programs* and specify the *properties* we want to verify. §3 gives a detailed presentation of the state of the art in *reachability analysis by abstract interpretation*.

The following three parts of the thesis deal with our contributions in the three areas of (I) numerical programs, (II) logico-numerical programs and (III) hybrid systems.

Part I deals with numerical program analysis: We first recall the concepts of acceleration and abstract acceleration §4 before we present our contributions w.r.t. extending (§5) and generalizing (§6) abstract acceleration.

Part II is dedicated to logico-numerical analysis of discrete programs: We start with a detailed presentation of the state of the art in *logico-numerical program analysis* in §7. Then, §8 and §9 explain our contributions concerning *logico-numerical abstract acceleration* and *logico-numerical max-strategy iteration* respectively.

Part III deals with the modeling and analysis of hybrid systems: We give first an introduction to *hybrid system modeling* and the Zelus language (§10). §11 details our translation from Zelus to logico-numerical hybrid automata. Then, we give an overview of the state of the art in *hybrid system verification* (§12). Finally, §13 presents some methods for partitioning and *analyzing logico-numerical hybrid automata*.

The last two chapters are dedicated to the *implementation* of our tool ReaVer (§14) and to the *conclusions and perspectives* (§15).

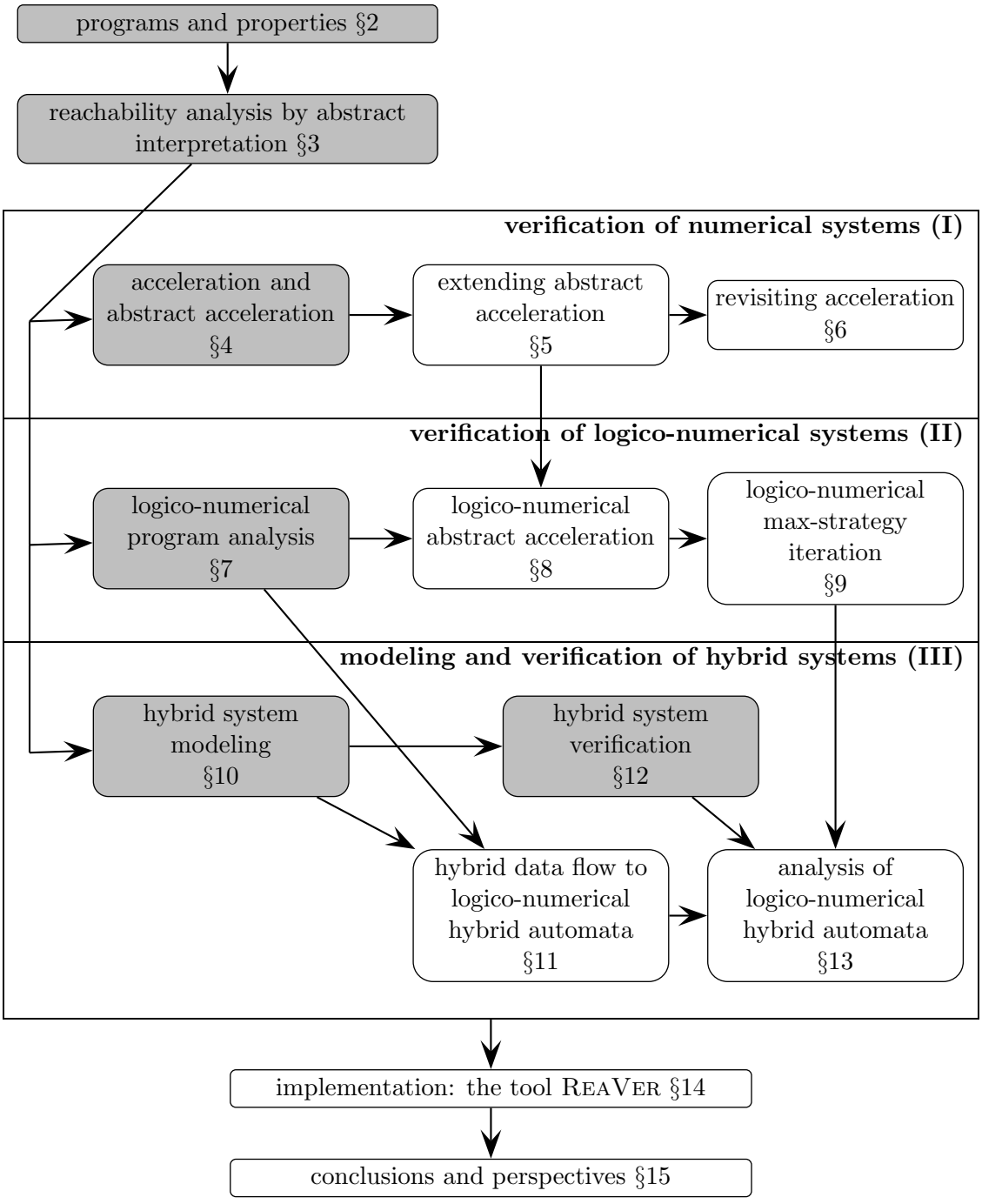


Figure 1.2: Organization of the thesis: state of the art (shaded) and contributions.

Chapter 2

Programs and Properties

Safety-critical embedded systems consist of reactive programs in interaction with their environment. Reactive programs can be modeled as *discrete transition systems* (§2.1).

Synchronous languages (§2.2) like LUSTRE [CPHP87] and LUCID SYNCHRONE [CP96] are high-level programming languages for reactive systems, which combine data-flow and synchronous paradigms and which can be compiled to discrete transition systems.

Safety properties express statements about a system of the form "something bad never happens". In the context of synchronous languages such properties can be specified by *synchronous observers* (§2.3).

2.1 Program Models

We model reactive programs as *discrete transition systems* (§2.1.1), possibly considering their *control flow graphs* (CFGs, §2.1.2).

2.1.1 Discrete Transition Systems

We will model reactive programs as discrete (input/output) transition systems (*discrete dynamical systems*):

Definition 2.1 (Discrete transition system) *A discrete transition system is defined by* $\langle \Sigma, \Upsilon, R, \mathcal{I} \rangle$ *where*

- Σ *and* Υ *are the state and input spaces,*
- $R \subseteq \Sigma \times \Upsilon \times \Sigma$ *is the transition relation and*
- $\mathcal{I} \subseteq \Sigma$ *is the set of initial states.*

W.l.o.g. we can assume that the output of a system equals its state.

Definition 2.2 (Semantics) *An* execution *of such a system is a sequence*

$$\boldsymbol{s}_0 \xrightarrow{\boldsymbol{i}_0} \boldsymbol{s}_1 \xrightarrow{\boldsymbol{i}_1} \ldots \boldsymbol{s}_k \xrightarrow{\boldsymbol{i}_k} \ldots$$

such that $\boldsymbol{s}_0 \in \mathcal{I}$ *and* $\forall k \geqslant 0 : (\boldsymbol{s}_k, \boldsymbol{i}_k, \boldsymbol{s}_{k+1}) \in R$.

A discrete transition system is *deterministic* iff the transition relation is a function, *i.e.*, $R \subseteq (\Sigma \times \Upsilon \to \Sigma)$, in other words: iff for each initial state and sequence of input valuations there is a unique execution trace.

Numerical and logico-numerical transition systems. We will mostly deal with deterministic transition systems over state and input spaces Σ and Υ represented as $\begin{cases} \mathcal{I}(\boldsymbol{s}) \\ \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \end{cases}$ where $\boldsymbol{s}$ and $\boldsymbol{i}$ are the vectors of state and input variables respectively, and $\boldsymbol{f}$ is the vector of transition functions. The transition relation R is obtained by $\{(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}') \mid \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i})\}$.

Depending on the state and input spaces we can instantiate different types of discrete transition systems: *Numerical transition systems* with $\Sigma = \mathbb{R}^n$ and $\Upsilon = \mathbb{R}^m$ are finite-difference equations. We denote the state and input vectors $\boldsymbol{x}$ and $\boldsymbol{\xi}$ respectively.

Example 2.1 (Numerical transition system) *A simple example for a numerical transition system with $\Sigma = \mathbb{R}$ and $\Upsilon = \mathbb{R}$ is the program that memorizes the maximum positive integer in the input sequence. On the right-hand side we give the beginning of a possible execution:* $\begin{cases} \mathcal{I} = (x = 0) \\ x' = \begin{cases} \xi & \text{if } \xi > x \\ x & \text{else} \end{cases} \end{cases}$

k	0	1	2	3	4	...
ξ	2	−3	4	1	4	...
x	0	2	2	4	4	...

Boolean transition systems with $\Sigma = \mathbb{B}^n$ and $\Upsilon = \mathbb{B}^m$ are Mealy machines for instance.

Logico-numerical transition systems have state and input spaces of the form $\mathbb{B}^n \times \mathcal{N}^m$, where $\mathcal{N}$ can be any cartesian product of numerical sets; usually we assume $\mathcal{N} = \mathbb{R}^m$. We denote the state and input vectors consisting of Boolean and numerical components $(\boldsymbol{b}, \boldsymbol{x})$ and $(\boldsymbol{\beta}, \boldsymbol{\xi})$ respectively. We will provide a detailed presentation of logico-numerical programs in §7.

Remark 2.1 (Number representations) *Discrete transition systems are idealized versions of actual programs assuming unbounded integers instead of machine integers and real numbers instead of floating point numbers. In practice, verification tools often represent machine integers as arbitrary-sized integers and floating point numbers as arbitrary precision rationals resulting in sets of the form $\mathbb{Z}^p \times \mathbb{Q}^q$.*

Remark 2.2 (Continuous dynamical systems) *Continuous dynamical systems have exactly the same form as discrete numerical dynamical systems, except that time is continuous and the dynamics is specified by differential equations instead of difference equations:* $\forall t \geqslant 0 : \begin{cases} \boldsymbol{x}(0) \in \mathcal{I} \\ \dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{\xi}(t)) \end{cases}$

2.1.2 Control Flow Graphs

The transition relation of a discrete transition system has a complex structure, *e.g.*, involving conditionals (if-then-else), which makes it hard to understand for humans. In program analysis it can be favorable to make this control structure explicit by encoding it into a control flow graph:

Definition 2.3 (Control flow graph) *A control flow graph (CFG) $\langle \Sigma, \Upsilon, L, \rightsquigarrow, \Sigma^0 \rangle$ is a directed graph where*

- *Σ and Υ are the state and input spaces,*
- *L is the set of locations (the vertices of the graph),*
- *$\rightsquigarrow \subseteq L \times \mathcal{R} \times L$ defines arcs (the edges of the graph) between the locations. The arcs are labeled with a transition relation $R \in \mathcal{R} = (\Sigma \times \Upsilon \times \Sigma)$.*

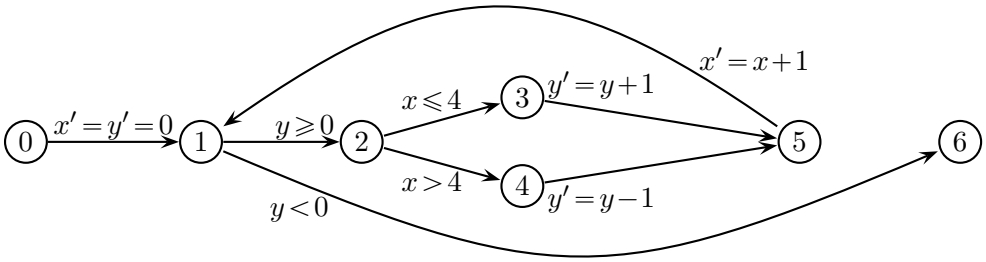


Figure 2.1: CFG for the program in Ex. 2.2.

- $\Sigma^0 : L \to \Sigma$ *defines for each location the set of initial states.*

Definition 2.4 (Semantics) *An* execution *of a CFG is a sequence*

$$(\ell_0, \boldsymbol{s}_0) \xrightarrow{\boldsymbol{i}_0} (\ell_1, \boldsymbol{s}_1) \xrightarrow{\boldsymbol{i}_1} \ldots (\ell_k, \boldsymbol{s}_k) \xrightarrow{\boldsymbol{i}_k} \ldots$$

such that for any $k \geqslant 0 : \exists (\ell_k, R, \ell_{k+1}) \in \rightsquigarrow : (\boldsymbol{s}_k, \boldsymbol{i}_k, \boldsymbol{s_{k+1}}) \in R$.

Imperative programs, *e.g.*, C programs, can be translated directly into a CFG representation by associating control points with programming constructs as if-then-else or while:

Example 2.2 (CFG of an imperative program) *The CFG of the following C program (cf. [GR07]) is depicted in Fig. 2.1. The numbers in double parentheses are the control points.*

```
((0)) x=0; y=0;
((1)) while(y>=0) {
    ((2)) if(x<=4) ((3)) y++;
          else ((4)) y--;
    ((5)) x++; } ((6))
```

For any discrete transition system $\langle \Sigma, \Upsilon, R, \mathcal{I} \rangle$ we can easily obtain the trivial control flow graph representation $\langle \Sigma, \Upsilon, \{\ell_0\}, \{(\ell_0, R, \ell_0)\}, \{\ell_0 \to \mathcal{I}\} \rangle$.

In Ex. 2.2, a detailed CFG is extracted from the imperative program source by syntactical criteria. In §7.3 we will discuss *state space partitioning* techniques for generating CFGs of arbitrary granularity from discrete transition systems.

Remark 2.3 (Hybrid automata) *Hybrid automata [Hen96] (see §10.2) are CFGs where, in addition, locations are labeled by differential equations. That way, discrete and continuous dynamical systems are modeled in a single formalism. An execution of such a hybrid system is an alternation of discrete transitions between locations and continuous evolutions while staying in a location.*

2.2 Synchronous Languages

Synchronous languages [Hal93b, Hal98, BCE+03] were developed for programming reactive systems. These languages apply the synchronous principle of digital hardware

circuits to software: parallel computation steps are executed in lockstep. Such steps (called *reactions*) are triggered by a (logical) *clock*. This computation model guarantees deterministic concurrency, *i.e.*, synchronous programs can be composed in parallel by their synchronous product [HLR93] (no interleavings as in the asynchronous case).

A large number of synchronous languages have been developed with different flavors. Generally, we distinguish languages that have an imperative language syntax like ESTEREL [BC85] from those with a data-flow syntax like LUSTRE [CPHP87]. We will focus on the verification of synchronous *data-flow* languages, but the methods apply also to imperative programs after having compiled them into a data-flow representation.

2.2.1 Lustre

LUSTRE was designed as a synchronous version of the data-flow language LUCID [AW85] for programming real-time systems (hence the name (in French) LUcid Synchrone Temps RÉel) [CPHP87]. By combining data-flow and synchronous paradigms, it has a semantics that is surprisingly simple and easy to understand.

In LUSTRE variables represent streams of values. Operators, *e.g.*, `if then else` or `+`, are lifted to streams and applied point-wise to the elements of their operand streams. The previous value operator `pre` delays a stream by one clock tick:

$$(\texttt{pre}\ s)(k) = \begin{cases} \mathsf{uninitialized} & \text{for } k=0 \\ s(k-1) & \text{for } k>0 \end{cases}$$

The initialization operator `->` is used to initialize a stream:

$$(s^0\ \texttt{->}\ s)(k) = \begin{cases} s^0(0) & \text{for } k=0 \\ s(k) & \text{for } k>0 \end{cases}$$

A program is structured in `nodes`, *i.e.*, program blocks that have inputs and outputs, and encapsulate their internal state. Nodes can be called (instantiated) by other nodes; each instantiation has its own internal state. A node contains exactly one equation for each state and output variable s_j of the form $s_j = f_j(\boldsymbol{s}, \boldsymbol{i})$.

Example 2.3 (Lustre program) *The following program outputs at each clock tick the maximum value encountered so far in its integer input stream:*

```
node max(xi:int) returns (x:int);
let
    x = xi -> if xi > pre x then xi else pre x;
tel
```

Equations like `x = y + 1; y = 2*x` are forbidden because of the *instantaneous causality cycle* they induce: we need to know `y` in order to compute `x` and vice versa. The language semantics does not resolve such cycles by computing fixed points, but it requires each cyclic dependency to be cut by a delay operator `pre`, otherwise the program is rejected by the compiler.

Accessing an uninitialized value of a stream, like in the equation `x = pre x + 1`, leads to a runtime error.

Compilation to a discrete transition system. We can easily obtain the form of a discrete transition system (§2.1.1) by

– inlining the instantiated nodes (involves a renaming of the state variables in order to make them unique);
– adding a finite number of variables $\boldsymbol{p}$ (called "memories") for memorizing the delayed values and their initialization, and replacing the delay operators by these variables, *e.g.*, `y=2*(0 -> (pre (x+1)))` becomes $y = 2p$ and we have the transition function $p' = x+1$ with the initial value $p = 0$; and
– replacing the variables $\boldsymbol{s}$ occurring on the left-hand sides of the equations by the left-hand side of the equations defining $\boldsymbol{s}$. For example, `z=y+1` with the above definition of y becomes $z = 2 * p + 1$. This rewriting terminates because dependencies are acyclic.

The result consists of transition functions for the state variables $\boldsymbol{p}$ of the discrete transition system, their initial values, and the equations defining the outputs (which we are not interested in).

Example 2.4 (Lustre to discrete transition system) *For the program computing the Fibonacci numbers*

```
node fibonacci (dummy:bool) returns (x:int);
let
    x = 1 -> pre(x + (0 -> pre x))
tel
```

we obtain the discrete transition system

$$\mathcal{I} = (p_1 = 1 \wedge p_2 = 0) \qquad \begin{cases} p_1' &= p_1 + p_2 \\ p_2' &= p_1 \end{cases}$$

where the output x of the node has the value of p_1.

We have seen in §2.1.2 that the trivial CFG of such a program consists of a single location with a self-loop.

2.2.2 Lucid Synchrone

Lucid Synchrone [CP96, Pou02, CGHP04, CPP05, Pou06, CHP08] combines the synchronous data-flow language Lustre with the features of a functional language like ML [MTH90]. From the former it inherits the data-flow operators, from the latter, *inter alia*, the syntax and the type inference.

Lucid Synchrone brings a lot of new features to synchronous languages – we will only give a small selection here. For an exhaustive presentation we refer to the user manual [Pou06]. All the features presented in the following are mere syntactical sugar in order to lift programming to higher level. Such programs can still be reduced to a discrete dynamical system as explained above.

The initialized delay operator `fby` initializes a stream by the first value of its first operand and continues with the second operand delayed by one clock tick. Thus, s^0 `fby` s is equivalent to s^0 `-> pre` s.

Unlike Lustre V4, the Lucid Synchrone compiler features an initialization analysis that checks if all accessed stream elements have actually been initialized.

While Lustre V4 offers only a sampling (`when`) and a projection (`current`) operator for dealing with multi-clocked systems, Lucid Synchrone provides also the oversampling operator `merge` for combining streams with complementary clocks. Furthermore it comes with an automatic clock inference based on a dependent type system.

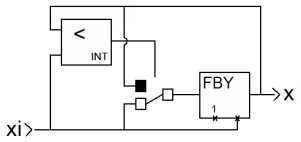


Figure 2.2: SCADE diagram of the program in Ex. 2.3.

Hierarchical automata. Since programming state machines in a data-flow language is tedious, LUCID SYNCHRONE includes also hierarchical automata. They consist of a list of locations with associated equations and a list of outgoing transitions:

Example 2.5 (Automata) *We give an example (cf. [Pou06]) of an automaton with two locations; the initial location is implicitly the first location declared:*

```
let node triangle () = x where rec
  automaton
    | Up ->
      do
        x = 0 -> last x + 1
      until (x >= 9) continue Down
    | Down ->
      do
        x = last x - 1
      until (x <= 1) continue Up
  end
```

There are transitions with weak preemption (`until`) where the transition guard is checked after executing the equations associated with the location, and transitions with strong preemption (`unless`) where the transition can be taken before executing the equations at all. The target locations of a transition can be entered by reset (`then`), *i.e.*, as if it was the initial instant, or by history (`continue`), *i.e.*, with the values the streams had the last time the location was visited. Communication between the locations of an automaton is handled via shared variables (`last x`). Moreover, automata can be nested (hierarchy).

Automata are translated to data-flow using multi-clocking: equations associated to locations are subsampled by the clock taking the values of the respective location labels and the streams for the shared variables are then merged over this multi-valued clock.

Scade. The commercial tool SCADE [Sca] (Software Critical Application Development Environment) extends LUSTRE with several features of LUCID SYNCHRONE, like `fby`, `merge` and automata. It provides a graphical, block diagram representation, *i.e.*, data flow graphs, of programs. Fig. 2.2 shows the SCADE diagram of the program of Ex. 2.3.

2.3 Properties and Observers

The goal of formal verification is to prove the compliance of a program to its specification. The specification consists of a set of properties.

Safety and liveness properties. One distinguishes two types of properties [Lam77]:

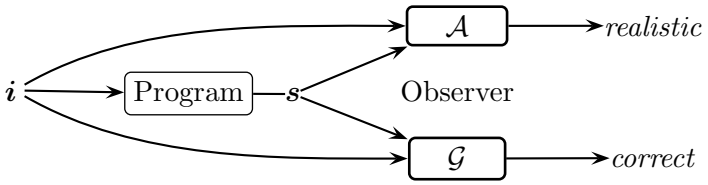


Figure 2.3: Verification of a synchronous program with the help of a synchronous observer

– *Safety properties* state that nothing bad happens. Many important properties in practice fall into this category: state properties (*invariants*) such as the absence of runtime errors (*e.g.*, overflows and division-by-zero) and mutual exclusion, and trace properties like deadlock freedom or guarantees w.r.t. response times and deadlines. The violation of safety properties can be shown by a finite execution trace (counterexample).
– *Liveness properties* state that something good eventually happens, *i.e.*, a desirable action is eventually executed. Examples for such properties are termination or fair choice. Such properties cannot be violated by finite execution traces.

We will consider only safety properties.

An example for a specification language of properties is *linear temporal logic* (LTL) [Pnu77]. In LTL, invariants are of the form $\mathbf{G}\mathcal{G}$ meaning that a "predicate $\mathcal{G}$ holds globally": for all execution traces $\boldsymbol{s}_0 \to \boldsymbol{s}_1 \to \ldots$ we have $\forall k \geqslant 0 : \mathcal{G}(\boldsymbol{s}_k)$.

Synchronous observers. In the context of synchronous languages a safety property can be specified with the help of a synchronous observer [HLR93], *i.e.*, a program written in the same language, synchronously composed in parallel with the program under verification and with a Boolean output which is true iff the current prefix of the execution satisfies the property. A synchronous observer allows any safety property to be expressed as the invariant "the output of the observer is always true".

In practice, we have to take into account hypotheses about the environment, *i.e.*, which input values (in dependency of the current system state) are considered realistic. Thus, one actually needs two observers: one for the assumption (or *assertion*) about the environment $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ and one to specify the program behavior considered correct, *i.e.*, the actual property $\mathcal{G}(\boldsymbol{s})$ which shall be guaranteed if the assertion is satisfied: $\mathbf{G}\mathcal{A} \Rightarrow \mathbf{G}\mathcal{G}$. Fig. 2.3 illustrates this verification schema.

Example 2.6 (Observer) LUSTRE *provides an* `assert` *statement which can be used to specify* $\mathcal{A}$*. For* $\mathcal{G}$*, we define a Boolean output variable* `ok`*.*

```
node program(xi:int) returns (x:int);
let
  x = 1 -> pre x*xi;
tel
node property(x,xi:int) returns (ok:bool);
var t:int;
let
  t = 1 -> pre t + 1;
  ok = true -> pre ok and ((t<=8) => (1<=x and x<=16*t));
tel
```

```
node system(xi:int) returns (ok:bool);
var x:int;
let
  x = program(xi);
  assert 1<=xi and xi<=2;
  ok = property(x,xi);
tel
```

A static analyzer actually truncates the traces at the point where $\mathcal{A}$ ceases to hold. If $\mathcal{G}$ holds along all these (partial) traces, the property is satisfied: $\mathbf{G}(\mathcal{A}\wedge\mathcal{G})\vee(\mathcal{A}\wedge\mathcal{G})\mathbf{U}\neg\mathcal{A}$. This formula is equivalent to $\mathbf{G}\mathcal{A}\Rightarrow\mathbf{G}\mathcal{G}$ under the condition that the assertion observer does not block the program under verification [HLR93], *i.e.*, for every $\boldsymbol{s}_k$ in a program execution $\exists\boldsymbol{s}_{k+1}\exists\boldsymbol{i} : R(\boldsymbol{s}_k,\boldsymbol{i},\boldsymbol{s}_{k+1})\wedge\mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i})$.
To put everything together, a discrete transition system with observers has the form

$$\left\{\begin{array}{l}\mathcal{I}(\boldsymbol{s})\\ \mathcal{A}(\boldsymbol{s},\boldsymbol{i})\rightarrow\boldsymbol{s}'=\boldsymbol{f}(\boldsymbol{s},\boldsymbol{i})\\ \mathcal{G}(\boldsymbol{s})\end{array}\right\}\text{ with }\mathcal{A}\subseteq(\Sigma\rightarrow\wp(\Upsilon))\text{ and }\mathcal{G}\subseteq\Sigma.$$

Property specification with observers can be used for all synchronous representations of discrete and hybrid systems (see Rem. 2.3 and §10).

Remark 2.4 (Assume/guarantee) *Another possible observer semantics comes from assume/guarantee reasoning:* $\mathbf{G}(\mathcal{A}\Rightarrow\mathcal{G})$, i.e., *the property is considered to be satisfied if for all configurations the satisfaction of* $\mathcal{A}$ *implies the satisfaction of* $\mathcal{G}$. *We have the following relationship w.r.t. the above semantics:* $\mathbf{G}(\mathcal{A}\Rightarrow\mathcal{G})\Longrightarrow(\mathbf{G}\mathcal{A}\Rightarrow\mathbf{G}\mathcal{G})$.

Chapter 3

Reachability Analysis by Abstract Interpretation

Static analysis is the umbrella term for automatic program analysis methods that infer properties about a program without actually executing it.

Reachability analysis (§3.1) is a static analysis that computes the states reachable by all possible program executions. Hence, it enables the verification of invariance properties. We consider reachability analyses using *abstract interpretation* [CC76, CC77, Cou81, CC92a], an analysis framework based on lattice theory (§3.2) that consists of two ingredients:

– an *abstract domain* (§3.3), *i.e.*, an abstract state space to which concrete program states are mapped; also, the concrete semantics of the program is mapped to an abstract semantics in the abstract domain; and
– a method for resolving the abstract semantic *fixed point* equations associated to the program (§3.4).

Abstract interpretation methods are guaranteed to terminate for any program, but at the price of an approximate analysis result. There are several commercial and academic static analysis libraries and tools based on this framework (§3.5).

3.1 Reachability Analysis

Reachability analysis aims at computing the reachable state space of a discrete transition system $\langle \Sigma, \Upsilon, R, \mathcal{I} \rangle$.

Post- and pre-condition (or image and pre-image) operators define the sets of states reachable from a given set of states by taking a transition in forward or backward direction respectively:

Definition 3.1 (Pre- and post-condition operators)

Post-condition operator: $\mathsf{post}(S) = \{\boldsymbol{s}' \mid \exists \boldsymbol{s} \in S, \exists \boldsymbol{i} \in \Upsilon : (\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}') \in R\},\ S \subseteq \Sigma$

Pre-condition operator: $\mathsf{pre}(S) = \{\boldsymbol{s} \mid \exists \boldsymbol{s}' \in S, \exists \boldsymbol{i} \in \Upsilon : (\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}') \in R\},\ S \subseteq \Sigma$

The set of (co-)reachable states, *i.e.*, the (co-)reachable state space of a discrete transition system, is defined by the reflexive and transitive closure of these operators:

Definition 3.2 ((Co-)reachable state space)

Reachable state space: $\mathsf{reach}(\mathcal{I}) = \mathsf{post}^*(\mathcal{I}) = \bigcup_{k\geqslant 0} \mathsf{post}^k(\mathcal{I})$
Co-reachable state space: $\mathsf{co\text{-}reach}(\mathcal{I}) = \mathsf{pre}^*(\mathcal{I}) = \bigcup_{k\geqslant 0} \mathsf{pre}^k(\mathcal{I})$

$$\textit{with} \begin{cases} \mathsf{post}^0(S) &= S \\ \mathsf{post}^{k+1}(S) &= \mathsf{post} \circ \mathsf{post}^k(S) \end{cases} \quad \textit{and analogously for } \mathsf{pre}^k.$$

An invariance property $\mathcal{G}$ (§2.3) can be checked by either a (forward) reachability analysis or a co-reachability analysis (backward analysis):
– $\mathsf{reach}(\mathcal{I}) \subseteq \mathcal{G}$, or $\mathsf{reach}(\mathcal{I}) \cap \mathcal{E} = \varnothing$ with the error states $\mathcal{E} = \Sigma - \mathcal{G}$
– $\mathsf{co\text{-}reach}(\mathcal{E}) \cap \mathcal{I} = \varnothing$

Finite and infinite-state systems. If Σ is finite the computation of $\mathsf{reach}(\mathcal{I})$ is guaranteed to terminate in a finite number of computation steps, whereas in systems with an infinite state space termination is generally not guaranteed. As a fundamental consequence, reachability analysis in finite-state systems is *decidable*, whereas in infinite-state systems, *e.g.*, systems with $\Sigma = \mathbb{Z}^n$, it is *undecidable* in general: a system with two counters with increment, decrement and test-to-zero (two-counter machine) is known to be Turing-complete [Min67].

However, there are some important classes of infinite-state systems with restricted transition relations R such that reachability analysis becomes decidable, *e.g.*, timed automata [AD94]. We will deal with general infinite state systems.

Analysis approaches. *Model checking* subsumes methods for checking properties of finite state systems or systems abstracted to finite state systems. Model checking is successfully applied in digital circuit verification, *i.e.*, on Boolean transition systems, for which the reachability problem is an NP-complete problem [Coo71]. Model checking methods rely either on enumeration, symbolic representations like BDDs, or SAT-solving. *Bounded model checking* [CBRZ01] considers only traces of bounded length. It is an efficient method to find bugs, but it cannot prove properties.

An infinite state system can be model-checked by abstracting it (by an appropriate simulation relation) to a finite system (also called "quotient system" [Hen96]) such that the property to be verified is preserved [BBLS92, CGL94]. For example, *predicate abstraction* [GS97, FQ02] considers a finite set of predicates in order to generate a finite state system, which is a (conservative, safe) over-approximation of the original system: this means that each execution trace of the original system is an execution trace of its abstraction, but conversely, a trace of the abstracted system may not be an actual trace of the original system. Hence, if the property is verified on the abstraction, it is also true on the original system. However, the additional behavior of the abstraction may disprove the property although it is true. Such spurious counterexamples can be used to derive additional predicates for refining the abstraction (*counterexample-guided abstraction refinement* [CGJ$^+$00]). Yet, this refinement process is not guaranteed to terminate in general.

Other methods for infinite state systems make use of the power of modern SMT-solvers: *k-induction* for instance [SSS00, HT08, DHKR11], unrolls the transition relation k times and proves by induction that the property also holds for $k+1$. This method terminates if there exists a bounded k that enables the proof of the property.

Abstract interpretation is a framework for over-approximating the concrete states and semantics by abstract ones. Termination is guaranteed by an extrapolation operator. Similarly to finite state abstraction in model checking, the over-approximation

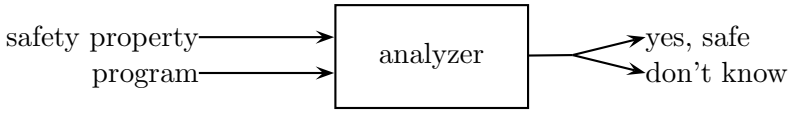


Figure 3.1: Abstract interpreter for verifying safety properties.

adds behaviors, and thus the method can prove properties, but it cannot falsify them in general. Fig. 3.1 shows the scheme of an abstract interpreter for verifying safety properties: either the property is proved ("yes, safe") or the result remains *inconclusive* ("don't know").

3.2 Lattice Theory and the Principle of Abstract Interpretation

Abstract domains (§3.3) are often *complete lattices*, *i.e.*, algebras over partially ordered sets. We recall in this section the fundamental notions of lattice theory, which are the basis for the soundness of over-approximations of reachable state sets computed by abstract interpretation methods. For further details and proofs we refer to textbooks on lattice theory [DP90] and static analysis [NNH05].

From partial orders to complete lattices

Definition 3.3 (Partial order) *A* partial order $\langle S, \sqsubseteq \rangle$ *is a set* S *equipped with a reflexive, transitive and anti-symmetric binary relation* $\sqsubseteq$*:* $S \times S$*.*

For example, the powerset of integers with the set inclusion $\langle \wp(\mathbb{Z}), \subseteq \rangle$ is a partial order. Partial orders can be visualized with the help of Hasse diagrams.

Definition 3.4 (Upper and lower bounds) *Let* $S' \subseteq S$*:*
- *Upper bound* $s \in S$*:* $\forall s' \in S' : s' \sqsubseteq s$
- *Lower bound* $s \in S$*:* $\forall s' \in S' : s' \sqsupseteq s$
- *Least upper bound (also called* join *or* union *operator)* $s = \bigsqcup S'$*: for all upper bounds* $s' \in S'$*:* $s \sqsubseteq s'$
- *Greatest lower bound (also called* meet *or* intersection *operator)* $s = \bigsqcap S'$*: for all lower bounds* $s' \in S'$*:* $s \sqsupseteq s'$

For the powerset of integers $\langle \wp(\mathbb{Z}), \subseteq \rangle$, we can define these operations as the usual intersection $\cap$ and union $\cup$ for sets. For example, take $S' = \{\{1,2,3\}, \{2,4,5\}, \{2,5\}\}$, then $\bigsqcup S' = \{1,2,3\} \sqcup \{2,4,5\} \sqcup \{2,5\} = \{1,2,3,4,5\}$, and $\bigsqcap S' = \{2\}$.

Abstract domains are often complete lattices:

Definition 3.5 (Complete lattice) *A* complete lattice *is an algebraic structure* $\langle S, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ *with a partially ordered set* S *such that*
- *all subsets have least upper and greatest lower bounds,*
- *the least element* $\bot = \bigsqcup \varnothing = \bigsqcap S$*, and*
- *the greatest element* $\top = \bigsqcap \varnothing = \bigsqcup S$*.*

The powerset of integers is a complete lattice with $\bot = \varnothing$ and $\top = \mathbb{Z}$.

Definition 3.6 (Ascending and descending chains)
– *A* chain *is a totally ordered subset* $S' \subseteq S$, i.e., $\forall s_1, s_2 \in S' : s_1 \sqsubseteq s_2 \vee s_1 \sqsupseteq s_2$.
– *A sequence* $(s_n)_{n\in\mathbb{N}}$ *with* $s_n \in S$ *is called*
  – *an* ascending chain *iff* $m \leqslant n \Rightarrow s_m \sqsubseteq s_n$,
  – *a* descending chain *iff* $m \leqslant n \Rightarrow s_m \sqsupseteq s_n$.

The *height* of a lattice is defined as the length of its longest chain. A lattice has *finite height* iff all of its chains have finite length, otherwise it has *infinite height*. The powerset of integers lattice, for example, has infinite height.

Functions over lattices and their fixed points. The following definitions are needed to state the theorems that allow us to solve fixed point equations over complete lattices.

Definition 3.7 (Monotonic and continuous functions) *Let* $\langle X, \sqsubseteq_X\rangle$ *and* $\langle Y, \sqsubseteq_Y\rangle$ *be partial orders and* $f : X \to Y$ *a function, then* f *is*
– monotonic *iff* $\forall x_1, x_2 \in X : x_1 \sqsubseteq_X x_2 \Longrightarrow f(x_1) \sqsubseteq_Y f(x_2)$
– semi-$\sqcup$-continuous *iff* f *is monotonic and preserves the upper bounds of ascending chains:*
  – *for each ascending chain* C *of* X: $\bigsqcup_Y\{f(X) \mid X \in C\} = f(\bigsqcup_X C)$, *and*
– semi-$\sqcap$-continuous *iff* f *is monotonic and preserves the lower bounds of descending chains:*
  – *for each descending chain* C *of* X: $\bigsqcap_Y\{f(X) \mid X \in C\} = f(\bigsqcap_X C)$

Definition 3.8 (Least and greatest fixed point) *Let* $\langle S, \sqsubseteq\rangle$ *be a partial order and* $f : S \to S$ *a function:*
– *The elements* $s' \in S$ *satisfying* $f(s') \sqsubseteq s'$ *are called the* post-fixed points *of* f.
– *The elements* $s' \in S$ *satisfying* $f(s') \sqsupseteq s'$ *are called the* pre-fixed points *of* f.
– *The elements* $s \in S$ *satisfying* $s = f(s)$ *are called the* fixed points *of* f.
– $s = lfp(f)$ *is the* least fixed point *of* f *iff* $\forall s' \in S : f(s') \sqsubseteq s' \Rightarrow s \sqsubseteq s'$.
– $s = gfp(f)$ *is the* greatest fixed point *of* f *iff* $\forall s' \in S : f(s') \sqsupseteq s' \Rightarrow s \sqsupseteq s'$.

The theorem of Knaster-Tarski [Tar55] characterizes the set of fixed points of f:

Theorem 3.1 (Knaster-Tarski) *Let* S *be a complete lattice and* f *a monotonic function, then the set of fixed points forms a (non-empty) complete lattice:*
– $lfp(f) = \bigsqcap\{S \mid f(S) \sqsubseteq S\}$
– $gfp(f) = \bigsqcup\{S \mid S \sqsubseteq f(S)\}$

Finally, the theorem of Kleene [Kle52] suggests an iterative method for computing fixed points (*Kleene iteration*, see §3.4.1):

Theorem 3.2 (Kleene) *Let* S *be a complete lattice and* f *a semi-*$\sqcup$*-continuous (resp. semi-*$\sqcap$*-continuous) function* f*, then*
– $lfp(f) = \bigsqcup_{k\in\mathbb{N}^{\geqslant 0}} f^k(\bot)$ *(ascending Kleene iteration)*
– $gfp(f) = \bigsqcap_{k\in\mathbb{N}^{\geqslant 0}} f^k(\top)$ *(descending Kleene iteration)*

We have the following relationships, which are illustrated in Fig. 3.2:

$$\bot \sqsubseteq f^k(\bot) \sqsubseteq \bigsqcup_k f^k(\bot) \sqsubseteq \begin{array}{l} lfp(f) \\ gfp(f) \end{array} \sqsubseteq \bigsqcap_k f^k(\top) \sqsubseteq f^k(\top) \sqsubseteq \top$$

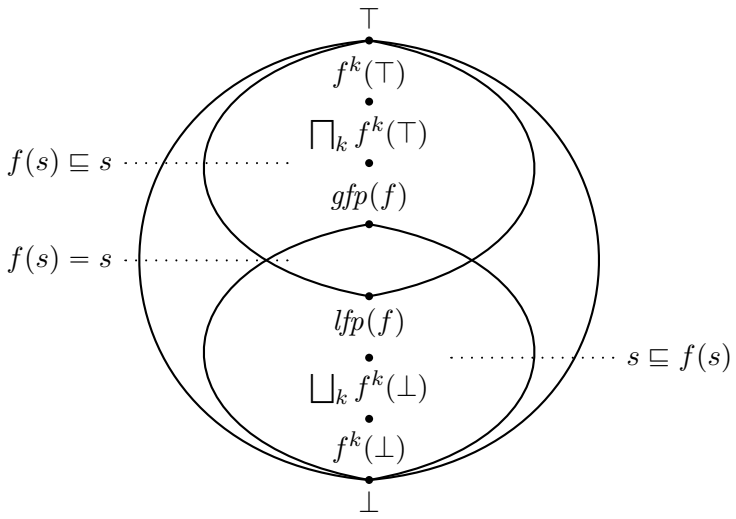


Figure 3.2: Fixed points of f

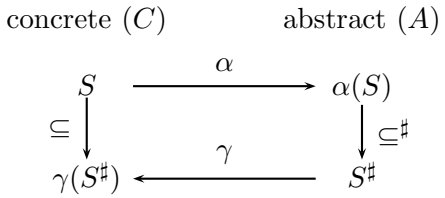


Figure 3.3: Galois connection

Principle of reachability analysis using abstract interpretation

In abstract interpretation, sets of concrete program states in $S \in C = \wp(\Sigma)$ are abstracted by abstract values $S^\sharp$ from an abstract domain A. Their relationship is formalized by the notion of a Galois connection (illustrated in Fig. 3.3):

Definition 3.9 (Galois connection) *Let $\langle C, \sqsubseteq \rangle$ and $\langle A, \sqsubseteq^\sharp \rangle$ be partial orders, and $\alpha : C \to A$ and $\gamma : A \to C$ functions, then*

$$C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \text{ is a Galois connection } \textit{iff } \forall S \in C, S^\sharp \in A : \alpha(S) \sqsubseteq^\sharp S^\sharp \iff S \sqsubseteq \gamma(S^\sharp).$$

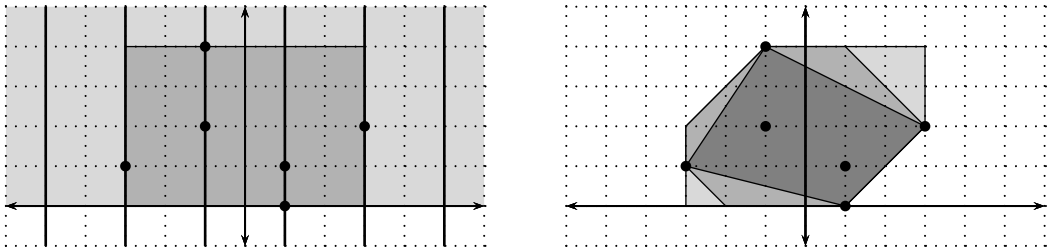
The functions γ and α are called concretization and abstraction function respectively.

The goal of a reachability analysis is to compute $\mathsf{reach}(\mathcal{I}) = \mathsf{post}^*(\mathcal{I})$, which is equivalent to computing the least fixed point of the monotonic function $F = \lambda S.\mathcal{I} \cup \mathsf{post}(S)$. An abstract interpretation-based reachability analysis computes this fixed point in the abstract domain, *i.e.*, it computes $lfp(F^\sharp)$ where $F^\sharp = \alpha \circ F \circ \gamma$.

The following theorem [CC77] states the soundness of an abstract interpretation-based analysis:

Theorem 3.3 (Sound over-approximation) *Let $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$ be a Galois connection and F a monotonic function, then $\alpha(lfp(F)) \sqsubseteq^\sharp lfp(\alpha \circ F \circ \gamma)$.*

This means that the fixed point in the abstract domain is an over-approximation of the fixed point in the concrete domain.



(a) Signs (light gray), parity (vertical lines) and intervals (dark gray)

(b) Zones (outermost), octagons and convex polyhedra (innermost)

Figure 3.4: Comparison of the precision of various abstract domains w.r.t. abstracting a concrete set of points.

3.3 Abstract Domains

Abstract domains are used to represent and manipulate sets of program states. Since we are generally dealing with infinite state systems a *finite representation* of infinite sets is needed. For numerical state spaces convex geometric shapes like rectangles, octagons or polyhedra are typical candidates for numerical abstract domains. However, such shapes are not capable of representing the exact sets of program states. Thus, the choice of an abstract domain induces a *static approximation* (*i.e.*, prior to the analysis) and restricts the set of possible invariants we can compute by a reachability analysis.

Characterizing domains. Fig. 3.4 compares the precision of some classical numerical abstract domains. Abstract domains can be characterized by their ability to represent relations between variables:

- An abstract value in *non-relational* domains like signs ($x \bowtie 0$ with $\bowtie \in \{<, \leqslant, =, \geqslant, >\}$), parity ($x$ is even or odd) and intervals ($l \leqslant x \leqslant u$) is the cartesian product of the values of the individual dimensions.
- *Weakly relational* domains allow a restricted coupling between dimensions. For example, octagons are defined by sum and difference constraints between two variables.
- *Fully relational* domains like convex polyhedra – actually "polytopes", *i.e.*, a generalization of polygons to n dimensions – allow a coupling between all dimensions.

The precision of the domains increases from non-relational to fully relational domains, but so does the computational complexity of the associated domain operations.

Defining domains. Besides a concretization function γ and an abstraction function α, an abstract domain must provide the following operations:

- The *test for inclusion* $\sqsubseteq$ is used for checking convergence (*i.e.*, having reached a fixed point $F^\sharp(S^\sharp) \sqsubseteq S^\sharp$) of the analysis.
- The *join* (or *union*) *operator* $\sqcup$ is used to merge abstract values from several incoming arcs of a location in a CFG for example.
- *Transformation by a program statement* (image, post-condition) $[\![f]\!]^\sharp$: During the analysis abstract values are transformed by the operators (statements) f occurring in the program. Ideally, we have the $[\![f]\!]^\sharp = \alpha \circ [\![f]\!] \circ \gamma$ (where $[\![f]\!]$ denotes the concrete semantics of f), but sometimes this "best abstract transformer" can only be over-approximated. We denote $[\![f]\!]^{-1}$ the inverse of a transformation (pre-image).
  Program statements are essentially
  - *assignments*, like $\boldsymbol{x}' = \mathbf{A}\boldsymbol{x} + \boldsymbol{b}$, which performs a linear (to be precise: affine)

transformation of the abstract value; and
- *guards* (*e.g.*, the conditions of an if-then-else construct) like the linear guard $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}$, which the abstract value is intersected with. We will call this operator *intersection by a guard* $\sqcap^g$.

– The *test for emptiness* $= \bot$ is used for checking whether the intersection of the reachable states with a given error set is empty.
– The *projection* (*existential quantification* of a variable $\exists x_i$) is needed for handling input variables, for example.

A low computational complexity of the domain operations is crucial for an efficient implementation. A *canonical representation* of abstract values is desirable because it enables a cheap test for equality and lowers memory consumption by sharing values.

We will now describe three abstract domains, intervals (§3.3.1), convex polyhedra (§3.3.2) and template polyhedra (§3.3.3) in detail and we give a short overview of other abstract domains (§3.3.4). §3.3.5 explains how to combine abstract domains.

3.3.1 Intervals

The interval domain [CC76] (also called *box* domain) abstracts the values of the program variables by their lower and upper bounds:

$$Int(\mathbb{R}^n) = \{\bot\} \cup \left(\{[l,u] \mid l,u \in \overline{\mathbb{R}} \wedge l \leqslant u\}\right)^n$$

where $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, \infty\}$. Moreover, we denote $\top =]-\infty, \infty[$.

The domain is ordered by the *inclusion* of intervals:

$$X \sqsubseteq Y \iff \begin{cases} \mathsf{tt} & \text{if } X = \bot \\ \mathsf{ff} & \text{if } X \neq \bot \wedge Y = \bot \\ \bigwedge_{i\in[1\ldots n]} l_i^Y \leqslant l_i^X \leqslant u_i^X \leqslant u_i^Y & \text{otherwise} \end{cases}$$

The *union* of two abstract values is the component-wise union of the intervals:

$$X \sqcup Y = \begin{cases} X & \text{if } \quad Y = \bot \\ Y & \text{if } \quad X = \bot \\ \times_{i\in[1\ldots n]}[\min\{l_i^X, l_i^Y\}, \max\{u_i^X, u_i^Y\}] & \text{otherwise} \end{cases}$$

Transformations by program statements are computed using interval arithmetic [Moo66]. For example: $\left[\!\!\left[\begin{matrix} x_1' = 2x_1 + x_2 + 1 \\ x_2' = -x_2 \end{matrix} \right]\!\!\right]^\sharp \begin{pmatrix} [0,2] \\ [1,3] \end{pmatrix} = \begin{pmatrix} 2[0,2] + [1,3] + 1 \\ -[1,3] \end{pmatrix} = \begin{pmatrix} [2,8] \\ [-3,-1] \end{pmatrix}$.

Existential quantification of a variable x_i simply means setting the respective component to $\top$: $\exists x_i : X = ([l_1,u_1], \ldots, [l_{i-1},u_{i-1}], \top, [l_{i+1},u_{i+1}], \ldots, [l_n,u_n])^T$

Interval analysis is very cheap: all the domain operations can be performed in linear time and space in the number of dimensions.

3.3.2 Convex Polyhedra

The convex polyhedra domain [CH78, Hal79] is a fully relational domain.

A convex polyhedron can be either represented by

Inclusion	$(V_1, R_1) \sqsubseteq (\mathbf{A}_2\boldsymbol{x} \leqslant \boldsymbol{b}_2) \iff \forall \boldsymbol{v} \in V_1 : \mathbf{A}_2\boldsymbol{v} \leqslant \boldsymbol{b}_2 \ \wedge \ \forall \boldsymbol{r} \in R_1 : \mathbf{A}_2\boldsymbol{r} \geqslant \mathbf{0}$
Canonicalization	$\mathsf{can}(X)$ Chernikova's algorithm
Abstraction	$\alpha(X)$ Chernikova's algorithm
Concretization	$\gamma(X^\sharp)$ Chernikova's algorithm
(Linear guard) intersection	$(\mathbf{A}_1\boldsymbol{x} \leqslant \boldsymbol{b}_1) \sqcap (\mathbf{A}_2\boldsymbol{x} \leqslant \boldsymbol{b}_2) = \mathsf{can}\left(\begin{pmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{pmatrix} \boldsymbol{x} \leqslant \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \end{pmatrix}\right)$
Union	$(V_1, R_1) \sqcup (V_2, R_2) = \mathsf{can}\left(\begin{pmatrix} V_1 \\ V_2 \end{pmatrix}, \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}\right)$
Linear transformation	$[\![\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}]\!]^\sharp(V, R) = (\{\mathbf{C}\boldsymbol{v} + \boldsymbol{b} \mid \boldsymbol{v} \in V\}, \{\mathbf{C}\boldsymbol{r} \mid \boldsymbol{r} \in R\})$ $[\![\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}]\!]^{\sharp -1}(\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}) = (\mathbf{A}\mathbf{C}\boldsymbol{x} \leqslant \boldsymbol{b} - \mathbf{A}\boldsymbol{d})$
Emptiness	$\gamma(V, R) = \varnothing \iff V = \varnothing$
Projection	$\exists x_i : X^\sharp$ Fourier-Motzkin elimination

Table 3.1: Convex polyhedra domain operations.

– the sets of *generators* (V, R), *i.e.*, the convex closure of vertices and rays

$$\gamma(V, R) = \{\boldsymbol{x} \mid \exists \boldsymbol{\lambda} \geqslant \mathbf{0}, \boldsymbol{\mu} \geqslant \mathbf{0} \ : \ \sum_i \lambda_i = 1 \wedge \boldsymbol{x} = \sum_i \boldsymbol{v}_i\lambda_i + \sum_j \boldsymbol{r}_j\mu_j\}$$

with the vertices $V = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_p\}$, $\boldsymbol{v}_i \in \mathbb{R}^n$ and the rays $R = \{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_q\}$, $\boldsymbol{r}_j \in \mathbb{R}^n$,

– or by a conjunction of linear *constraints* $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}$, *i.e.*, an intersection of halfspaces

$$\gamma(\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}) = \{\boldsymbol{x} \mid \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}\}$$

with $\boldsymbol{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{b} \in \mathbb{R}^m$.

$\top$ and $\bot$ denote the polyhedra $\mathbb{R}^n$ and $\varnothing$ respectively.

We can convert from one representation to the other one using the double description method [MRTT53, FP95], also known as *Chernikova's algorithm* [Che65, Ver92]. The conversion between constraint and generator representations has exponential worst case complexity in the number of dimensions n (see [FP95]). However, it also depends on the size of the polyhedra representation (number of generators or number of constraints), which is unbounded in general.

Domain operations. The definitions of the domain operations (cf., *e.g.*, [HPR97]) are summarized in Table 3.1.

Once a polyhedron is represented in both ways, some domain operations can be performed more efficiently using the generator representation only, others based on the constraint representation, and some making use of both. Hence any composition of domain operations requiring both representations has a worst-case exponential complexity.

Assuming that concrete sets X are given by constraints or points (vertices), Chernikova's algorithm transforms them into non-redundant (canonical) generator and constraint representations $X^\sharp$. In the case of generators this operation involves a convex approximation (convex hull) of the concrete set of points.

Mind that a linear guard, *i.e.*, the conjunction of linear constraints, is a convex polyhedron. The intersection of two convex polyhedra is again a convex polyhedron: $X_1 \cap X_2 = X_1 \sqcap X_2$. In contrast, the union of two convex polyhedra is not convex in

general. It is computed by the union of their generators which implies that the result is the convex hull of the original polyhedra: $X_1 \cup X_2 \subseteq X_1 \sqcup X_2$.

For the projection $\exists x_i : X$ *Fourier-Motzkin elimination* is used, which is an algorithm for eliminating variables from a system of linear inequalities (constraint representation) and which has doubly exponential worst-case complexity (see, *e.g.*, [Wil86] for details).

Furthermore, we will use the following operations:

– The *Minkowski sum* of two polyhedra $X = X_1 + X_2$ is defined by $X = \{\boldsymbol{x}_1 + \boldsymbol{x}_2 \mid \boldsymbol{x}_1 \in X_1, \boldsymbol{x}_2 \in X_2\}$.
– The *time elapse* operation [HPR97], defined as $X_1 \nearrow X_2 = \{\mathbf{x}_1 + t\mathbf{x}_2 \mid \mathbf{x}_1 \in X_1, \mathbf{x}_2 \in X_2, t \in \mathbb{R}^{\geqslant 0}\}$, can be implemented using the generator representations: $(V_1, R_1) \nearrow (V_2, R_2) = (V_1, R_1 \cup V_2 \cup R_2)$.

All these domain operations are implemented in polyhedra libraries like PPL [BHZ05] or NewPolka [Jea00].

Discussion. As already mentioned convex polyhedra operations have a high computational complexity. However, they provide a good precision: Since they are closed by affine transformations, images by statements $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ can be computed without approximations. In comparison to template polyhedra (see below), which have a fixed shape, the shape of convex polyhedra is dynamic and, thus, analyses with convex polyhedra can "discover" non-trivial shapes of invariants.

We will use convex polyhedra in the context of abstract acceleration (see §§4–6 and §8) and hybrid system analysis (§12 and §13).

3.3.3 Template Polyhedra

Abstract interpretation with template polyhedra was introduced by Sankaranarayanan et al [SSM05]. It is based on polyhedra the shape of which is fixed by a so-called (linear) template constraint matrix, or short template, $\mathbf{T} \in \mathbb{R}^{m \times n}$ where each row contains at least one non-zero entry. The set of template polyhedra $Pol^{\mathbf{T}}$ generated by $\mathbf{T}$ is $\{X_{\boldsymbol{d}} \mid \boldsymbol{d} \in \overline{\mathbb{R}}^m\}$ with $X_{\boldsymbol{d}} = \{\boldsymbol{x} \mid \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d}, \boldsymbol{x} \in \mathbb{R}^n\}$.

For example, $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ is a template constraint matrix of intervals for a system with a single variable x: it represents the constraints $x \leqslant d_1 \wedge -x \leqslant d_2$, *i.e.*, $-d_2 \leqslant x \leqslant d_1$. Similarly, zonal $(\pm x_i \leqslant d, x_i - x_j \leqslant d)$ and octagonal $(\pm x_i \leqslant d, \pm x_i \pm x_j \leqslant d)$ constraints can be expressed using templates.

$\top$ and $\bot$ are naturally represented by the bound vectors ∞ and $-\infty$ respectively. When analyzing a CFG, the templates may vary from location to location.

In the following, the operators min, sup, a.s.o. are point-wisely lifted to vectors.

Domain operations. Table 3.2 summarizes the domain operations. They involve min and max operators over infinite sets. These optimization problems are solved using linear programming.

We recall the most important notions of *linear programming* (LP): A linear programming problem is of the form (primal problem): $\max \boldsymbol{c}^T\boldsymbol{x}$ subject to $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{x} \geqslant 0$. $\boldsymbol{c}^T\boldsymbol{x}$ is the objective function and $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{x} \geqslant 0$ is the feasible region, which is a convex polyhedron. The dual problem is $\min \boldsymbol{b}^T\boldsymbol{y}$ subject to $\mathbf{A}^T\boldsymbol{y} \geqslant \boldsymbol{c} \wedge \boldsymbol{y} \geqslant 0$. The strong duality theorem says that if $\boldsymbol{x}^*$ is an optimal solution of the primal problem, then $\boldsymbol{y}^*$ is an optimal solution of dual problem, such that $\boldsymbol{c}^T\boldsymbol{x}^* = \boldsymbol{b}^T\boldsymbol{y}^*$. If the primal problem is unbounded (*i.e.*, the optimal solution is ∞), then the dual problem is infeasible and vice

Inclusion	$\boldsymbol{c} \sqsubseteq_{\mathbf{T}} \boldsymbol{d} \iff \bigwedge_i c_i \leqslant d_i$
Abstraction	$\alpha_{\mathbf{T}}(X) = \min\{\boldsymbol{d} \in \overline{\mathbb{R}}^m \mid \gamma_{\mathbf{T}}(\boldsymbol{d}) \supseteq X\},\ X \subseteq \mathbb{R}^n$
Concretization	$\gamma_{\mathbf{T}}(\boldsymbol{d}) = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{R}^n, \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d}\},\ \boldsymbol{d} \in \overline{\mathbb{R}}^m$
Union	$\boldsymbol{d} \sqcup_{\mathbf{T}} \boldsymbol{d}' = \big(\max(d_1, d'_1), \ldots, \max(d_m, d'_m)\big)$
Linear guard intersection	$\boldsymbol{d} \sqcap^g_{\mathbf{T}} (\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}) = \min\{\boldsymbol{d}' \mid \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d} \wedge \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d}'\}$
Linear transformation	$[\![\boldsymbol{x}' = \mathbf{A}\boldsymbol{x} + \boldsymbol{b}]\!]^\sharp(\boldsymbol{d}) = \min\{\boldsymbol{d}' \mid \boldsymbol{x}' = \mathbf{A}\boldsymbol{x} + \boldsymbol{b} \wedge \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d} \wedge \mathbf{T}\boldsymbol{x}' \leqslant \boldsymbol{d}'\}$
Emptiness	$\gamma(\boldsymbol{d}) = \varnothing \iff \boldsymbol{d} = -\infty$
Projection	$\exists x_i : \boldsymbol{d} = \min\{\boldsymbol{d}' \mid \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d} \wedge \mathbf{T}\boldsymbol{x}' \leqslant \boldsymbol{d}' \wedge \forall j \neq i : x'_j = x_j\}$

Table 3.2: Template polyhedra domain operations.

versa. The classical algorithm for solving LP problems is the simplex algorithm [DT97]. There are other methods, *e.g.*, interior point methods [Kar84], which have polynomial complexity, but the simplex algorithm is despite its exponential worst-case complexity the most efficient and most-widely used in practice.

Hence, the theoretical complexity of the operations of the template polyhedra domain is polynomial in the number of dimensions and the size of the template constraint matrix.

Besides, there are extensions [AGG10a, GS10] from linear to quadratic templates using semi-definite programming for resolving the optimization problems involved.

Discussion. Template polyhedra are computationally efficient and they possess a great flexibility in defining the shape of possible invariants. However the shape must be fixed *prior to* the analysis, when no information except the program code itself is available. Guard conditions might be good candidates, but in general the shape of an invariant cannot be guessed from the source code. The proposed methods are heuristic and include the use of general shapes like octagonal constraints, deriving additional constraints from a given template [SSM05], performing a truncated polyhedral analysis to discover constraints or adding random constraints.

However, a large template size penalizes the efficiency of the domain operations. Thus, finding templates consisting of a small number of relevant constraints is still an open problem.

We will use template polyhedra in the context of max-strategy iteration (see §3.4.3 and §9).

3.3.4 Other Domains

We pick some other domains to show the large variety of abstract domains:

There are weakly relational domains with low polynomial complexity, like *zones* [HNSY92, Min01a] (defined by constraints $\pm x_i \leqslant d, x_i - x_j \leqslant d$), *octagons* [Min01b] ($\pm x_i \leqslant d, \pm x_i \pm x_j \leqslant d$) and their generalization *logahedra* [HK09], *i.e.*, to two-variable constraints with coefficients that are powers of two up to some bound parameter.

Among the fully relational domains count *zonotopes* [Gir05, GP06], which are bounded, central-symmetric polyhedra defined by $\boldsymbol{x} = \boldsymbol{c} + \sum_i \boldsymbol{g}_i \lambda_i, \lambda_i \in [-1, 1]$, *i.e.*, the Minkowski

sum of a central point and a finite set of line segments; and *ellipsoids* [KV00] ($\mathbf{x}^T \boldsymbol{A}\mathbf{x} \leqslant \boldsymbol{b}$). These domains are both closed under linear transformations.

Tropical polyhedra [AGG10b] are based on max-plus algebra and they are able to represent some non-convex shapes. *Donut domains* [GIB+12] are shapes computed by the difference between two convex sets.

The linear *congruence* domain [Gra91] ($\bigwedge_i \boldsymbol{a}_i^T \boldsymbol{x} \equiv c_i \mod m_i$) is used for proving divisibility properties, *e.g.*, in algorithms involving *gcd* and *lcm*. *Varieties* [SSM04] are algebraic sets defined by the common zeros of a set of polynomials p_j, *i.e.*, $\{\boldsymbol{x} \in \mathbb{C}^n \mid \bigwedge_j p_j(\boldsymbol{x}) = 0\}$. This domain has been used for analyzing hardware division algorithms for instance.

3.3.5 Composite Abstract Domains

Composite abstract domains [CC79] enable, for example, to analyze different types of variables using suitable abstract domains. In a logico-numerical program with state space $\mathbb{B}^n \times \mathbb{R}^m$, for instance, sets of Boolean states $\wp(\mathbb{B}^n)$ can be presented (exactly) by Boolean formulas, while sets of numerical states $\wp(\mathbb{R}^m)$ are abstracted by convex polyhedra $Pol(\mathbb{R}^m)$.

Another reason to combine domains is to improve the precision. For example, analyzing a numerical program with state space $\mathbb{R}^n$ using a single analysis combining convex polyhedra $Pol(\mathbb{R}^n)$ and linear congruences $LinCongr(\mathbb{R}^n)$ yields more precise results than analyzing the program first with convex polyhedra and then with linear congruences or the other way around [CC79].

We consider the following two ways to construct composite abstract domains:

– A *product domain* $A \times B$ is able to represent a conjunctive relation between the two subdomains A and B. For example in the logico-numerical setting above, we have the abstraction: $\wp(\mathbb{B}^n \times \mathbb{R}^m) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \wp(\mathbb{B}^n) \times Pol(\mathbb{R}^m)$: in this domain $b \wedge x \geqslant 0$ can be presented exactly whereas $b \Leftrightarrow x \geqslant 0$ is abstracted to $\top \times \top$.

– A *power domain* $B^A = A \to B$ can represent one abstract value of B for each value of the set A. For example for the logico-numerical state space above, we have the abstraction $\wp(\mathbb{B}^n \times \mathbb{R}^m) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \mathbb{B}^n \to Pol(\mathbb{R}^m)$: this domain can exactly represent $b \Leftrightarrow x \geqslant 0$, whereas $b \wedge x \geqslant 10 \vee b \wedge x \leqslant 0$ is abstracted to $b \to \top$.

Domain operations and reduction. Since the subdomains in a composite domain are not semantically independent, a naive combination of the domain operations would lead to non-canonical representations of abstract values. Hence, a canonicalization operation is needed in order to *reduce* the result of operations:

For example, given two abstract domains A and B combined into a product domain with the abstraction and concretization functions α_A, α_B and γ_A, γ_B respectively, we have the canonicalization $\mathsf{can}(a_1, b_1) = \bigsqcap\{(a_2, b_2) \mid \gamma_A(a_1) \wedge \gamma_B(b_1) = \gamma_A(a_2) \wedge \gamma_B(b_2)\}$. A consequence of the canonicalization is, for example, that in the combined domain an abstract value (a, b) equals $\bot$ if $a = \bot_A$ or $b = \bot_B$.

We give the resulting operations of the (reduced) product domain:

$$\langle A, \sqsubseteq_A, \bot_A, \top_A, \sqcap_A, \sqcup_A \rangle \times \langle B, \sqsubseteq_B, \bot_B, \top_B, \sqcap_B, \sqcup_B \rangle = \langle A \times B, \sqsubseteq, \bot, (\top_A, \top_B), \sqcap, \sqcup \rangle$$

$$\text{with } \begin{cases} \sqsubseteq & = & \lambda(a_1, b_1), (a_2, b_2).(a_1 \sqsubseteq_A a_2) \wedge (b_1 \sqsubseteq_B b_2) \\ \sqcap & = & \lambda(a_1, b_1), (a_2, b_2).\mathsf{can}(a_1 \sqcap_A a_2, b_1 \sqcap_B b_2) \\ \sqcup & = & \lambda(a_1, b_1), (a_2, b_2).(a_1 \sqcup_A a_2, b_1 \sqcup_B b_2) \end{cases}$$

The abstraction and concretization functions are defined as $\alpha(S) = \mathsf{can}(\alpha_A(S), \alpha_B(S))$ and $\gamma(a,b) = \gamma_A(a) \wedge \gamma_B(b)$ respectively. A transformation $[\![f]\!]$ is then abstracted to $[\![f]\!]^\sharp = \alpha \circ [\![f]\!] \circ \gamma$ in the combined abstract domain.

Similarly, one can define the domain operations for the (reduced) power domain (see [CC79] for details). We will return to logico-numerical domains constructed by product and power domain combinations in §7.

Besides, *disjunctive* combinations of elements of the same domain have been considered too, *e.g.*, in [CC79, GR98, BHZ04, SISG06]. This helps for example to overcome the convexity limitations of most numerical domains, but it entails an exponentially higher complexity and difficulties in canonicalizing representations.

3.4 Fixed Point Computation

The classical method for fixed point computation is *Kleene iteration* (§3.4.1). However, in general this iteration does not terminate. Therefore an extrapolation operator called *widening* is used in order to speed up convergence for the price of additional approximations. Numerous *improvements for widening* (§3.4.2) have been developed to reduce the involved loss of precision.

Strategy improvement (or strategy iteration) methods (§3.4.3) enable the computation of the fixed point without the use of widening for certain abstract domains with infinite height, *e.g.*, template polyhedra.

3.4.1 Kleene Iteration with Widening

We denote $F^\sharp = \lambda S^\sharp . S^{\sharp 0} \sqcup \mathsf{post}^\sharp(S^\sharp)$ with $S^{\sharp 0} = \alpha(\mathcal{I})$ and $\mathsf{post}^\sharp = \alpha \circ \mathsf{post} \circ \gamma$. In order to avoid clutter, we will not put the $^\sharp$ superscript in the following when it is clear from the context that we are manipulating abstract values. Moreover, we will use the term "Kleene iteration" to refer to the *ascending* Kleene iteration (see Thm. 3.2).

Widening. Kleene iteration computes the least fixed point $lfp(F)$ by $\bigsqcup_{k \in \mathbb{N}^{\geqslant 0}} (F)^k(\bot)$. However, it may require an infinite number of iterations when the abstract domain has infinite ascending chains or a large number of iterations for lengthy finite ascending chains.

For this reason, abstract interpretation introduces a *widening operator* that guarantees convergence:

Definition 3.10 (Widening operator ∇) *$\nabla : S \times S \to S$ is a widening operator iff*

- $\forall s_1, s_2 \in S : s_1 \sqsubseteq s_1 \nabla s_2 \wedge s_2 \sqsubseteq s_1 \nabla s_2$
- *for any ascending chain $s_0 \sqsubseteq s_1 \sqsubseteq \ldots$, the chain (s'_i) with $s'_0 = s_0$, $s'_{i+1} = s'_i \nabla s_{i+1}$ eventually stabilizes,* i.e., $\exists n \in \mathbb{N} : \forall n' > n : s'_{n'} = s'_n$

The standard widening operators of numerical abstract domains suppose that the program has a certain regularity, and thus, a constraint that changes in successive iterations will continue to be shifted in the same direction. Hence, it is extrapolated to infinity (resp. minus infinity). For instance the standard widening operator for intervals is defined as follows (component-wise):

$$\left\{ \begin{array}{rcl} \bot \nabla X & = & X \nabla \bot = X \\ [l_1, u_1] \nabla [l_2, u_2] & = & \left[\begin{array}{l} \text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \\ \text{if } u_2 > u_1 \text{ then } \infty \text{ else } u_1 \end{array} \right] \end{array} \right.$$

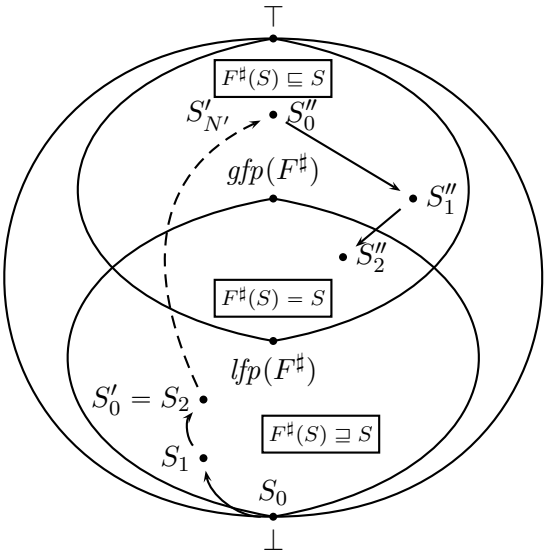


Figure 3.5: Kleene iteration with delayed widening ($N=2$) and two descending iterations

For convex polyhedra $X_1 \nabla X_2$ consists, roughly speaking, of those constraints of X_1 that are satisfied by X_2 (see [Hal79, HPR97] for a detailed presentation).

Widening for template polyhedra [SSM05] works similarly to intervals.

Fixed point computation. The classical analysis procedure [CC77, CC92b] consists of three phases:

(1) an *ascending sequence* $(S_n)_{0 \leqslant n \leqslant N}$ of post-condition computations:

$$\begin{cases} S_0 &= S^0 \\ S_{n+1} &= F(S_n) \text{ for } n < N \end{cases}$$

(2) a (delayed) *widening sequence* $(S'_n)_{0 \leqslant n \leqslant N'}$ that is guaranteed to converge to a post-fixed point $S'_{N'}$ in a finite number of steps:

$$\begin{cases} S'_0 &= S_N \\ S'_{n+1} &= S'_n \nabla (F(S'_n)) \quad \text{until convergence } (S'_{N'} \sqsubseteq S'_{N'+1}) \end{cases}$$

(3) a (truncated) *descending sequence* $(S''_n)_{0 \leqslant n \leqslant N''}$ of post-condition computations for approaching the fixed point:

$$\begin{cases} S''_0 &= S'_{N'} \\ S''_{n+1} &= F(S''_n) \quad \text{for } n < N'' \end{cases}$$

This procedure is illustrated in Fig. 3.5.

Descending iterations generally do not converge: Cousot and Cousot [CC77] propose the use of a narrowing operator in order to force the convergence to a fixed point. In practice, the descending sequence is usually truncated, which is sound because the result of the widening sequence satisfies $F(S'_{N'}) \sqsubseteq S'_{N'}$, and hence, by Thm. 3.2 all elements of the descending sequence are upper bounds of the least fixed point.

Example 3.1 (Kleene Iteration with widening) *We analyze the following program*

$$\begin{cases} \mathcal{I} &= (x=0) \\ x' &= \mathit{if}\ x \leqslant 9\ \mathit{then}\ x+1\ \mathit{else}\ 0 \end{cases}$$

in the interval domain with $N = 2$ *and* $N'' = 1$*:*

$$\begin{array}{lllllll} S_0 &=& [0,0] & & & & \\ S_1 &=& [0,0] \sqcup [1,1] &=& [0,1] & & \\ S_2 &=& [0,0] \sqcup [1,2] &=& [0,2] &=& S'_0 \\ S'_1 &=& [0,2] \nabla [0,3] &=& [0,\infty] &=& S'_2 \quad = \quad S''_0 \\ S''_1 &=& [0,10] & & & & \end{array}$$

Iteration strategies

When analyzing a CFG one assigns to each location $\ell \in L$ an abstract value from an abstract domain A. Thus the overall abstract domain has the structure of the power domain $(L \to A)$. Therefore, one has to solve the fixed point equation

$$S = S^0 \sqcup \lambda \ell' \,.\, \bigsqcup_{\ell' \in L} \mathsf{post}_{\ell,\ell'}(S(\ell))$$

where $S, S^0 \in (L \to A)$.

A naive way to perform Kleene iteration is to apply all the $\mathsf{post}_{\ell,\ell'}$ operators in parallel as suggested by the above equation. However, Kleene iteration actually *propagates* the abstract values through the graph in a wave-like fashion while transforming them by the $\mathsf{post}_{\ell,\ell'}$ operators associated to the arcs (transitions) of the graph. Thus, the computation should be serialized.

It can be shown [Cou77, CC77] that any order, *i.e.*, iteration strategy, which does not forget any transition indefinitely, computes the least fixed point ("chaotic iteration").

Bourdoncle [Bou93] proposes heuristics to derive efficient iteration strategies from the graph structure. He first computes a weak topological ordering by depth-first exploration of the CFG. The ordering can be written as a well-parenthesized permutation of the locations L, such that the locations within two matching parentheses correspond to the strongly connected components of the graph. For example, the ordering obtained for the CFG in Fig. 3.6a is $(\ell_1\ (\ell_2))\ \ell_3$.

He proposes two strategies: the *iterative strategy* ($(\ell_1\ \ell_2)^*\ \ell_3$ for Fig. 3.6a) stabilizes the outer strongly connected components, whereas the *recursive strategy* ($(\ell_1\ (\ell_2)^*)^*\ \ell_3$) recursively stabilizes components from the innermost to the outermost ones such that the inner components are re-stabilized in each iteration of an outer component.

Moreover, it suffices to apply widening only in the loop heads, *i.e.*, the first location of a strategy component. Also, convergence can be detected by the stabilization of the loop heads. For Fig. 3.6a the widening points (marked with W) are $(\ell_1 W\ (\ell_2 W))\ \ell_3$.

Backward analysis

Co-reachability analysis of a CFG is performed by transposing the graph and inverting the transition relations associated to the arcs, *i.e.*, the arcs are labeled by post^{-1}. The Kleene iteration is started from the error set $\mathcal{E}$, and the set to be avoided is the set of initial states $\mathcal{I}$. Hence, a backward analysis is just a forward analysis on the reverse CFG. Everything stated in this section applies analogously to backward analyses.

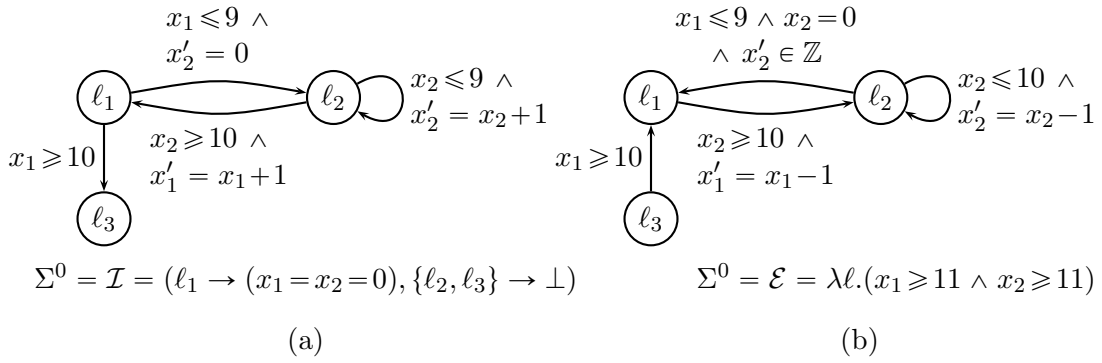


Figure 3.6: Example 3.2: a CFG (a) and its reverse CFG (b).

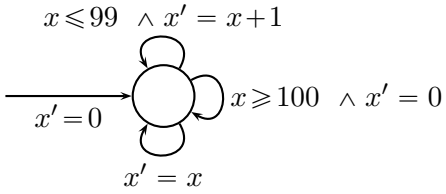


Figure 3.7: CFG for Example 3.3: the descending sequence fails.

Example 3.2 (Backward analysis) *Fig. 3.6b shows the reverse CFG of Fig. 3.6a. Starting from the set of error states* $\mathcal{E} = (x_1 \geqslant 11 \wedge x_2 \geqslant 11)$*, the backward analysis converges with the invariant* $\{\ell_1, \ell_3\} \to (x_1 \geqslant 11 \wedge x_2 \geqslant 11), \ell_2 \to (x_1 \geqslant 10 \wedge x_2 \geqslant 11)$*, which does not intersect with* $\mathcal{I} = (x_1 = x_2 = 0)$*. Mind that if* post *is not injective,* post^{-1} *is a relation.*

3.4.2 Improvements of Widening

Although Cousot and Cousot [CC92b] show that the approach using Kleene iteration with widening and infinite height lattices can discover invariants that finite height lattices cannot discover, the dynamic approximations induced by widening lead quite often to an important loss of precision. There are several reasons for these problems:

- The standard widening operators are *not monotonic*, *e.g.*, $[0,2]\nabla[0,4] = [0,\infty]$, but $[1,2]\nabla[0,3] = \top$ (although $[1,2] \sqsubseteq [0,2]$ and $[0,3] \sqsubseteq [0,4]$).
- Descending iterations fail to recover information if the result of the widening sequence $S'_{N'}$ is already a fixed point, *i.e.*, $\mathsf{post}(S'_{N'}) = S'_{N'}$.

Example 3.3 (Descending sequence fails) *The program in Fig. 3.7 (cf. [Mon09]) exhibits the latter problem: After widening we obtain* $S'_1 = [0,\infty] = S''_0$*, but then* $S''_1 = [0,100] \sqcup [0,0] \sqcup [0,\infty] = [0,\infty] = S''_0$*. The descending sequence fails to improve the result, because the transition* $x' = x$ *keeps "injecting" the value* $S''_0 = [0,\infty]$ *in the join of the three incoming transitions.*

Moreover, the delay N of widening has an important effect on precision, as shows Ex. 3.4. However, delaying widening might be expensive, especially with convex polyhedra where coefficients quickly become huge. Mind that delaying widening is not equivalent to loop unrolling, because the convex union is taken after each iteration, whereas loop unrolling creates separate locations.

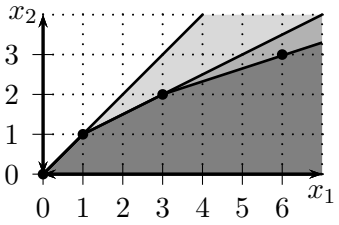


Figure 3.8: Delayed widening (Ex. 3.4): Abstract reachable sets for $N = 1$ (whole shaded area), $N = 2$ (gray and dark gray) and $N = 3$ (dark gray)

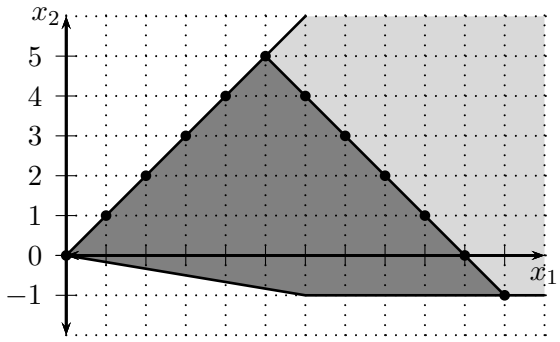


Figure 3.9: Taking into account loop phases (Ex. 3.5): Reachable sets computed by Kleene iteration (whole shaded area) with widening ($N=1$) and descending iterations ($N''=1$), and by guided static analysis (dark gray).

Example 3.4 (Delayed widening) *For the following program we get increasingly better approximations with increasing delay N (depicted in Fig. 3.8 for $N = 1, 2, 3$):*

$$\begin{cases} \mathcal{I}(x_1, x_2) &= (0, 0) \\ x_1' &= \textit{if } x_2 \geqslant 0 \textit{ then } x_1 + x_2 + 1 \textit{ else } 0 \\ x_2' &= \textit{if } x_2 \geqslant 0 \textit{ then } x_2 + 1 \textit{ else } 0 \end{cases}$$

Taking into account loop phases. Standard widening performs an extrapolation based on the hypothesis that the program behavior is somehow regular. However, this is not the case if a loop body consists of several paths (loop phases, see *e.g.*, CFG in Fig. 2.1). Distinguishing these loop phases can be exploited to obtain a better precision [Hal93a].

Guided static analysis [GR07] follows this idea by alternating ascending and descending sequences on a strictly increasing, finite sequence of restrictions of the program (by adding new transitions) which converges towards the original program. In many cases this method improves the precision, but it ultimately relies on the effectiveness of the descending iterations.

Example 3.5 (Loop phases) *The overall reachable states for the CFG in Fig. 2.1 (§2.1.2) obtained by a classical Kleene iteration with widening respectively guided static analysis are depicted in Fig. 3.9.*

Widening with thresholds. The idea of widening with thresholds (or "widening up to") is to limit widening by a given set of constraints [Hal93a, CCF+09]. To achieve this, the widening operator is parametrized with a set of *threshold constraints* $\mathcal{T}$:

$$S_n \nabla_{\mathcal{T}} (S_n \sqcup \mathsf{post}(S_n)) = (S_n \nabla (S_n \sqcup \mathsf{post}(S_n))) \sqcap \bigsqcap \{T \in \mathcal{T} \mid S_n \sqsubseteq T \wedge (S_n \sqcup \mathsf{post}(S_n)) \sqsubseteq T\}$$

Hence, widening with thresholds is computed by applying the standard widening operator and then intersecting with those constraints in $\mathcal{T}$ that are satisfied by both arguments of the widening operator.

Example 3.6 (Widening with thresholds) *An analysis of the CFG in Fig. 3.6a (cf. [LCJG11]) with Kleene iteration and widening ($N = 2, N'' = 2$) yields the overall invariant $x_1 \geqslant 0 \wedge 0 \leqslant x_2 \leqslant 10$. Using widening with the threshold sets in locations ℓ_1, ℓ_2 respectively $\mathcal{T}_1 = \{x_1 \leqslant 10, x_2 = 10\}$ and $\mathcal{T}_2 = \{x_1 \leqslant 9, x_2 \leqslant 10\}$ yields the desired invariant $0 \leqslant x_1 \leqslant 10 \wedge 0 \leqslant x_2 \leqslant 10$.*

The problem is, though, how to find a set of relevant threshold constraints. A *static threshold inference* method based on propagating the post-condition of a loop guard to the widening points of the CFG is proposed in [LCJG11]. *Dynamic threshold inference* methods are for example *counterexample-refined widening* [WYGI07], which is based on an (under-approximating) backward analysis, and *widening with landmarks* [SK06], which extrapolates threshold constraints by estimating the number of loop iterations until the guard of the loop is violated.

Loop acceleration methods. These techniques take into account information about the dynamics of the loop body in order to compute the precise effect of an unbounded number of loop iterations (*e.g.*, [GH06]). Since these methods have shown promising results, we will use them in this thesis for improving the precision. We will discuss them in detail in §5.

3.4.3 Strategy Iteration

Strategy (or policy) iteration methods [CGG+05, GGTZ07, AGG10a, GS07a, GS07b, GSA+12] are a way to solve fixed point equations over infinite height lattices *without* the need for a widening operator.

Originally, strategy iteration was developed for solving stochastic control problems. Later it was applied to two-player zero-sum games and min-max-plus systems, and more recently to solving abstract semantic equations in static analysis. More information on the historical background can be found, *e.g.*, in [CGG+05].

The main idea of strategy iteration is to iteratively approximate the least fixed point of the abstract semantic equation $S^\sharp = F^\sharp(S^\sharp)$ by fixed points of "simpler", more efficiently computable semantic equations $F^{\sharp(i)}$, called strategies, such that a fixed point of $F^\sharp$ is guaranteed to be found after solving a finite number of equations $S^\sharp = F^{\sharp(i)}(S^\sharp)$. Depending on whether the least fixed point of $F^\sharp$ is approached from above or below, the methods are called min- or max-strategy iteration respectively.

Current strategy iteration methods are limited to template domains (§3.3.3).

Min-strategy iteration

Min-strategy iteration [CGG+05, GGTZ07, AGG10a] aims at computing least fixed points of monotone self-maps F where $F(\boldsymbol{x}) = \min\{\pi(\boldsymbol{x}) \mid \pi \in \Pi\}$ for all $\boldsymbol{x}$. The set of self-maps Π is called the min-strategies. For example, F could be the min of max of affine functions. Then a min-strategy π is the choice of an argument of the min operator.

The idea is that the least fixed point of a strategy *lfp* π can be computed more efficiently than *lfp* F, *e.g.*, with the help of an LP solver. Starting with $\pi^{(0)}$, the decreasing sequence of abstract values $(\textit{lfp}\ \pi^{(k)})_k$ stabilizes if *lfp* $\pi^{(k)}$ is a fixed point of F.

The improvement of a strategy $\pi^{(k)}$ amounts to finding a $\pi^{(k+1)}$ such that $\pi^{(k+1)}(\boldsymbol{x}^{(k)}) = F(\boldsymbol{x}^{(k)})$: Let $\boldsymbol{x}^{(k)} = \mathit{lfp}\ \pi^{(k)}$, then we have $F(\boldsymbol{x}^{(k)}) \sqsubset \boldsymbol{x}^{(k)}$ because otherwise $\boldsymbol{x}^{(k)}$ would be a fixed point of F and we are done. This means that there is at least one component $x_i^{(k)}$ of $\boldsymbol{x}^{(k)}$ for which $F(\boldsymbol{x}^{(k)})_i < x_i^{(k)}$. Thus, we have to choose an argument $\pi^{(k+1)}$ of the min operator in F such that $\pi^{(k+1)}(\boldsymbol{x}^{(k)})_i = F(x^{(k)})_i$. Furthermore, this ensures that the new strategy actually improves the fixed point: $\mathit{lfp}\ \pi^{(k+1)} \sqsubset \mathit{lfp}\ \pi^{(k)}$.

However, min-strategy iteration does not necessarily find the least fixed point of f: in order to guarantee that the procedure terminates with the least fixed point, f must be non-expansive, *i.e.*, $\forall x_1, x_2 : \|f(x_1) - f(x_2)\|_\infty \leqslant \|x_1 - x_2\|_\infty$. Moreover, the choice of the initial min-strategy is important, because otherwise the procedure may fail to find the least fixed point as well.

An advantage of min-strategy iteration is that the computation can be stopped before convergence and the obtained result will be a sound over-approximation.

Max-strategy iteration

Max-strategy iteration [GS07a, GS07b, GS08, GS10, GS11] is a method for computing the least solution of a system of equations $\mathcal{M}$ of the form $\boldsymbol{\delta} = \boldsymbol{F}(\boldsymbol{\delta})$, where $\boldsymbol{\delta}$ are the template bounds, and F_i, $0 \leqslant i \leqslant n$ is a finite maximum of monotonic and concave operators $\mathbb{R}^n \to \mathbb{R}$, *e.g.*, affine functions.

The system of equations $\mathcal{M}$ is constructed from the abstract semantics of the CFG transitions in the template domain:

$$\text{for each } \ell' \in L : \boldsymbol{\delta}_{\ell'} = \max\Big(\{\boldsymbol{d}^0_{\ell'}\} \cup \{[\![R]\!]^\sharp(\boldsymbol{\delta}_\ell) \mid (\ell, R, \ell') \in \mathcal{R}\}\Big)$$

where $\boldsymbol{d}^0_{\ell'}$ are the initial bounds in location ℓ'. A max-strategy μ induces a subsystem of $\mathcal{M}$ by selecting an argument of the max-operator for each template bound variable $\delta_i \in \boldsymbol{\delta}$ in each location ℓ'.

The least solution of the system of equations $\mathit{lfp}[\![\mathcal{M}]\!]$ is computed with the help of the max-strategy improvement algorithm: Starting from the strategy $\mu^{(0)}$ that yields the abstract value $\bot$, an increasing sequence of abstract values $\boldsymbol{d}^{(k)} = \mathit{lfp}[\![\mu^{(k)}]\!]$ is computed by improving the strategies until no more improvement is possible. The least fixed point of the current strategy $\mu^{(k)}$ is computed with the help of mathematical programming.

The max-strategy improvement algorithm is guaranteed to compute the least fixed point. A more detailed presentation of max-strategy iteration will be given in §9.

3.5 Tools and Libraries

Until now, the commercial success of abstract interpretation manifests itself in proving the absence of runtime errors in C programs, which represent the majority of industrial embedded code. We list some commercial and academic tools that particularly target this kind of applications:

– PolySpace Verifier[1] targets the languages C, C++ and Ada.
– Astrée[2] (in French: Analyseur Statique de logiciels Temps-RÉel Embarqués) [BCC+02, BCC+03, CCF+05, CCF+09] is at its origin an academic development that is now

[1] `http://www.mathworks.com/products/polyspace`
[2] `http://www.astree.ens.fr`

commercialized by ABSINT[3]. It has been successfully used in the verification of avionics software.

– F-SOFT[4] and FRAMA-C[5] are static analyzers for C programs.

Regarding other applications, there is a large number of academic tools and prototypes that implement a variety of methods. However, these tools often have proprietary input formats and abstract domain implementations. We list here some academic developments that had a significant impact in the last decade:

– Libraries:
  - The abstract domain library APRON[6] [JM09] aims at providing a common application programming interface for abstract domains: it includes a variety of domains like intervals, octagons, convex polyhedra, linear congruences, and the reduced product of convex polyhedra and linear congruences. APRON relies on the polyhedra library PPL[7] (Parma Polyhedra Library) for some if its domains.
  - BDDAPRON [Jea] is an extension of APRON that adds the direct product and power domains of Boolean formulas and any APRON domain.
– Tools:
  - TVLA[8] (Three-Valued Logic Analysis engine) is a static analysis framework for shape analysis, *i.e.*, properties about memory usage.
  - FLUCTUAT[9] [GMP02, GP06] is a static analyzer for evaluating the effect of rounding errors introduced by floating point operations, which is being adopted in industry to verify avionics software [DGP+09].
  - NBAC[10] [JHR99, Jea00, Jea03] is an analyzer for LUSTRE programs based on logico-numerical abstract domains and partition refinement techniques.
  - CONCURINTERPROC[11] is an abstract interpreter for computing invariants for a simple imperative, multi-threaded language. It is based on the BDDAPRON library.

[3] http://www.absint.com/astree/
[4] http://www.nec-labs.com/research/system/systems_SAV-website/fsoft-publications.php
[5] http://www.frama-c.com
[6] http://apron.cri.ensmp.fr/library/
[7] http://www.cs.unipr.it/ppl/
[8] http://www.cs.tau.ac.il/~tvla/
[9] http://www.di.ens.fr/~cousot/projects/DAEDALUS/synthetic_summary/CEA/Fluctuat/
[10] http://pop-art.inrialpes.fr/~bjeannet/nbac/
[11] http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi

Part I

Verification of Numerical Systems

Chapter 4

Acceleration and Abstract Acceleration

This chapter gives an overview of acceleration and abstract acceleration techniques. Acceleration methods aim at computing the *exact* set of reachable states in numerical transition systems. They are motivated by the analysis of communication protocols often modeled using counter machines or Petri nets. First achievements date back to the 1990s [BW94, BG97, FO97, CJ98, Boi98].

Unlike abstract interpretation, which overcomes the undecidability issue by computing a conservative approximation, acceleration identifies classes of systems for which the reachability problem is decidable and can be solved exactly, *e.g.*, programs of a certain structure with certain affine tests and assignments.

The idea of acceleration is to accelerate cycles labeled by a transition relation τ in the control structure of a program by computing the exact effect of its reflexive and transitive closure $\tau^* = \bigcup_{k \geqslant 0} \tau^k$ on a set of states X. Applied to the program of Fig. 4.1(a), we obtain the program of Fig. 4.1(b). If the program does not contain nested loops and all loops can be accelerated, then the method is complete. Otherwise, there are techniques for accelerating selected cycles in the hope to converge, but there is no guarantee to terminate in general.

Gonnord et al. [GH06, Gon07] have proposed the concept of *abstract acceleration* that integrates the acceleration idea into the abstract interpretation framework with convex polyhedra: wherever possible, simple loops are accelerated *in the abstract domain*, and in any other cases (multiple self-loops, nested loops, too expressive transi-

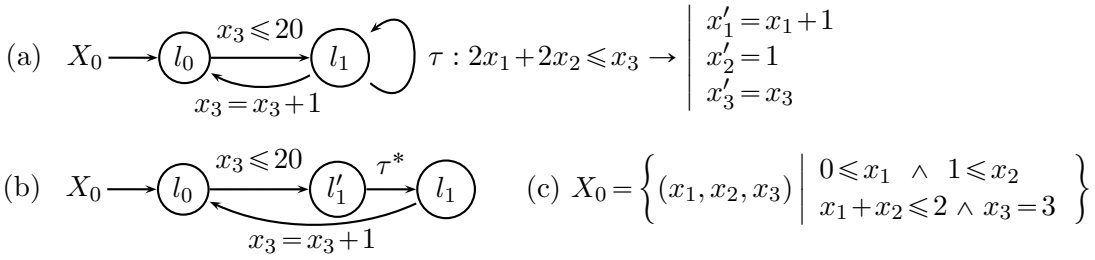


Figure 4.1: Example program (a), transformed program (b) where τ^* denotes the reflexive and transitive closure of the transition τ, and set of initial states (c).

tions) one resorts to the use of widening to guarantee the convergence of the fixed point computation at the cost of over-approximations.

After recalling in §4.1 some concepts of linear algebra used in the following chapters, we present classical exact acceleration in §4.2 and abstract acceleration in §4.3.

4.1 Introduction to Linear Algebra

In this section we recall some basic concepts of linear algebra. For further details we refer to textbooks on matrix theory and linear algebra, *e.g.*, [LT84].

A matrix $\mathbf{A} \in \mathbb{R}^{n\times m}$ has n rows and m columns. The element in the i^{th} row and j^{th} column is denoted $\mathbf{A}_{i,j}$. The i^{th} row vector and the j^{th} column vector are denoted respectively $\mathbf{A}_{i,\cdot}$ and $\mathbf{A}_{\cdot,j}$.

Square matrices. A square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, *i.e.*, of *dimension* $dim(\mathbf{A}) = n$, is called
- a *diagonal* matrix $\mathbf{A} = diag(a_1, \ldots, a_n)$ iff $\mathbf{A}_{i,i} = a_i$ and all other entries are zero;
- the *unit* (or *identity*) matrix, denoted $\mathbf{I}$, iff $\mathbf{A} = diag(1, \ldots, 1)$;
- the *zero* matrix $\mathbf{0}$ iff all entries equal zero.
- *idempotent* iff $\mathbf{A}^2 = \mathbf{A}$;
- *nilpotent* iff $\exists p > 0 : \mathbf{A}^p = \mathbf{0}$.
- *invertible* iff $\exists \mathbf{B} : \mathbf{AB} = \mathbf{BA} = \mathbf{I}$; $\mathbf{B} = \mathbf{A}^{-1}$ is called the inverse matrix; non-invertible matrices are called *singular*.

The *determinant* of $\mathbf{A}$ is defined by $det(\mathbf{A}) = \sum_{\boldsymbol{j}} (-1)^{t(\boldsymbol{j})} \mathbf{A}_{1,j_1} \cdot \ldots \cdot \mathbf{A}_{n,j_n}$ where $t(\boldsymbol{j})$ is the number of inversions in the permutation of indices $(j_1, \ldots, j_n)$ and the sum runs over all $n!$ permutations $(j_1, \ldots, j_n)$. The determinant of a 2-dimensional matrix is for example $\mathbf{A}_{1,1}\mathbf{A}_{2,2} - \mathbf{A}_{1,2}\mathbf{A}_{2,1}$.

Vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ are *linearly independent* iff $\sum_{i=1}^{n} c_i \boldsymbol{a}_i = \mathbf{0} \Longrightarrow \forall 1 \leqslant i \leqslant n : c_i = 0$. $rank(\mathbf{A})$ denotes the number of linearly independent column vectors, which is equal to the number of linearly independent row vectors.

Linear spaces. $\langle \mathbb{V}, +, \mathbb{F} \rangle$ is a *linear space* over a field $\mathbb{F}$ iff $\langle \mathbb{V}, + \rangle$ is a commutative group and there is for each $\alpha \in \mathbb{F}$ and for each $\boldsymbol{v} \in \mathbb{V}$ an associative and distributive scalar multiplication $\alpha \cdot \boldsymbol{v} \in \mathbb{V}$. We will consider linear spaces $\langle \mathbb{R}^n, +, \mathbb{R} \rangle$ or $\langle \mathbb{C}^n, +, \mathbb{C} \rangle$.

A *linear subspace* is a subset $S \subseteq \mathbb{V}$ closed under the operations (vector addition $+$ and scalar multiplication $\cdot$). The *kernel* (or null space) $ker(\mathbf{A})$ of a matrix is the linear space defined by $\{\boldsymbol{x} \mid \mathbf{A}\boldsymbol{x} = \mathbf{0}\}$. The *image* (or range) $im(\mathbf{A})$ is the linear space $\{\boldsymbol{x}' \mid \boldsymbol{x}' = \mathbf{A}\boldsymbol{x}\}$.

The *linear hull* denoted $span\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$ is the linear space generated by linear combinations of the vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, *i.e.*, $\{\boldsymbol{a} \mid \boldsymbol{a} = \sum_{i=1}^{n} c_i \boldsymbol{a}_i\}$.

A minimal number of vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ spanning a linear space are called a *basis* of the linear space. The number n of the basis vectors equals the dimension of the space.

Two bases $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ and $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n$ are related by an invertible *transition matrix* $\mathbf{P}$ such that $\forall i : \boldsymbol{b}_i = \mathbf{P}\boldsymbol{a}_i$. Thus any vector $\boldsymbol{x}$ w.r.t. the first basis can be transformed into a vector $\mathbf{P}\boldsymbol{x}$ w.r.t. the second basis. This transformation is called the *change of basis*.

A linear space L can be decomposed into a *direct sum* of subspaces $\dot{+}_i L_i$ iff $\bigcap_i L_i = \mathbf{0}$, *i.e.*, iff the union of basis vectors of the L_i forms a basis of the sum space L.

Linear transformations. A linear transformation is a homomorphism between n and m-dimensional linear spaces L_1 and L_2.

We will only deal with endomorphisms, *i.e.*, a linear transformation from an n-dimensional linear space to itself. Such a transformation can be represented by a square matrix $\mathbf{A}$ with dimension n. $\mathbf{A}$ is not invariant w.r.t. the basis of the linear space. The basis of the transformation matrix can be changed by the transition matrix $\mathbf{Q}$: $\mathbf{A}' = \mathbf{Q}^{-1}\mathbf{A}\mathbf{Q}$. The set of all representations $\mathbf{A}$ w.r.t. all bases is an equivalence class of *similar* matrices.

L' is an *invariant subspace* of $\mathbf{A}$ if it satisfies $\forall \boldsymbol{x} \in L' : \mathbf{A}\boldsymbol{x} \in L'$. If a linear space L can be decomposed into a direct sum of invariant subspaces L_i w.r.t. $\mathbf{A}$, then $\mathbf{A}$ has a representation in *block diagonal form* $\begin{pmatrix} \mathbf{A}_1 & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{A}_k \end{pmatrix}$ s.t. a block $\mathbf{A}_i$ corresponds to L_i.

Eigenvalues. The set $eig(\mathbf{A}) = \{\lambda \mid \exists \boldsymbol{x} \neq \mathbf{0} : \mathbf{A}\boldsymbol{x} = \lambda\boldsymbol{x}\}$ is the *spectrum* of $\mathbf{A} \in \mathbb{R}^{n\times n}$ (or $\mathbb{C}^{n\times n}$). The values $\lambda \in \mathbb{C}$ and the vectors $\boldsymbol{x} \in \mathbb{C}^n$ are called respectively the *eigenvalues* and *eigenvectors* of $\mathbf{A}$. An n-dimensional matrix has at most n distinct eigenvalues. Eigenvectors corresponding to distinct eigenvalues are linearly independent. The *eigenspace* associated with an eigenvalue is the linear hull of the eigenvectors. Similar matrices have the same spectrum. If λ is an eigenvalue of $\mathbf{A}$ then λ^p is an eigenvalue of $\mathbf{A}^p$. A matrix is singular iff it has a zero eigenvalue.

An alternative characterization of the eigenvalues is $(\mathbf{A} - \lambda\mathbf{I})\boldsymbol{x} = \mathbf{0}$ with $\boldsymbol{x} \neq \mathbf{0}$. Hence, the eigenvalues can also be seen as the zeros in $\mathbb{C}$ of the *characterstic polynomial* (in λ) $det(\mathbf{A} - \lambda\mathbf{I})$. The *algebraic multiplicity* of an eigenvalue is its multiplicity as a root of the characteristic polynomial. The *geometric multiplicity* of an eigenvalue is the dimension of its associated eigenspace: $dim(ker(\mathbf{A} - \lambda\mathbf{I}))$. The geometric multiplicity is always less than or equal to the algebraic multiplicity.

If $\mathbf{A}$ has n distinct eigenvalues then the eigenvectors form a basis (*eigenbasis*): we can change the basis of $\mathbf{A}$ with a transition matrix $\mathbf{Q} \in \mathbb{C}^{n\times n}$ of which the column vectors are the eigenvectors: $\mathbf{Q}^{-1}\mathbf{A}\mathbf{Q} = diag(\lambda_1, \dots, \lambda_n)$, *i.e.*, $\mathbf{A}$ is then called diagonizable.

Jordan canonical form. The Jordan canonical form $\mathbf{J} \in \mathbb{C}^{n\times n}$ is the unique decomposition of a linear transformation $\mathbf{A} \in \mathbb{R}^{n\times n}$ (or $\mathbb{C}^{n\times n}$) into a direct sum of k linear transformations, *i.e.*, $\mathbf{J}$ is a block diagonal matrix consisting of k blocks. Each block is associated with an eigenvalue λ_i and has the form $\begin{pmatrix} \lambda_i & 1 & 0 & \ddots \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & \lambda_i & 1 \\ 0 & \dots & 0 & \lambda_i \end{pmatrix}$. The number of Jordan blocks associated with the same eigenvalue equals its geometric multiplicity; the sum of the sizes of these blocks is its algebraic multiplicity.

The *generalized eigenspace* associated to an eigenvalue with algebraic multiplicity m is $ker((\mathbf{A} - \lambda\mathbf{I})^m)$. There are m generalized eigenvectors forming a basis of the generalized eigenspace. The matrix where the column vectors are the generalized eigenvectors of $\mathbf{A}$ is a transition matrix $\mathbf{Q} \in \mathbb{C}^{n\times n}$ such that $\mathbf{J} = \mathbf{Q}^{-1}\mathbf{A}\mathbf{Q}$.

For real matrices, complex eigenvalues are always *conjugate*, that is of the form $\rho e^{\pm i\theta}$ with $\rho, \theta \in \mathbb{R}$. In this case there is a *real Jordan form* in which conjugate

complex eigenvalues are associated with a block of the form $\begin{pmatrix} \mathbf{A} & \mathbf{I} & \mathbf{0} \\ \vdots & \ddots & \mathbf{I} \\ \mathbf{0} & \dots & \mathbf{A} \end{pmatrix}$ with $\mathbf{A} = \begin{pmatrix} \rho\cos\theta & -\rho\sin\theta \\ \rho\sin\theta & \rho\cos\theta \end{pmatrix}$.

4.2 Acceleration

Acceleration methods consider the program model of counter systems, but they have also been shown to be applicable to pushdown systems and systems of FIFO channels for instance (cf. [BFLS05]). The acceleration of counter systems is based on Presburger arithmetic.

Presburger arithmetic. Presburger arithmetic [Pre30] is the first-order additive theory over integers $\langle \mathbb{Z}, \leqslant, + \rangle$. Satisfiability and validity are decidable in this theory. A set is *Presburger-definable* if it can be described by a Presburger formula. For example, the set of odd natural numbers x can be defined by the Presburger formula $\exists k \geqslant 0 : x = 1 + 2k$, whereas for example the formula $\exists k : y = k \cdot k$ characterizing the quadratic numbers y is not Presburger because of the multiplication of variables.

Counter systems. Counter systems are a subclass of discrete transition systems (cf. Def. 2.3):

Definition 4.1 (Counter system) *A counter system $\langle \Sigma, L, \rightsquigarrow, \Sigma^0 \rangle$ is defined by*
- *the state space $\Sigma = \mathbb{Z}^n$,*
- *a set of locations L,*
- *a set of transitions $\rightsquigarrow$: $L \times \Sigma^2 \times L$, where transitions are labeled with a relation $R(\boldsymbol{x}, \boldsymbol{x}') \subseteq \Sigma^2$ defined by a Presburger formula, and*
- *$\Sigma^0 : L \to \Sigma$ defines for each location the set of initial states using a Presburger formula.*

Counter systems generalize Minsky machines [Min67], thus the reachability problem is undecidable. In general, the reachable set reach of a counter system is not Presburger-definable [KPRS96] because of the following two reasons:

(1) In the case where the system consists of a single self-loop and its transition relation R is Presburger-definable, then the reflexive and transitive closure R^* is not Presburger-definable in general.

(2) In the case of a system with nested loops where the reflexive and transitive closures R^* of all circuits in the system are Presburger-definable, reach of the whole system is not Presburger-definable in general, because there are infinitely many possible sequences of these circuits.

Issue (1) is addressed by identifying a class of *accelerable* relations R, *i.e.*, for which the transitive closure R^* is Presburger-definable:

Definition 4.2 (Presburger-linear relations with finite monoid) *The transition relation $R(\boldsymbol{x}, \boldsymbol{x}') = (\varphi(\boldsymbol{x}) \wedge \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d})$ is Presburger-linear with finite monoid iff φ is a Presburger formula and $\langle C^*, \cdot \rangle$ is a finite, multiplicative monoid,* i.e., *the set $C^* = \{\mathbf{C}^k \mid k \geqslant 0\}$ is finite.*

Theorem 4.1 (Presburger-definable transitive closure) *If R is a Presburger-linear relation with finite monoid, then R^* is Presburger-definable [Boi98, FL02].*

The finiteness of the monoid is polynomially decidable [Boi98]. The tool LASH [Boi98, WB98] implements these results.

Example 4.1 (Translation) *An example for transition relation of which the transitive closure is Presburger-definable are* translations*: $R(\boldsymbol{x}, \boldsymbol{x}') = (\varphi(\boldsymbol{x}) \wedge \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d})$. The variables are translated in each iteration by a constant vector $\boldsymbol{d}$. A translation is trivially finite monoid because $\mathbf{C} = \mathbf{I}$. The transitive closure is given by the Presburger formula:*

$$R^*(\boldsymbol{x}, \boldsymbol{x}') = \exists k \geqslant 0 : \boldsymbol{x}' = \boldsymbol{x} + k\boldsymbol{d} \wedge \forall k' \in [0, k-1] : \varphi(\boldsymbol{x} + k'\boldsymbol{d})$$

Issue (2) is adressed by the concept of flat acceleration:

Flat systems. A system is called *flat* if it has no nested loops, or more precisely if any location of the system is contained in at most one elementary cycle of the system [BFLP03]. This notion allows us to identify a class of systems for which the set of reachable states can be computed exactly:

Theorem 4.2 (Presburger-definable flat systems) *The reachability set* reach *of a counter system is Presburger-definable if the system is flat and all its transitions are Presburger-linear relations with finite monoid [FL02].*

Although there are many practical examples of flat systems [LS05], most systems are non-flat (like Fig. 4.1).

Application to non-flat systems. The idea of Finkel et al [FL02, BFLS05] is to partially unfold the outer loops (circuits) of nested loops in order to obtain a flat system that is simulated by the original system. Such a system is called a *flattening*.

The algorithm is based on heuristically selecting circuits of increasing length and enumerating flattenings of these circuits. Hence, the algorithm terminates in case the system is *flattable*, *i.e.*, at least one of its (finite) flattenings has the same reachability set as the original system.

Theorem 4.3 (Reachability in flattable systems) *The set of reachable states* reach *of flattable systems can be computed exactly [BFLS05].*

However, flattability is undecidable [BFLS05]. All these techniques are implemented in the tool FAST [BFLP03, Ler03, BFL04, BFLS05, BFLP08].

Acceleration of relations. It has been shown that the transitive closure of difference bounds constraints [CJ98] and octagonal relations [BGI09] is also Presburger-definable. A performant algorithm for accelerating such relations is implemented in the tool FLATA [BIK10].

4.3 Abstract Acceleration

Abstract acceleration introduced by Gonnord et al [GH06, Gon07] reformulates acceleration concepts within an abstract interpretation approach: it aims at computing the best correct approximation of the effect of loops in the abstract domain of convex polyhedra.

The objective of abstract acceleration is to over-approximate the set $\tau^*(X)$, $X \subseteq \mathbb{R}^n$ by a convex polyhedron $\gamma(\tau^{\otimes}(\alpha(X))) \supseteq \tau^*(X)$ that is "close" to the convex hull of the exact set.

We will focus on the acceleration of self-loops τ, but the technique can also deal with cycles by composing transitions.

Accelerable transitions. Abstract acceleration targets the model of affine counter automata, *i.e.*, with transition relations τ (actually functions) of the form of guarded actions $G \to A$, meaning "while guard G do action A":

$$\underbrace{\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}}_{\text{guard}} \to \underbrace{\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}}_{\text{action}} \tag{4.1}$$

where $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}$ is a convex polyhedron, and $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ is an affine transformation.

We will use the same notation for polyhedra X interexchangeably for both the predicate $X(\boldsymbol{x}) = (\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b})$ and the set $X = \{\boldsymbol{x} \mid \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}\}$.

Abstract acceleration distinguishes different types of transitions depending on the form of the square matrix $\mathbf{C}$:
- *Resets* iff $\mathbf{C}$ is the zero matrix.
- *Translations* iff $\mathbf{C}$ is the identity matrix.
- *Translations with resets* (or translation/reset) iff $\mathbf{C}$ is a diagonal matrix with zeros and ones only.
- *Periodic affine transformations* iff $\exists p > 0, \exists l > 0 : \mathbf{C}^{p+l} = \mathbf{C}^p$.
- *General affine transformations* otherwise.

We observe that Presburger-linear relations with finite monoid (Def. 4.2) include the first four cases, which are thus considered accelerable.

We will now explain how to accelerate these types of transitions:

Theorem 4.4 (Translations) *Let τ be a translation $G \to \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d}$, then for every convex polyhedron X, the convex polyhedron*

$$\tau^{\otimes}(X) = X \sqcup \Big(\big((X \sqcap G) \nearrow \{\boldsymbol{d}\} \big) \sqcap (G + \{\boldsymbol{d}\}) \Big)$$

is a convex over-approximation of $\tau^*(X)$ [Gon07].

Observe that the abstract acceleration of translations uses the time elapse operator $\nearrow$ for polyhedra, originally considered in the analysis of timed or hybrid automata (see §3.3.2).

Example 4.2 (Translation) *(see Fig. 4.2)*

$$\tau : \underbrace{x_1 + x_2 \leqslant 4 \ \wedge \ x_2 \leqslant 3}_{G} \to \begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \underbrace{\begin{pmatrix} 2 \\ 1 \end{pmatrix}}_{\boldsymbol{d}}$$

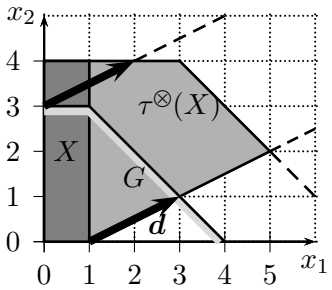


Figure 4.2: Abstract acceleration of a *translation* (Ex. 4.2) by vector $\boldsymbol{d}$ starting from X (dark gray) resulting in $\tau^{\otimes}(X)$ (whole shaded area).

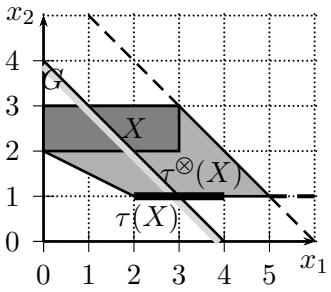


Figure 4.3: Abstract acceleration of a *translation with resets* (Ex. 4.3) starting from X (dark gray): $\tau(X)$ (bold line) and result $\tau^{\otimes}(X)$ (whole shaded area).

Starting from $X = (0 \leqslant x_1 \leqslant 1 \ \wedge \ 0 \leqslant x_2 \leqslant 4)$ *we compute* $\tau^{\otimes}(X)$*:*

$$\begin{array}{rcl} X \sqcap G & = & (0 \leqslant x_1 \leqslant 1 \ \wedge \ 0 \leqslant x_2 \leqslant 3) \\ (X \sqcap G) \nearrow \{\boldsymbol{d}\} & = & \left\{ \begin{array}{l} x_1 \geqslant 0 \ \wedge \ x_2 \geqslant 0 \ \wedge \ x_1 - 2x_2 \geqslant -6 \ \wedge \\ -x_1 + 2x_2 \geqslant -1) \end{array} \right. \\ ((X \sqcap G) \nearrow \{\boldsymbol{d}\}) \sqcap (G + \{\boldsymbol{d}\}) & = & \left\{ \begin{array}{l} x_1 \geqslant 0 \ \wedge \ 0 \leqslant x_2 \leqslant 4 \ \wedge \ x_1 - 2x_2 \geqslant -6 \ \wedge \\ -x_1 + 2x_2 \geqslant -1 \ \wedge \ x_1 + x_2 \leqslant 7 \end{array} \right. \\ \tau^{\otimes}(X) & = & \left\{ \begin{array}{l} x_1 \geqslant 0 \ \wedge \ 0 \leqslant x_2 \leqslant 4 \ \wedge \ -x_1 + 2x_2 \geqslant -1 \ \wedge \\ x_1 + x_2 \leqslant 7 \end{array} \right. \end{array}$$

Remark 4.1 *Ideally,* $\tau^{\otimes}(X)$ *as defined in Thm. 4.4 should be the best over-approximation of* $\tau^*(X)$ *by a convex polyhedron. This is not the case as shown by the following example in one dimension. Let* $X = [1, 1]$ *and* $\tau : x_1 \leqslant 4 \rightarrow x_1' = x_1 + 2$*.* $\tau^{\otimes}(X) = [1, 6]$*, whereas the best convex over-approximation of* $\tau^*(X) = \{1, 3, 5\}$ *is the interval* $[1, 5]$*. This is because the operations involved in the definition of* $\tau^{\otimes}(X)$ *manipulate dense sets and do not take into account arithmetic congruences.*

Theorem 4.5 (Translations with resets) *Let* τ *be a translation with resets* $G \rightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$*, then for every convex polyhedron* X*, the convex polyhedron*

$$\tau^{\otimes}(X) = X \sqcup \tau(X) \sqcup \Big(\big((\tau(X) \sqcap G) \nearrow \{\mathbf{C}\boldsymbol{d}\} \big) \sqcap (G + \{\mathbf{C}\boldsymbol{d}\}) \Big)$$

is a convex over-approximation of $\tau^*(X)$ *[Gon07].*

Example 4.3 (Translation with resets) *(see Fig. 4.3)*

$$\tau : x_1 + x_2 \leqslant 4 \ \wedge \ \left\{ \begin{array}{l} x_1' = x_1 + 2 \\ x_2' = 1 \end{array} \right.$$

Starting from $X = (0 \leqslant x_1 \leqslant 3 \ \wedge \ 2 \leqslant x_2 \leqslant 3)$ *we compute* $\tau^{\otimes}(X)$*:*

$$\begin{array}{rcl} \tau(X) & = & (2 \leqslant x_1 \leqslant 4 \ \wedge \ x_2 = 1) \\ \tau(X) \sqcap G & = & (2 \leqslant x_1 \leqslant 3 \ \wedge \ x_2 = 1) \\ (\tau(X) \sqcap G) \nearrow \{\mathbf{C}\boldsymbol{d}\} & = & (x_1 \geqslant 2 \ \wedge \ x_2 = 1) \\ ((\tau(X) \sqcap G) \nearrow \{\mathbf{C}\boldsymbol{d}\}) \sqcap (G + \{\mathbf{C}\boldsymbol{d}\}) & = & (2 \leqslant x_1 \leqslant 5 \ \wedge \ x_2 = 1) \\ \tau^{\otimes}(X) & = & \left\{ \begin{array}{l} x_1 \geqslant 0 \ \wedge \ 1 \leqslant x_2 \leqslant 3 \ \wedge \ x_1 + 2x_2 \geqslant 4 \ \wedge \\ x_1 + x_2 \leqslant 6 \end{array} \right. \end{array}$$

Remark 4.2 *Thm. 4.5 exploits the property that a translation with resets to* constants *iterated N times is equivalent to the same translation with resets, followed by a pure translation iterated $N-1$ times. Hence the structure of the obtained formula.*

Periodic affine transformations. Let τ be a transition $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ such that the powers of $\mathbf{C}$ form a finite monoid (cf. Def. 4.2) with $\exists p > 0, \exists l > 0 : \mathbf{C}^{p+l} = \mathbf{C}^p$, *i.e.*, the powers of $\mathbf{C}$ generate an ultimately periodic sequence with prefix p and period l. Gonnord [Gon07] uses the periodicity condition $\exists q > 0 : \mathbf{C}^{2q} = \mathbf{C}^q$. This condition is equivalent to the one above with $q = lcm(p, l)$.

With the latter condition, τ^* can be rewritten by enumerating the transitions induced by the powers of $\mathbf{C}$:

$$\tau^*(X) = \bigcup_{0 \leqslant j \leqslant q-1} (\tau^q)^*(\tau^j(X)) \tag{4.2}$$

This means that one only has to know how to accelerate τ^q which equals:

$$\tau^q = \underbrace{\bigwedge_{0 \leqslant i \leqslant q-1} \left(\mathbf{A}\mathbf{C}^i\boldsymbol{x} + \sum_{0 \leqslant j \leqslant i-1} \mathbf{C}^j\boldsymbol{d} \leqslant \boldsymbol{b} \right)}_{\mathbf{A}'\boldsymbol{x} \leqslant \boldsymbol{b}'} \rightarrow \underbrace{\boldsymbol{x}' = \mathbf{C}^q\boldsymbol{x} + \sum_{0 \leqslant j \leqslant q-1} \mathbf{C}^j\boldsymbol{d}}_{\boldsymbol{x}' = \mathbf{C}'\boldsymbol{x} + \boldsymbol{d}'} \tag{4.3}$$

The periodicity condition above implies that $\mathbf{C}^q$ is diagonizable and all eigenvalues of $\mathbf{C}^q$ are in $\{0, 1\}$. Hence, τ^q is a *translation with resets in the eigenbasis of* $\mathbf{C}^q$. Let us denote $\mathbf{C}' = \mathbf{C}^q$:

Lemma 4.1 (Translation with resets in the eigenbasis) *A transition $\tau : \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \rightarrow \boldsymbol{x}' = \mathbf{C}'\boldsymbol{x} + \boldsymbol{d}$ where $\mathbf{C}' = \mathbf{C}^{'2}$ is a translation with resets τ' in the eigenbasis of $\mathbf{C}'$:*

$$\tau' : \mathbf{A}\mathbf{Q}\boldsymbol{x} \leqslant \boldsymbol{b} \rightarrow \boldsymbol{x}' = \mathbf{Q}^{-1}\mathbf{C}'\mathbf{Q}\boldsymbol{x} + \mathbf{Q}^{-1}\boldsymbol{d}$$

where $\mathbf{Q}^{-1}\mathbf{C}'\mathbf{Q} = diag(\lambda_1, \ldots, \lambda_n)$ and λ_i the eigenvalues of $\mathbf{C}'$.

We can now state the theorem for abstract acceleration of finite monoid transitions:

Theorem 4.6 (Finite monoid) *Let τ be a transition $G \wedge \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ where $\exists q > 0 : \mathbf{C}^{2q} = \mathbf{C}^q$, then for every convex polyhedron X, the convex polyhedron*

$$\tau^{\otimes}(X) = \bigsqcup_{0 \leqslant j \leqslant q-1} (\tau^q)^{\otimes}(\tau^j(X))$$

is a convex over-approximation of $\tau^*(X)$, where τ^q is defined by Eq. 4.3 and $(\tau^q)^{\otimes}$ is computed using Lem. 4.1 [Gon07].

Example 4.4 (Finite monoid) *(see Fig. 4.4)*

$$\tau : x_1 + x_2 \leqslant 4 \ \wedge \ \begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

We have $\mathbf{C}^2 = \mathbf{I} = \mathbf{C}^4$, thus $q = 2$. Obviously $\mathbf{C}^2$ has its eigenvalues in $\{0, 1\}$ and $\mathbf{Q} = \mathbf{I}$.

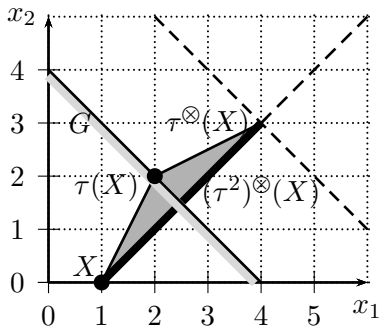


Figure 4.4: Abstract acceleration of a *finite monoid* (Ex. 4.4) starting from $X = \{(1,0)\}$, $(\tau^2)^{\otimes}(X)$ (bold line), $(\tau^2)^{\otimes}(\tau(X)) = \tau(X) = \{(2,2)\}$, and result $\tau^{\otimes}(X)$ (whole shaded area).

According to Eq. 4.3 we strengthen the guard by $\mathbf{A}(\mathbf{C}\boldsymbol{x} \leqslant \boldsymbol{b}) = (x_1 + x_2 \leqslant 1)$ *and compute* $\boldsymbol{d}' = (\mathbf{C}+\mathbf{I})\boldsymbol{d} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$. *Hence we get:*

$$\tau^2 : (x_1+x_2 \leqslant 1) \ \wedge \ \begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 3 \\ 3 \end{pmatrix}$$

Starting from $X = (x_1 = 1 \ \wedge \ x_2 = 0)$ *we compute* $\tau^{\otimes}(X) = X \sqcup (\tau^2)^{\otimes}(X) \sqcup (\tau^2)^{\otimes}(\tau(X))$*:*

$$\begin{array}{rcl} \tau(X) & = & (x_1 = x_2 = 2) \\ (\tau^2)^{\otimes}(X) & = & (1 \leqslant x_1 \leqslant 4 \ \wedge \ x_1 - x_2 = 1) \\ (\tau^2)^{\otimes}(\tau(X)) & = & (x_1 = x_2 = 2) \\ \tau^{\otimes}(X) & = & \left\{ \begin{array}{l} 2x_1 - x_2 \geqslant 2 \ \wedge \ x_1 - 2x_2 \geqslant -2 \ \wedge \\ x_1 - x_2 \leqslant 1 \end{array} \right. \end{array}$$

Widening and acceleration. Abstract acceleration gives us a formula for computing the transitive closure of accelerable transitions. Precision is gained in comparison to standard abstract interpretation relying only on Kleene iteration and widening to converge for the following reasons:

– Abstract acceleration aims at precisely approximating $\alpha \circ \tau^*(X)$, *i.e.*, abstracting the result of the transitive closure of the concrete transition, whereas standard abstract interpretation iterates the abstracted transition $(\alpha \circ \tau \circ \gamma)^* \circ \alpha(X)$. We will discuss this issue in §5.3.1.
– Abstract acceleration is *monotonic*, *i.e.*, $X_1 \sqsubseteq X_2 \Rightarrow \tau^{\otimes}(X_1) \sqsubseteq \tau^{\otimes}(X_2)$, whereas standard widening is not, which makes the analysis more robust and predictible.
– Widening is not needed at all in flat systems.

Yet, widening is required in the case of non-accelerable transitions, outer loops of nested loops, and to guarantee convergence when there are multiple self-loops in the same control location.

Further results. Some special cases of multiple self-loops can be accelerated without widening [GH06, Gon07]. The path focussing technique of Monniaux and Gonnord [MG11] is an improvement w.r.t. the abstract acceleration of cycles and nested loops.

Chapter 5

Extending Abstract Acceleration

In this chapter we extend abstract acceleration to systems with numerical inputs and to backward, *i.e.*, co-reachability, analysis.

Reactive programs such as Lustre programs (§2.2) interact with their environment: at each computation step, they have to take into account the values of *input* variables, which typically correspond to values acquired by sensors. Boolean input variables can be encoded in a CFG by finite non-deterministic choices, but numerical input variables require a more specific treatment. Indeed, they induce transitions of the form

$$\tau : g(\boldsymbol{x}, \boldsymbol{\xi}) \rightarrow \boldsymbol{x}' = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{\xi}) \qquad \boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^n \quad \boldsymbol{\xi} \in \mathbb{R}^p$$

that depend on both, numerical state variables $\boldsymbol{x}$ and numerical input variables $\boldsymbol{\xi}$.

A second aspect we are looking at is backward analysis (cf. §3.4.1): so far, abstract acceleration has only been defined to perform forward reachability analysis. Yet, applications of verification methods, like parameter synthesis, make use of backward analysis techniques. Moreover, the experience of verification tools, *e.g.*, [Jea03], shows that combining forward and backward analyses results in more powerful tools.

Outline. This chapter describes the following contributions (see Fig. 5.1):

1. We show how to extend abstract acceleration from closed to open systems, *i.e.*, systems with numerical *inputs* (§5.1).
2. We also extend abstract acceleration techniques from forward (reachability) analysis to *backward* (co-reachability) analysis (§5.2).
3. In §5.3 we *compare* the abstract acceleration approach with Kleene iteration, widening and the affine derivative closure algorithm of Ancourt et al. [ACI10], which is another abstract interpretation-based approach to computing transitive closures of transitions.

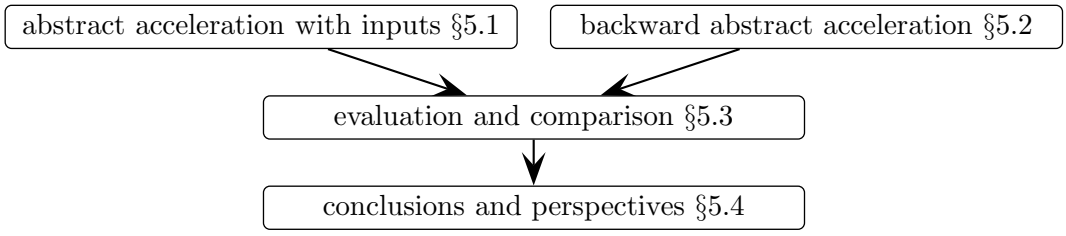


Figure 5.1: Chapter organization

5.1 Abstract Acceleration with Numerical Inputs

We consider transitions τ of the form[1]

$$\underbrace{\begin{pmatrix}\mathbf{A} & \mathbf{L}\\ \mathbf{0} & \mathbf{J}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} \leqslant \begin{pmatrix}\boldsymbol{b}\\ \boldsymbol{k}\end{pmatrix}}_{\mathbf{A}\boldsymbol{x}+\mathbf{L}\boldsymbol{\xi}\leqslant\boldsymbol{b}\ \wedge\ \mathbf{J}\boldsymbol{\xi}\leqslant\boldsymbol{k}} \rightarrow \boldsymbol{x}' = \underbrace{\begin{pmatrix}\mathbf{C} & \mathbf{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} + \boldsymbol{u}}_{\mathbf{C}\boldsymbol{x}+\mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}} \tag{5.1}$$

General and simple guards. The following proposition shows the challenge raised by adding inputs:

Proposition 5.1 (General guards and general affine transformations) *Any general affine transformation without inputs* $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \rightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ *can be expressed*
– as a "reset with inputs" $(\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{\xi} = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}) \rightarrow \boldsymbol{x}' = \boldsymbol{\xi}$,
– as well as a "translation with inputs" $(\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \boldsymbol{\xi} = (\mathbf{C}-\mathbf{I})\boldsymbol{x} + \boldsymbol{d}) \rightarrow \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\xi}$.

This means that there is no hope to get precise acceleration for such resets or translations with inputs, unless we know how to accelerate precisely general affine transformations without inputs, which is out of the scope of the current state of the art.

Nevertheless, we can accelerate transitions with inputs if the constraints on the state variables do not depend on the inputs, *i.e.*, the guard is of the form $\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}$, *i.e.*, when $\mathbf{L} = 0$ in Eqn. (5.1). We call the resulting guards *simple guards*.

In the following, we show how to accelerate translations and translations with resets with simple guards. We provide in §5.1.3 an over-approximation of our results for *general guards*. The extension of further cases such as periodic affine transformations will be discussed in §5.4.

5.1.1 Translations with Inputs and Simple Guards

Definition 5.1 *Translations with inputs and simple guards are defined as*

$$\underbrace{\begin{pmatrix}\mathbf{A} & \mathbf{0}\\ \mathbf{0} & \mathbf{J}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} \leqslant \begin{pmatrix}\boldsymbol{b}\\ \boldsymbol{k}\end{pmatrix}}_{\underbrace{\mathbf{A}\boldsymbol{x}\leqslant\boldsymbol{b}}_{G}\ \wedge\ \mathbf{J}\boldsymbol{\xi}\leqslant\boldsymbol{k}} \rightarrow \boldsymbol{x}' = \underbrace{\begin{pmatrix}\mathbf{I} & \mathbf{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} + \boldsymbol{u}}_{\boldsymbol{x}+\mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}}$$

The first step we perform is to reduce such a translation with inputs to a *polyhedral translation* $\tau : G \rightarrow \boldsymbol{x}' = \boldsymbol{x} + D$ and defined by $\tau(X) = (X \sqcap G) + D$.

Proposition 5.2 (Translation with inputs = polyhedral translation) *A translation τ with inputs and a simple guard (Def. 5.1) is equivalent to a polyhedral translation defined by*

$$G \rightarrow \boldsymbol{x}' = \boldsymbol{x} + D \quad \textit{with} \quad D = \{\boldsymbol{d} \mid \exists \boldsymbol{\xi} : \boldsymbol{d} = \boldsymbol{T}\boldsymbol{\xi} + \boldsymbol{u} \wedge \boldsymbol{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}\}$$

(D can be computed by standard polyhedra operations.)

[1] Note that the $\mathbf{0}$ in the matrix of the guard does not imply a loss of generality.

Proof

$$\begin{aligned}
& \boldsymbol{x}' \in \tau(X) \\
\iff\ & \exists \boldsymbol{x} \in X, \exists \boldsymbol{\xi}\ :\ \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \wedge \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k} \wedge \boldsymbol{x}' = \boldsymbol{x} + \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u} \\
\iff\ & \exists \boldsymbol{x} \in X \sqcap G, \exists \boldsymbol{\xi}, \exists \boldsymbol{d}\ :\ \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k} \wedge \boldsymbol{d} = \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u} \wedge \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d} \\
\iff\ & \exists \boldsymbol{x} \in X \sqcap G, \exists \boldsymbol{d} \in D\ :\ \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d} \\
& \text{with } D = \{\boldsymbol{d} \mid \exists \boldsymbol{\xi}\ :\ \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k} \wedge \boldsymbol{d} = \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u}\}
\end{aligned}$$

∎

We now generalize Thm. 4.4 from ordinary translations to polyhedral translations.

Proposition 5.3 (Polyhedral translation) *Let τ be a polyhedral translation $G \to \boldsymbol{x}' = \boldsymbol{x} + D$. Then, the set*

$$\tau^{\otimes}(X) = X \sqcup \tau\big((X \sqcap G) \nearrow D\big)$$

is a convex over-approximation of $\tau^*(X)$.

Proof $\boldsymbol{x}' \in \bigcup_{k\geqslant 1} \tau^k(X) \iff \boldsymbol{x}' \in \tau(\bigcup_{k\geqslant 0} \tau^k(X))$

$$\iff \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_k, \exists \boldsymbol{d}_1 \dots \boldsymbol{d}_k \in D\ :\ \begin{cases} \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \\ \boldsymbol{x}_k = \boldsymbol{x}_0 + \sum_{j=1}^{k} \boldsymbol{d}_j \\ G(\boldsymbol{x}_0)\ \wedge\ \forall k' \in [1,k]\ :\ G(\boldsymbol{x}_0 + \sum_{j=1}^{k'} \boldsymbol{d}_j) \end{cases}$$

$$\iff \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_k, \exists \boldsymbol{d} \in D\ :\ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k)\ \wedge\ \boldsymbol{x}_k = \boldsymbol{x_0} + k\boldsymbol{d}\ \wedge\ G(\boldsymbol{x}_0)\ \wedge\ G(\boldsymbol{x}_k)$$

(because D and G are convex, see Rem. 5.1)

$$\implies \exists \alpha \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_k, \exists \boldsymbol{d} \in D\ :\ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k)\ \wedge\ \boldsymbol{x}_k = \boldsymbol{x}_0 + \alpha\boldsymbol{d}\ \wedge\ G(\boldsymbol{x}_0)$$

(dense approximation; $G(\boldsymbol{x}_k)$ implied by $\boldsymbol{x}' \in \tau(\boldsymbol{x}_k)$)

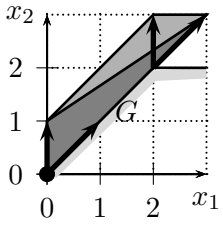
$$\iff \exists \boldsymbol{x}_0 \in X \sqcap G, \exists \boldsymbol{x}_k\ :\ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k)\ \wedge\ \boldsymbol{x}_k \in (\{\boldsymbol{x}_0\} \nearrow D)$$

$$\iff \boldsymbol{x}' \in \tau((X \sqcap G) \nearrow D)$$

∎

Mind that the only approximation takes place in the line ($\Rightarrow$) where the integer coefficient $k \geqslant 0$ is replaced by a real coefficient $\alpha \geqslant 0$. This is the technical explanation of Rem. 4.1.

Remark 5.1 (Convexity argument) *For any k loop iterations with $\boldsymbol{d}_1, \dots, \boldsymbol{d}_k \in D$ s.t. $\forall k' \in [0,k]$: $G(\boldsymbol{x} + \sum_{j=1}^{k'} \boldsymbol{d}_j)$ we have $\exists \boldsymbol{d} \in D, \exists \alpha \geqslant 0$: $G(\boldsymbol{x}) \wedge G(\boldsymbol{x} + \alpha\boldsymbol{d})$ s.t. $\sum_{j=1}^{k} \boldsymbol{d}_j = \alpha\boldsymbol{d}$: any intermediate point $\boldsymbol{x} + \alpha\boldsymbol{d}$ must be in G, because G is convex; moreover $\alpha \geqslant 0$ and a vector $\boldsymbol{d} \in D$ actually exist: for example, take $\alpha = k$ and $\boldsymbol{d} = \frac{1}{k}\sum_{j=1}^{k} \boldsymbol{d}_j$, which is in D because D is convex.*



Remark 5.2 (Inputs vs. constants) *One might think that Thm. 4.4 can be applied directly by accelerating the transition for each $\boldsymbol{d} \in D$ and taking the union, i.e. computing $\tau^{\otimes}(X)$ by $X \sqcup \bigsqcup_{\boldsymbol{d}\in D} X_{\boldsymbol{d}}$ with $X_{\boldsymbol{d}} = \big((X \sqcap G) \nearrow \{\boldsymbol{d}\}\big) \sqcap (G + \{\boldsymbol{d}\})$. However, this formula is not correct for the last step beyond the guard, which is illustrated in the figure on the right-hand side for a polyhedral translation with $X = \{(0,0)\}$, $D = (0 \leqslant d_1 \leqslant 1 \wedge d_2 = 1)$ and the guard G given in the figure. For the step crossing the guard,* e.g., *at $(2,2)$, there is actually a choice among all values in D (correct abstract acceleration: whole shaded area), whereas the wrong acceleration (dark gray) considers only the vector $\boldsymbol{d} = (1,1)$ which led to $(2,2)$.*

We combine Propositions 5.2 and 5.3 to formulate the following theorem:

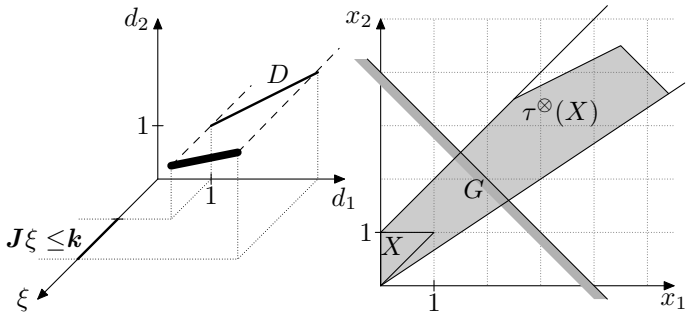


Figure 5.2: Translation with inputs (Ex. 5.1): The left-hand side shows the computation of D: $\mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k} \wedge \boldsymbol{d} = \mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}$ (bold line) is projected on variables $\boldsymbol{d}$ to obtain D. The shaded area in the right-hand side figure is $\tau^{\otimes}(X)$.

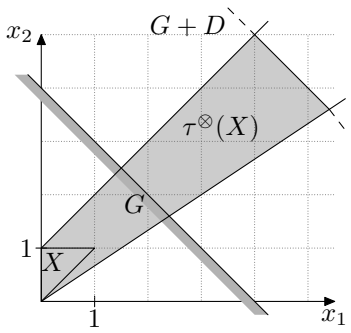


Figure 5.3: Precision loss in example 5.1 when using the approximate formula according to Rem. 5.3.

Theorem 5.1 (Translation with inputs and simple guards) *Let τ be a translation with inputs and a simple guard*

$$\tau : (\underbrace{\mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}}_{G}) \wedge (\mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}) \quad \rightarrow \quad \boldsymbol{x}' = \boldsymbol{x} + \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u}$$

Then, the set

$$\tau^{\otimes}(X) = X \sqcup \tau\big((X \sqcap G) \nearrow D\big)$$

with $D = \{\boldsymbol{d} \mid \exists \boldsymbol{\xi} : \boldsymbol{d} = \boldsymbol{T}\boldsymbol{\xi} + \boldsymbol{u} \wedge \boldsymbol{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}\}$ is a convex over-approximation of $\tau^*(X)$.

Proof Follows directly from Prop. 5.2 and 5.3.

Example 5.1 (Translation with inputs and simple guards) *Consider the polyhedron $X = \{(x_1, x_2) \mid 0 \leqslant x_1 \leqslant x_2 \leqslant 1\}$ and the transition*

$$\tau : \left| \begin{array}{rcl} x_1 + x_2 & \leqslant & 4 \\ 1 \leqslant \xi & \leqslant & 2 \end{array} \right. \rightarrow \left| \begin{array}{l} x_1' = x_1 + 2\xi - 1 \\ x_2' = x_2 + \xi \end{array} \right.$$

Eliminating the inputs as in Proposition 5.2 yields $D = \{(d_1, d_2) \,|\, 1 \leqslant d_1 \leqslant 3 \wedge -d_1 + 2d_2 = 1\}$, see Fig. 5.2 left-hand side. After translation of X by D (Fig. 5.2 right-hand side) we obtain the polyhedron $\{(x_1, x_2) \,|\, x_1 \geqslant 0 \wedge -x_1 + x_2 \leqslant 1 \wedge x_1 + x_2 \leqslant 9 \wedge -2x_1 + 4x_2 \leqslant 9 \wedge 2x_1 - 3x_2 \leqslant 0\}$.

Remark 5.3 (Alternative, less precise formula) *In analogy to Thm. 4.4, we could alternatively consider the formula*

$$X \sqcup ((((X \sqcap G) \nearrow D) \sqcap (G + D)).$$

In order to justify this, we extend the proof of Proposition 5.3 by continuing at the label (dense approximation):

$\iff \exists \alpha \geqslant 0, \exists \boldsymbol{x}_0 \in X \sqcap G, \exists \boldsymbol{x}_k, \exists \boldsymbol{d}, \boldsymbol{d}' \in D : \boldsymbol{x}' = \boldsymbol{x}_k + \boldsymbol{d}' \ \wedge \ \boldsymbol{x}_k = \boldsymbol{x}_0 + \alpha \boldsymbol{d} \ \wedge \ G(\boldsymbol{x}_k)$

$\iff \exists \alpha \geqslant 0, \exists \boldsymbol{x}_0 \in X \sqcap G, \exists \boldsymbol{d}, \boldsymbol{d}' \in D : \boldsymbol{x}' = \boldsymbol{x}_0 + \alpha \boldsymbol{d} + \boldsymbol{d}' \ \wedge \ G(\boldsymbol{x}' - \boldsymbol{d}')$

$\implies \big(\exists \alpha \geqslant 0, \exists \boldsymbol{x}_0 \in X \sqcap G, \exists \boldsymbol{d}, \boldsymbol{d}' \in D : \boldsymbol{x}' = \boldsymbol{x}_0 + \alpha \boldsymbol{d} + \boldsymbol{d}'\big) \ \wedge \ \big(\exists \boldsymbol{d}' \in D : G(\boldsymbol{x}' - \boldsymbol{d}')\big)$

$\iff \big(\exists \alpha' \geqslant 1, \exists \boldsymbol{x}_0 \in X \sqcap G, \exists \boldsymbol{d}'' \in D : \boldsymbol{x}' = \boldsymbol{x}_0 + \alpha' \boldsymbol{d}''\big) \ \wedge \ \big(\exists \boldsymbol{d}' \in D : G(\boldsymbol{x}' - \boldsymbol{d}')\big)$

$\implies \boldsymbol{x}' \in (X \sqcap G) \nearrow D \ \wedge \ \boldsymbol{x}' \in (G + D)$

using $\{\boldsymbol{x} \mid \exists \boldsymbol{d} \in D \ \wedge \ G(\boldsymbol{x} - \boldsymbol{d})\} = \{\boldsymbol{z} + \boldsymbol{d} \mid \boldsymbol{d} \in D \wedge G(\boldsymbol{z})\} = (G + D)$. It can be

observed that for the translation of example 5.1 the latter formula results in an over-approximation (see Fig. 5.3) as compared to the result in Fig. 5.2. This reflects the additional approximation steps in the proof indicated by ($\Longrightarrow$).

5.1.2 Translations/Resets with Inputs and Simple Guards

Definition 5.2 *Translations/resets with inputs and simple guards are defined as*

$$\underbrace{\begin{pmatrix}\mathbf{A} & \mathbf{0}\\ \mathbf{0} & \mathbf{J}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} \leqslant \begin{pmatrix}\boldsymbol{b}\\ \boldsymbol{k}\end{pmatrix}}_{\mathbf{A}\boldsymbol{x}\leqslant\boldsymbol{b}\ \wedge\ \mathbf{J}\boldsymbol{\xi}\leqslant\boldsymbol{k}} \rightarrow \boldsymbol{x}' = \underbrace{\begin{pmatrix}\mathbf{C} & \mathbf{T}\end{pmatrix}\begin{pmatrix}\boldsymbol{x}\\ \boldsymbol{\xi}\end{pmatrix} + \boldsymbol{u}}_{\mathbf{C}\boldsymbol{x}+\mathbf{T}\boldsymbol{\xi}+\boldsymbol{u}}$$

where $\mathbf{C}$ *is a diagonal matrix with* $\mathbf{C}_{i,i} \in \{0,1\}$ *for all* i.

Notations. Let $\mathbf{C}' = \mathbf{I} - \mathbf{C}$, then we can decompose any vector $\boldsymbol{x}$ in $\mathbf{C}\boldsymbol{x}+\mathbf{C}'\boldsymbol{x}$. We denote $\mathbf{C}\boldsymbol{x} = \boldsymbol{x}^{t,0}$ a vector where the reset dimensions are set to zero, and $\mathbf{C}'\boldsymbol{x} = \boldsymbol{x}^{0,r}$ a vector where the translated dimensions are set to zero. We extend such notations to sets: $X^{t,0} = \{\boldsymbol{x}^{t,0} \mid \boldsymbol{x} \in X\}$ and $X^{0,r} = \{\boldsymbol{x}^{0,r} \mid \boldsymbol{x} \in X\}$. We use a similar notation for projection: $X^{t,\bullet} = \{\boldsymbol{x} \mid \boldsymbol{x}^{t,0} \in X^{t,0}\}$ and $X^{\bullet,r} = \{\boldsymbol{x} \mid \boldsymbol{x}^{0,r} \in X^{0,r}\}$ denote the sets obtained by existential quantification of the reset (resp. translated) dimensions.

Observe that the over-approximation $X^{t,\bullet} \sqcap X^{\bullet,r}$ of a set X by the cartesian product w.r.t. to translated and reset dimensions is equal to the Minkowski sum $X^{t,0} + X^{0,r}$.

The case of translations/resets with inputs can be handled similarly to translations: we combine Proposition 5.2 and Thm. 4.5 to reduce translations/resets with inputs to *polyhedral translations with resets* $\tau : G \rightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x}+D$ defined by $\tau(X) = (X \sqcap G)^{t,0} + D$.

Mind, however, that Rem. 4.2 does not apply any more and cannot be exploited in the presence of inputs, because the variables being reset may be assigned a different value in each iteration.

Proposition 5.4 (Polyhedral translation with resets) *Let* τ *be a polyhedral translation with resets* $G \rightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + D$. *Then, the set*

$$\tau^{\otimes}(X) = X \sqcup \tau(X) \sqcup \tau\Big(\big((\tau(X) \sqcap G)^{t,0} \nearrow D^{t,0}\big) + D^{0,r}\Big)$$

is a convex over-approximation of $\tau^*(X)$.

In the formula above and in the proof below, we unfold τ twice, that is, we accelerate only the central part of the sequence $\boldsymbol{x} \xrightarrow{\tau} \boldsymbol{x}_0 \dots \boldsymbol{x}_n \xrightarrow{\tau} \boldsymbol{x}'$ with $\boldsymbol{x} \in X$ because we have $\forall k \in [0,n] : \boldsymbol{x_k} \in G \sqcap D^{\bullet,r}$, whereas we only have $\boldsymbol{x} \in G$ at the start-point, and $\boldsymbol{x}' \in D^{\bullet,r}$ at the end-point.

Proof The formula is trivially correct for 0 or 1 iterations of the self-loop τ. It remains to show that, for the case of $k \geqslant 2$ iterations, our formula yields an over-approximation of $\bigcup_{k\geqslant 2} \tau^k(X)$.

$$
\begin{aligned}
&\boldsymbol{x}' \in \textstyle\bigcup_{k\geqslant 2} \tau^k(X) \iff \boldsymbol{x}' \in \tau\left(\bigcup_{k\geqslant 0} \tau^k(\tau(X))\right)\\
&\iff \exists k \geqslant 0, \exists \boldsymbol{x} \in X, \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1 \dots \boldsymbol{d}_k \in D \ :\\
&\quad \left\{ \begin{array}{lll} & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) & \\ \wedge & \forall k' \in [1,k] \ : & \left\{ \begin{array}{ll} \boldsymbol{x}^i_{k'} = \boldsymbol{x}^i_0 + \sum_{j=1}^{k'} \boldsymbol{d}^i_j & \text{for } i \in I^t \\ \boldsymbol{x}^i_{k'} = \boldsymbol{d}^i_{k'} & \text{for } i \in I^r \end{array} \right. \\ \wedge & \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) & \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) & \end{array} \right.\\
&\iff \exists k \geqslant 0, \exists \boldsymbol{x} \in X, \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1 \dots \boldsymbol{d}_k \in D \ :\\
&\quad \left\{ \begin{array}{lll} & \forall k' \in [1,k] \ : \ \boldsymbol{x}_{k'} = \boldsymbol{x}_0^{t,0} + (\sum_{j=1}^{k'} \boldsymbol{d}_j^{t,0}) + \boldsymbol{d}_{k'}^{0,r} \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) \\ \wedge & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) \ \wedge \ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \end{array} \right.\\
&\implies \exists k \geqslant 0, \exists \boldsymbol{x} \in X, \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1^{t,0} \dots \boldsymbol{d}_k^{t,0} \in D^{t,0}, \exists \boldsymbol{d}_1^{0,r} \dots \boldsymbol{d}_k^{0,r} \in D^{0,r} \ :\\
&\quad \left\{ \begin{array}{lll} & \forall k' \in [1,k] \ : \ \boldsymbol{x}_{k'} = \boldsymbol{x}_0^{t,0} + (\sum_{j=1}^{k'} \boldsymbol{d}_j^{t,0}) + \boldsymbol{d}_{k'}^{0,r} \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) \\ \wedge & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) \ \wedge \ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \end{array} \right.\\
&\qquad (D \text{ approximated by the sum } (D^{t,0} + D^{0,r}))\\
&\iff \exists k \geqslant 0, \exists \boldsymbol{x} \in X, \exists \boldsymbol{x}_0, \boldsymbol{x}_k, \exists \boldsymbol{d}^{t,0} \in D^{t,0}, \exists \boldsymbol{d}_k^{0,r} \in D^{0,r} \ :\\
&\quad \left\{ \begin{array}{lll} & \boldsymbol{x}_k = \boldsymbol{x}_0^{t,0} + k\boldsymbol{d}^{t,0} + \boldsymbol{d}_k^{0,r} \\ \wedge & G(\boldsymbol{x}_0) \ \wedge \ G(\boldsymbol{x}_k) \\ \wedge & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) \ \wedge \ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \end{array} \right.\\
&\qquad (\text{because } D^{t,0}, D^{0,r} \text{ and } G \text{ are convex and } \boldsymbol{x}_0^{0,r} \in D^{0,r})\\
&\implies \exists \alpha \geqslant 0, \exists \boldsymbol{x} \in X, \exists \boldsymbol{x}_0, \boldsymbol{x}_k, \exists \boldsymbol{d}^{t,0} \in D^{t,0}, \exists \boldsymbol{d}_k^{0,r} \in D^{0,r} \ :\\
&\quad \left\{ \begin{array}{lll} \wedge & \boldsymbol{x}_k = \boldsymbol{x}_0^{t,0} + \alpha\boldsymbol{d}^{t,0} + \boldsymbol{d}_k^{0,r} \\ \wedge & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) \ \wedge \ G(\boldsymbol{x}_0) \ \wedge \ \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \end{array} \right.\\
&\qquad (\text{dense over-approximation; } G(\boldsymbol{x}_k) \text{ already implied by } \boldsymbol{x}' \in \tau(\boldsymbol{x}_k))\\
&\iff \boldsymbol{x}' \in \tau\Big(\big((\tau(X) \sqcap G)^{t,0} \nearrow D^{t,0}\big) + D^{0,r}\Big)
\end{aligned}
$$

∎

Theorem 5.2 (Translation with resets, inputs and simple guards) *The accelerated transition $\tau^{\otimes}$ for a translation/reset with inputs and a simple guard τ can be computed by applying Proposition 5.4 with D defined as in Proposition 5.2.*

Example 5.2 (Translation with resets, inputs and simple guards) *Consider the polyhedron $X = \{(x_1, x_2) \mid 0 \leqslant x_1 \ \wedge \ 1 \leqslant x_2 \ \wedge \ x_1 + x_2 \leqslant 2\}$ and the transition* $\tau : \left| \begin{array}{rcl} x_1 + 2x_2 & \leqslant & 3 \\ 0 \leqslant \xi & \leqslant & 1 \end{array} \right| \rightarrow \left| \begin{array}{l} x_1' = x_1 + \xi + 1 \\ x_2' = \xi \end{array} \right.$
Eliminating the inputs yields $D = \{(d_1, d_2) \mid 1 \leqslant d_1 \leqslant 2 \ \wedge \ d_1 - d_2 = 1\}$ and $D^{t,0} = \{(d_1, d_2) \mid 1 \leqslant d_1 \leqslant 2 \ \wedge \ d_2 = 0\}$. We obtain $\tau^{\otimes}(X) = \{(x_1, x_2) \mid x_1 + x_2 \geqslant 1 \ \wedge \ x_2 \geqslant 0 \ \wedge \ x_1 - x_2 \leqslant 4 \ \wedge \ x_1 + 5x_2 \leqslant 10 \ \wedge \ x_1 \geqslant 0\}$, see Fig. 5.4.

5.1.3 Relaxing General Guards to Simple Guards

As discussed at the beginning of §5.1, allowing constraints that relate state variables with input variables in guards, *i.e.* $G = \mathbf{A}\boldsymbol{x} + \mathbf{L}\boldsymbol{\xi} \leqslant \boldsymbol{b} \ \wedge \ \mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}$ with $\mathbf{L} \neq 0$ (see Eqn. (5.1)), makes acceleration very difficult (Prop. 5.1). Our solution is to relax the guard G to a simple guard (or cartesian product) $\overline{G} = \underbrace{(\exists \boldsymbol{\xi} : G)}_{\mathbf{A}'\boldsymbol{x} \leqslant \boldsymbol{b}'} \wedge \underbrace{(\exists \boldsymbol{x} : G)}_{\mathbf{J}'\boldsymbol{\xi} \leqslant \boldsymbol{k}'}$.

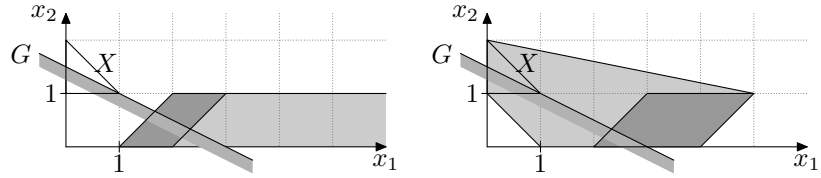


Figure 5.4: Translation/reset with inputs: Ex. 5.2. Left-hand side: $\tau(X)$ (dark shaded) and $((\tau(X) \sqcap G)^{t,0} \nearrow D^{t,0}) + D^{0,r}$ (whole shaded area). Right-hand side: $\tau(((\tau(X) \sqcap G)^{t,0} \nearrow D^{t,0}) + D^{0,r})$ (dark shaded) and $\tau^{\otimes}(X)$ (whole shaded area).

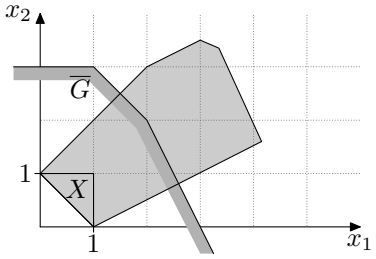


Figure 5.5: Ex. 5.3: accelerated transition $\tau^{\otimes}(X)$ using the relaxed guard $\overline{G}$ (result shaded).

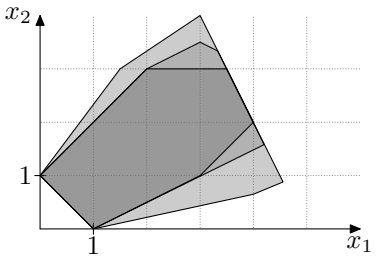


Figure 5.6: Ex. 5.3: comparison between convex hull of the exact result (dark gray), our method (gray), and widening with no delay and 3 descending iterations (light gray).

We can now apply the accelerated transition from Theorems 5.1 and 5.2 with $G' = (\mathbf{A}'\boldsymbol{x} \leqslant \boldsymbol{b}')$ and $D' = \{\boldsymbol{d} \mid \exists \boldsymbol{\xi} : \boldsymbol{d} = \boldsymbol{T}\boldsymbol{\xi} + \boldsymbol{u} \wedge \boldsymbol{J}'\boldsymbol{\xi} \leqslant \boldsymbol{k}'\}$. This trivially results in a sound over-approximation because a weaker guard is used for abstract acceleration.

Note however that in the corresponding acceleration formulas, we can still compute exactly the function τ using the original guard G. Indeed, the proofs of those theorems are not based on the assumption $\mathbf{L} \neq 0$ when they introduce the function τ.

Example 5.3 (Relaxed guard) *Consider the polyhedron* $X = \{(x_1, x_2) \,|\, x_1 \leqslant 1 \wedge x_2 \leqslant 1 \wedge x_1 + x_2 \geqslant 1\}$ *and the transition* $\tau : \left| \begin{array}{rcl} 2x_1 + x_2 + \xi & \leqslant & 6 \\ x_2 - \xi & \leqslant & 2 \\ 0 \leqslant \xi & \leqslant & 1 \end{array} \right| \rightarrow \left| \begin{array}{l} x_1' = x_1 + \xi + 1 \\ x_2' = x_2 + 1 \end{array} \right.$

The relaxed guard is $\overline{G} = (2x_1 + x_2 \leqslant 6 \wedge x_1 + x_2 \leqslant 4 \wedge x_2 \leqslant 3) \wedge (0 \leqslant \xi \leqslant 1)$. *Eliminating the inputs yields* $D = \{(d_1, d_2) \,|\, 1 \leqslant d_1 \leqslant 2 \wedge d_2 = 1\}$. *We obtain* $\tau^{\otimes}(X) = \{(x_1, x_2) \,|\, x_1 + x_2 \geqslant 1 \wedge x_2 - x_1 \leqslant 1 \wedge -4 \leqslant x_1 - 2x_2 \leqslant 1 \wedge x_1 + 2x_2 \leqslant 10 \wedge 2x_1 + x_2 \leqslant 10\}$, *see Fig. 5.5. The convex hull of the exact result is* $\{(x_1, x_2) \,|\, x_1 + x_2 \geqslant 1 \wedge -2 \leqslant x_2 - x_1 \leqslant 1 \wedge x_1 - 2x_2 \leqslant 1 \wedge x_2 \leqslant 3 \wedge 2x_1 + x_2 \leqslant 10\}$, *see Fig. 5.6.*

5.2 Backward Abstract Acceleration

Abstract acceleration has been applied to forward reachability analysis in order to compute the reachable states starting from a set of initial states. Backward analysis computes the states co-reachable from the *error* states. For example, combining forward and backward analysis allows to obtain an approximation of the sets of states belonging to a path from initial to error states (see for instance [Jea03]). Moreover, a backward

analysis allows us to *synthesize* constraints on *parameter* variables that ensure that a property is satisfied (see, *e.g.*, [ACH+95]).

In this section, we present how to compute the accelerated backward transitions in the case of translations and translations/resets. Although the inverse of a translation is a translation, the difference is that the intersection with the guard occurs after the (inverted) translation. The case of backward translations with resets is more complicated than for the forward case, because resets are not invertible. Finally, the relaxation of general guards to simple guards applies in the same way to backward acceleration.

5.2.1 Translations

Proposition 5.5 (Polyhedral backward translation) *Let τ be a polyhedral translation $G \to \boldsymbol{x}' = \boldsymbol{x} + D$. Then the set*

$$\tau^{-\otimes}(X') = X' \sqcup \big((\tau^{-1}(X') \nearrow (-D)\big) \sqcap G$$

is a convex over-approximation of $\tau^{-*}(X')$, where $\tau^{-*} = (\tau^{-1})^* = (\tau^*)^{-1}$ is the reflexive and transitive backward closure of τ.

$(-D)$ denotes the reflexion of D w.r.t. the origin: $\boldsymbol{d} \in (-D) \iff (-\boldsymbol{d}) \in D$.

Proof
$\boldsymbol{x}_0 \in \bigcup_{k\geqslant 1} \tau^{-k}(X') \iff \exists \boldsymbol{x}' \in X' : \boldsymbol{x}' \in \tau\left(\bigcup_{k\geqslant 0} \tau^k(\{\boldsymbol{x}_0\})\right)$

$$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_k, \exists \boldsymbol{d}_1, \ldots, \boldsymbol{d}_k \in D \ : \begin{cases} \boldsymbol{x}_k = \boldsymbol{x}_0 + \sum_{j=1}^{k} \boldsymbol{d}_j \\ \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_0 + \sum_{j=1}^{k'} \boldsymbol{d}_j) \\ \boldsymbol{x}' \in \tau(\{\boldsymbol{x}_k\}) \end{cases}$$

(forward reachability)

$$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_k, \exists \boldsymbol{d}_1, \ldots, \boldsymbol{d}_k \in D \ : \begin{cases} \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \\ \boldsymbol{x}_0 = \boldsymbol{x}_k - \sum_{j=1}^{k} \boldsymbol{d}_j \\ \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_k - \sum_{j=k'+1}^{k} \boldsymbol{d}_j) \end{cases}$$

(rewritten as backward reachability)

$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_k, \exists \boldsymbol{d} \in D \ : \ \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x}_0 = \boldsymbol{x}_k - k\boldsymbol{d} \ \wedge \ G(\boldsymbol{x}_0) \ \wedge \ G(\boldsymbol{x}_k)$
(because D and G are convex)

$\implies \exists \alpha \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_k, \exists \boldsymbol{d} \in D \ : \ \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x}_0 = \boldsymbol{x}_k - \alpha\boldsymbol{d} \ \wedge \ G(\boldsymbol{x}_0)$
(dense approximation; $G(\boldsymbol{x}_k)$ implied by $\boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\})$)

$\iff \boldsymbol{x}_0 \in \big((\tau^{-1}(X') \nearrow (-D)\big) \sqcap G.$ ∎

Example 5.4 (Polyhedral backward translation) *Consider the polyhedron $X' = \{(x_1, x_2) \mid 3 \leqslant x_1 \leqslant 6 \ \wedge \ 4 \leqslant x_2 \leqslant 5\}$ and the transition*

$$\tau : \left| \begin{array}{rc} x_1 + 2x_2 \leqslant 10 & \wedge\, 0 \leqslant x_1 \leqslant 4 \ \wedge \\ 0 \leqslant x_2 & \wedge\, 1 \leqslant \xi \leqslant 2 \end{array} \right. \to \left| \begin{array}{l} x_1' = x_1 + 1 \\ x_2' = x_2 + \xi \end{array} \right.$$

The polyhedron D is $\{(d_1, d_2) \,|\, d_1 = 1 \wedge 1 \leqslant d_2 \leqslant 2\}$. As result of the backward acceleration (Fig. 5.7) we obtain the polyhedron $\{(x_1, x_2) \mid 0 \leqslant x_1 \leqslant 6 \ \wedge \ 0 \leqslant x_2 \leqslant 5 \ \wedge \ -x_1 + x_2 \leqslant 2 \wedge 4x_1 - 3x_2 \leqslant 12\}$.

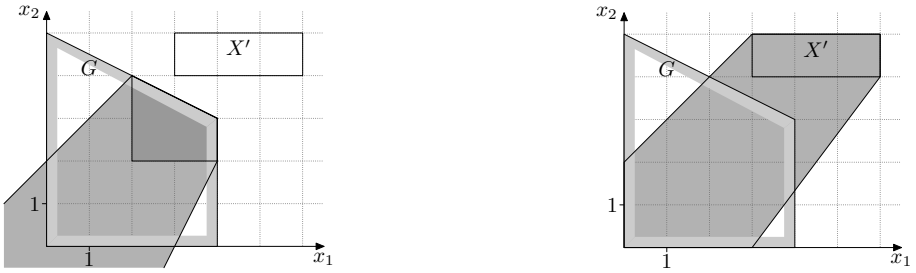


Figure 5.7: Backward acceleration of a translation loop (Ex. 5.4) starting from X' with $\tau^{-1}(X')$ (dark gray) and $\tau^{-1}(X') \nearrow (-D)$ (whole shaded area) on the left-hand side and the final result (right-hand side).

5.2.2 Translations with Resets

Proposition 5.6 (Polyhedral backward translation with resets) *Let τ be a polyhedral translation with resets $G \to \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + D$. Then, the set*

$$\tau^{-\otimes}(X') = X' \sqcup \tau^{-1}(X') \sqcup \tau^{-1}\Big(\big((\tau^{-1}(X')^{t,\bullet} \nearrow (-D^{t,0})\big) \sqcap D^{\bullet,r} \sqcap G\Big)$$

is a convex over-approximation of $\tau^{-*}(X')$.

Proof The formula is trivially correct for 0 or 1 backward iterations of the self-loop τ, thus, it remains to show that, for the case of $k \geqslant 2$ iterations, our formula yields an over-approximation of $\bigcup_{k\geqslant 2} \tau^{-k}(X)$.

$\boldsymbol{x} \in \bigcup_{k\geqslant 2} \tau^{-k}(X') \iff \exists \boldsymbol{x}' \in X' \ : \ \boldsymbol{x}' \in \tau\left(\bigcup_{k\geqslant 0} \tau^k(\tau(\boldsymbol{x}))\right)$

$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1 \dots \boldsymbol{d}_k \in D \ :$

$$\left\{\begin{array}{ll} & \boldsymbol{x}_0 \in \tau(\boldsymbol{x}) \wedge \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \\ \wedge & \forall k' \in [1,k] \ : \ \boldsymbol{x}_{k'} = \boldsymbol{x}_0^{t,0} + \sum_{j=1}^{k'} \boldsymbol{d}_j^{t,0} + \boldsymbol{d}_{k'}^{0,r} \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) \end{array}\right.$$

(forward reachability)

$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1 \dots \boldsymbol{d}_k \in D \ :$

$$\left\{\begin{array}{ll} & \forall k' \in [0,k-1] \ : \ \boldsymbol{x}_{k'} = \boldsymbol{x}_k^{t,0} - \sum_{j=k'+1}^{k} \boldsymbol{d}_j^{t,0} + \boldsymbol{d}_{k'}^{0,r} \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) \\ \wedge & \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x} \in \tau^{-1}(\{\boldsymbol{x}_0\}) \end{array}\right.$$

(rewritten as backward reachability)

$\implies \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_0 \dots \boldsymbol{x}_k, \exists \boldsymbol{d}_1^{t,0} \dots \boldsymbol{d}_k^{t,0} \in D^{t,0}, \exists \boldsymbol{d}_1^{0,r} \dots \boldsymbol{d}_k^{0,r} \in D^{0,r} \ :$

$$\left\{\begin{array}{ll} & \forall k' \in [0,k-1] \ : \ \boldsymbol{x}_{k'} = \boldsymbol{x}_k^{t,0} - \sum_{j=k'+1}^{k} \boldsymbol{d}_j^{t,0} + \boldsymbol{d}_{k'}^{0,r} \\ \wedge & \forall k' \in [0,k] \ : \ G(\boldsymbol{x}_{k'}) \\ \wedge & \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x} \in \tau^{-1}(\{\boldsymbol{x}_0\}) \end{array}\right.$$

(D approximated by the sum $(D^{t,0} + D^{0,r})$)

$\iff \exists k \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_0, \boldsymbol{x}_k, \exists \boldsymbol{d}^{t,0} \in D^{t,0}, \exists \boldsymbol{d}^{0,r} \in D^{0,r} \ :$

$$\left\{\begin{array}{ll} & \boldsymbol{x}_0 = \boldsymbol{x}_k^{t,0} - k\boldsymbol{d}^{t,0} + \boldsymbol{d}^{0,r} \\ \wedge & G(\boldsymbol{x}_0) \wedge G(\boldsymbol{x}_k) \\ \wedge & \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x} \in \tau^{-1}(\{\boldsymbol{x}_0\}) \end{array}\right.$$

(because $D^{t,0}, D^{0,r}$ and G are convex and $\boldsymbol{x}_k^{0,r} \in D^{0,r}$)

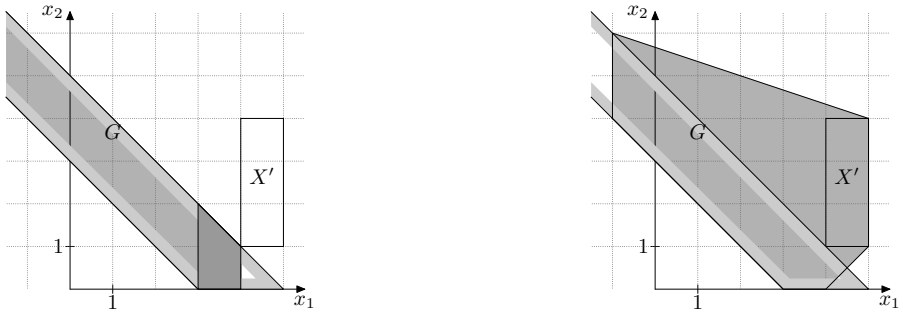


Figure 5.8: Backward acceleration of a loop with translations and resets (Ex. 5.5) starting from the initial set X'. Right-hand side: $\tau^{-1}(X')$ (dark gray) and $\big((\tau^{-1}(X')|_t \nearrow (-D^t)\big) \sqcap G$ (whole shaded area). Left-hand side: final result (whole shaded area).

$$\begin{array}{l}
\Longrightarrow \exists \alpha \geqslant 0, \exists \boldsymbol{x}' \in X', \exists \boldsymbol{x}_0, \boldsymbol{x}_k, \exists \boldsymbol{d}^{t,0} \in D^{t,0}, \exists \boldsymbol{d}^{0,r} \in D^{0,r} \ : \\
\quad \left\{ \begin{array}{ll} & \boldsymbol{x}_0 = \boldsymbol{x}_k^{t,0} - \alpha \boldsymbol{d}^{t,0} + \boldsymbol{d}^{0,r} \\ \wedge & \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\}) \ \wedge \ \boldsymbol{x} \in \tau^{-1}(\{\boldsymbol{x}_0\}) \ \wedge \ G(\boldsymbol{x}_0) \end{array} \right. \\
\quad \text{(dense approximation; } G(\boldsymbol{x}_k) \text{ implied by } \boldsymbol{x}_k \in \tau^{-1}(\{\boldsymbol{x}'\})) \\
\Longleftrightarrow \boldsymbol{x} \in \tau^{-1}\Big(\big((\tau^{-1}(X'))^{t,\bullet} \nearrow (-D^{t,0})\big) \sqcap D^{\bullet,r} \sqcap G\Big) \\
\quad \text{(because } \boldsymbol{x} \in \tau^{-1}(\{\boldsymbol{x}'\}) \Rightarrow \boldsymbol{x}' \in D^{\bullet,r}) \qquad \blacksquare
\end{array}$$

Example 5.5 (Polyhedral backward translation with resets) *Consider the polyhedron* $X' = \{(x_1, x_2) \mid 4 \leqslant x_1 \leqslant 5 \ \wedge \ 1 \leqslant x_2 \leqslant 4\}$ *and the transition*

$$\tau : \left| \begin{array}{rl} 3 \leqslant x_1 + x_2 \leqslant 5 & \wedge \\ 1 \leqslant \xi \leqslant 3 & \wedge 0 \leqslant x_2 \end{array} \right. \rightarrow \left| \begin{array}{l} x_1' = x_1 + 1 \\ x_2' = \xi \end{array} \right.$$

The polyhedron D *is* $\{(d_1, d_2) | d_1 = 1 \wedge 1 \leqslant d_2 \leqslant 3\}$. *As result of the backward acceleration (Fig. 5.8), we obtain the polyhedron* $\{(x_1, x_2) \mid -1 \leqslant x_1 \leqslant 5 \ \wedge \ 0 \leqslant x_2 \ \wedge \ x_1 + x_2 \geqslant 3 \ \wedge \ x_1 - x_2 \leqslant 4 \ \wedge \ x_1 + 3x_2 \leqslant 17\}$.

5.3 Evaluation and Comparison

This section discusses the advantages and shortcomings of abstract acceleration in comparison with more general abstract-interpretation-based methods like standard widening [CC77] and the affine derivative closure method of Ancourt et al [ACI10].

5.3.1 Comparing Abstract Acceleration with Kleene Iteration

Abstract acceleration aims at computing a tight over-approximation of $\alpha(\bigcup_{k \geqslant 0} \tau^k(X_0))$ where X_0 is a convex polyhedron and τ is an affine transformation with an affine guard. Since convex polyhedra are closed under affine transformations ($\alpha(\tau(X_0)) = \tau(X_0)$), we have $\alpha(\bigcup_{k \geqslant 0} \tau^k(X_0)) = \bigsqcup_{k \geqslant 0} \tau^k(X_0)$. The latter formula is known as *Merge-Over-All-Paths* (MOP) solution of the reachability problem [KU77], which computes the limit of the sequence:

$$X_0 \qquad X_1 = X_0 \sqcup \tau(X_0) \qquad X_2 = X_0 \sqcup \tau(X_0) \sqcup \tau^2(X_0) \qquad \ldots$$

In contrast, the standard approach in abstract interpretation computes the fixed point X'_∞ of $X = X_0 \sqcup \tau(X)$, known as the *Minimal-Fixed-Point* (MFP) solution. It proceeds as follows:

$$X'_0 = X_0 \qquad X'_1 = X'_0 \sqcup \tau(X'_0) \qquad X'_2 = X'_0 \sqcup \tau(X'_0 \sqcup \tau(X'_0)) \qquad \dots$$

The MOP solution is more precise than the MFP solution [KU77]. The reason is that, in general, τ does not distribute over $\sqcup$ and we have $\tau(X_1) \sqcup \tau(X_2) \sqsubseteq \tau(X_1 \sqcup X_2)$. For instance, if $X_0 = [0,0]$ and $\tau : x \leqslant 1 \to x' = x+2$, we have $X_2 = [0,0] \sqcup [2,2] \sqcup \bot = [0,2]$ and $X'_2 = [0,0] \sqcup \tau([0,2]) = [0,3]$. Since abstract acceleration should deliver a tight over-approximation of the MOP solution, we should generally have the relationship $\bigsqcup_{k \geqslant 0} \tau^k(X_0) \sqsubseteq \tau^{\otimes}(X_0) \sqsubseteq X'_\infty$, *i.e.*, abstract acceleration should be more precise than the standard abstract interpretation approach (MFP) even without widening.

Fig. 5.9 shows an example illustrating this issue: in the standard abstract interpretation approach (MFP), each iteration translates the approximation added by the convex union with the result of the previous iteration in each step and converges slowly. Abstract acceleration translates only the intersection with the guard and takes the union as a last step. Observe, however, that in the case where X_0 is contained in the guard G, MOP and MFP solutions give identical results for this example [LS07].

Non-flat systems. Generally, the invariants computed by abstract acceleration of a single self-loop τ are not inductive, which implies that $\tau^{\otimes}$ is not idempotent, *i.e.*, $\tau^{\otimes}(\tau^{\otimes}(X_0)) \neq \tau^{\otimes}(X_0)$. While this is not a problem for flat systems, it has negative effects in the presence of nested loops.

For example, in the system $(id \circ \tau^*)^*$ we can apply abstract acceleration to the innermost loop: $(id \circ \tau^{\otimes})^* = (\tau^{\otimes})^*$. If $\tau^{\otimes}$ is not idempotent (like in Fig. 5.9b), then the outer loop might not converge and thus widening is needed. Thus, the considerations w.r.t. MOP and MFP solutions for τ^* above apply to $(\tau^{\otimes})^*$ in the same manner.

Since this problem arises in particular when the initial set is not contained in the guard G (like in Fig. 5.9b), Leroux and Sutre [LS07] propose to accelerate translations by the formula $\tau^{\otimes}(X) = \tau(X \nearrow D)$, *i.e.*, without initially intersecting with G, which is idempotent and hence convergence without widening can be expected more often. However, this formula is clearly less precise and should not be used for accelerating non-nested loops.

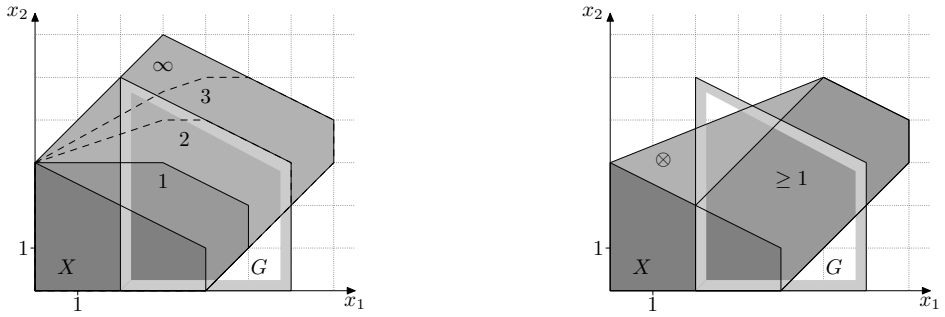
5.3.2 Comparing Abstract Acceleration with Widening

In this section, we illustrate by some examples where abstract acceleration helps increasing the predictability and the robustness of the analysis.

Widening in self-loops. We consider Ex. 5.3, which has the structure depicted in Fig. 5.10 when analyzed with abstract acceleration or Kleene iteration with widening and descending iterations respectively.

We try first an analysis with undelayed widening, *i.e.*, $N = 0$:

$$\begin{array}{lcll}
X'_1 & = & \{(x_1,x_2) \mid x_1+x_2 \geqslant 1\} & \text{(result of converged widening sequence)} \\
X''_1 & = & \{(x_1,x_2) \mid x_1+x_2 \geqslant 1 \wedge 2x_1+x_2 \leqslant 10 \wedge x_2 \leqslant 4 \wedge 0 \leqslant x_1 \leqslant 6 \wedge 3x1+5x_2 \geqslant 3\} & \\
\dots & & & \\
X''_3 & = & \{(x_1,x_2) \mid x_1+x_2 \geqslant 1 \wedge 2x_1+x_2 \leqslant 10 \wedge 3x_2-2x_1 \leqslant 6 \wedge 3x_2-4x_1 \leqslant 3 \wedge & \\
 & & \quad 5x_1-22x_2 \leqslant 8 \wedge 29x_1-157x_2 \leqslant 29\} \supset \tau^{\otimes}(X) & \\
\dots & & & \text{(descending sequence)}
\end{array}$$



(a) MFP: X dark gray, iteration 1 medium gray, iterations 2 and 3 dashed, final result whole shaded area.

(b) Abstract acceleration: X dark gray, iterations $\geqslant 1$ medium gray, final result whole shaded area.

Figure 5.9: Comparison between standard abstract interpretation (MFP) (a) and abstract acceleration (b): $\tau : G \to (x_1', x_2') = (x_1+1, x_2+1)$ with G and X as given in the figures.

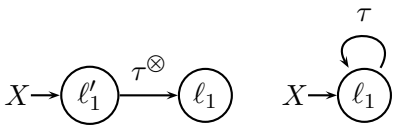


Figure 5.10: Analysis with acceleration (left-hand side) and with widening (right-hand side) for Ex. 5.3.

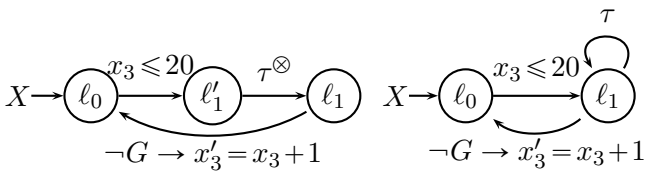


Figure 5.11: Analysis with acceleration (left-hand side) and with widening (right-hand side) for Ex. 5.6. $G = (2x_1+2x_2 \leqslant x_3)$.

Here, the descending iterations do not converge and improve the result slowly. See Fig. 5.6 for X_3''.

By increasing the delay to $N = 1$, we can improve the result, and we get the same result as with abstract acceleration: $X_1'' = \tau^{\otimes}(X)$. However, this is not guaranteed in general, because widening is not monotonic. Moreover, abstract acceleration is more efficient computationally: delaying widening and a long descending sequence increase the number of iterations. In flat programs, like in Ex. 5.3, abstract acceleration does not even require convergence tests.

Widening in nested loops. In the previous example, delayed widening and descending iterations allowed to get the same result as with abstract acceleration. However, this is less likely if the loop is embedded in an outer loop as in Fig. 5.11: descending iterations cannot be applied during the ascending iterations, but only after convergence of the widening sequence of the whole program, otherwise convergence would not be guaranteed.

Example 5.6 (Widening in nested loops) *We consider the program depicted in Fig. 5.11 in which the inner loop τ is adapted from Ex. 5.2:*

$$\tau : \left| \begin{array}{c} 2x_1 + 2x_2 \leqslant x_3 \\ 0 \leqslant \xi \leqslant 1 \end{array} \right. \to \left| \begin{array}{l} x_1' = x_1 + \xi + 1 \\ x_2' = \xi \\ x_3' = x_3 \end{array} \right. \qquad X = \left\{ (x_1, x_2, x_3) \;\middle|\; \begin{array}{l} 0 \leqslant x_1 \;\wedge\; 1 \leqslant x_2 \\ x_1 + x_2 \leqslant 2 \wedge x_3 = 3 \end{array} \right\}$$

The analysis without abstract acceleration yields for any widening delay $N \geqslant 1$ (widening point ℓ_1) and any number of descending iterations $N'' \geqslant 1$ the following very

weak invariant:

$$X_1' = \{(x_1, x_2, x_3) \mid 0 \leqslant x_1 \wedge 3 \leqslant x_3\} \quad X_1'' = \{(x_1, x_2, x_3) \mid 0 \leqslant x_1 \wedge 1 \leqslant x_1 + x_2 \wedge 3 \leqslant x_3\}$$

Abstract acceleration with widening delay $N \geqslant 1$ (widening point ℓ_1) and one descending iteration gives much better results: we give here a simplified (over-approximated) invariant, because the actual result consists of more constraints:

$$X_1'' = \{(x_1, x_2, x_3) \mid 0 \leqslant x_1 \leqslant 12 \wedge 0 \leqslant x_2 \leqslant 3 \wedge 3 \leqslant x_3 \leqslant 20 \wedge 1 \leqslant x_1 + x_2\}$$

One can also consider widening with thresholds (see §3.4.2). A natural threshold set for our example is the postcondition of the guard of τ by the body of τ: $\tau(\top) = \{(x_1, x_2) \mid 0 \leqslant x_2 \leqslant 1\}$. Yet, this does improve the result.

Extending the threshold set with the postcondition of the guard of the outer loop $x_3 \leqslant 21$ improves the result (all variables are bounded), but it is still less precise than the result obtained by combining abstract acceleration and widening (in particular the descending iteration does not converge).

5.3.3 Comparing Abstract Acceleration with the Affine Derivative Closure Algorithm

The affine derivative closure algorithm of Ancourt et al. [ACI10] is another abstract interpretation-based analysis method. The idea is to compute an abstract transformer, *i.e.*, a relation between variables $\boldsymbol{x}$ and $\boldsymbol{x}'$, independently of the initial state of the system. The abstract transformer *abstracts* the effect of the loop by a polyhedral translation

$$\mathit{true} \rightarrow \boldsymbol{x}' = \boldsymbol{x} + D_R \quad \text{with} \quad D_R = \{\boldsymbol{d} \mid \exists \boldsymbol{x}, \boldsymbol{\xi}, \boldsymbol{x}' : R(\boldsymbol{x}, \boldsymbol{\xi}, \boldsymbol{x}') \wedge \boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{d}\}$$

where R is the concrete transition relation. The polyhedron D_R is called the "derivative" of the relation R. The effect of several self-loops with relations $R_1, \ldots, R_k$ is abstracted by considering the convex union $\bigsqcup_i D_{R_i}$.

Then, the reflexive and transitive closure

$$R^* = \{(\boldsymbol{x}, \boldsymbol{x}') \mid \exists k \geqslant 0 : \boldsymbol{x}' = \boldsymbol{x} + k\boldsymbol{d} \wedge D_R(\boldsymbol{d})\}$$

is applied to a polyhedron X of initial states:

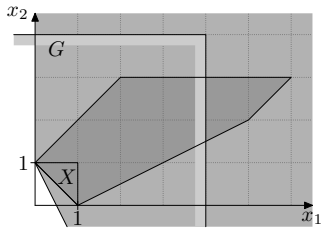
$$R^*(X) = \{\boldsymbol{x}' \mid \exists \boldsymbol{x} : R^*(\boldsymbol{x}, \boldsymbol{x}') \wedge X(\boldsymbol{x})\}$$

The final result is obtained by computing one "descending" iteration, in the same way as it is done in standard abstract interpretation after widening.

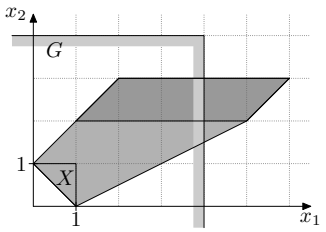
The affine derivative closure algorithm is implemented in the code optimization tool PIPS[2].

In single self-loops with translations or translations/resets, the method works similarly to abstract acceleration, as illustrated by the following example involving resets and inputs:

[2] `http://pips4u.org`



(a) Derivative closure method: R^* applied to X (whole shaded area), final result (dark gray).



(b) Abstract acceleration: iterations $\geqslant 1$ (dark gray), final result (whole shaded area).

Figure 5.12: Comparison between the affine derivative closure algorithm (a) and abstract acceleration (b), Ex. 5.7.

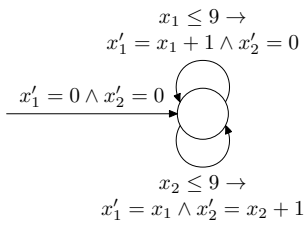


Figure 5.13: Ex. 5.8.

Example 5.7 (Single loop)

$$\tau : \left| \begin{array}{r} x_1 \leqslant 4 \\ x_2 \leqslant 4 \\ 0 \leqslant \xi \leqslant 1 \end{array} \right. \rightarrow \left| \begin{array}{l} x_1' = x_1 + \xi + 1 \\ x_2' = \xi + 2 \end{array} \right. \qquad X = \left\{ (x_1, x_2) \left| \begin{array}{l} x_1 \leqslant 1 \ \wedge \ x_2 \leqslant 1 \\ x_1 + x_2 \geqslant 1 \end{array} \right. \right\}$$

The transition relation

$$R = \{(x_1, x_2, \xi, x_1', x_2') \mid x_1' = x_1 + \xi + 1 \wedge x_2' = \xi + 2 \wedge x_1 \leqslant 4 \wedge x_2 \leqslant 4 \wedge 0 \leqslant \xi \leqslant 1\}$$

expressed in terms of derivatives is

$$\begin{aligned} D_R = \ & \{(d_1, d_2) \mid \exists x_1, x_2, \xi, x_1', x_2' : R(x_1, x_2, \xi, x_1', x_2') \wedge x_1' = x_1 + d_1 \wedge x_2' = x_2 + d_2\} \\ = \ & \{(d_1, d_2) \mid 1 \leqslant d_1 \leqslant 2 \wedge d_1 - d_2 \leqslant 3\} \end{aligned}$$

The closure of the loop starting from X gives

$$\begin{aligned} R^*(X) \ = \ & \{(x_1', x_2') \mid \exists k \geqslant 0, x_1, x_2 : x_1' \geqslant x_1 + k \wedge x_1' \leqslant x_1 + 2k \wedge \\ & \qquad x_1' - x_2' \leqslant x_1 - x_2 + 3k \wedge X(x_1, x_2)\} = \\ = \ & \{(x_1', x_2') \mid 0 \leqslant x_1' \wedge 2x_1' + x_2' \geqslant 1\} \end{aligned}$$

Finally, a descending iteration is computed:

$$\begin{aligned} X \sqcup \tau(R^*(X)) = \{(x_1', x_2') \mid \ & x_1' + x_2' \geqslant 1 \wedge x_1' - 2x_2' \leqslant 1 \wedge x_1' - x_2' \leqslant 3 \wedge \\ & x_2' \leqslant 3 \wedge x_1' - x_2' \geqslant -1\} \end{aligned}$$

This result equals the one obtained by abstract acceleration (see Fig. 5.12).

Resets cannot be expressed as polyhedral translations: for instance, if $R(x, x') = (x' = 0)$, then $D_R = \{d \mid \exists x, x' : x' = 0 \wedge x' = x + d\} = \top$. However, this information is recovered during the descending iteration. Hence, similarly to widening, these descending iterations may fail (cf. §3.4.2) in the presence of multiple loops:

Example 5.8 (Multiple loops) *For the CFG in Fig. 5.13, the derivatives for the upper and the lower loop are $D_{R_1} = (d_1 = 1)$ and $D_{R_2} = (d_1 = 0 \wedge d_2 = 1)$ respectively. Their convex union is $D_{R_{1,2}} = (0 \leqslant d_1 \leqslant 1)$. The transitive closure applied to $X = (x_1 = x_2 = 0)$ gives $R^*_{1,2}(X) = (x_1 \geqslant 0)$, and the final result after the descending iteration is $X' = (x_1 \geqslant 0 \wedge x_2 \leqslant 10)$. Here, in the same way as in Ex. 3.3, the descending iteration fails to recover the upper bound of x_1.*

In contrast, abstract acceleration converges after two ascending iterations with the invariant $X_2 = (0 \leqslant x_1 \leqslant 10 \wedge 0 \leqslant x_2 \leqslant 10)$.

Forward acceleration:

Translations	$\tau^{\otimes}(X) = X \sqcup \tau\big((X \sqcap G) \nearrow D\big)$
Translations/resets	$\tau^{\otimes}(X) = X \sqcup \tau(X) \sqcup \tau\Big(\big((\tau(X) \sqcap G)^{t,0} \nearrow D^{t,0}\big) + D^{0,r}\Big)$

Backward acceleration:

Translations	$\tau^{-\otimes}(X') = X' \sqcup \big((\tau^{-1}(X') \nearrow (-D)\big) \sqcap G$
Translations/resets	$\tau^{-\otimes}(X') = X' \sqcup \tau^{-1}(X') \sqcup$ $\sqcup \tau^{-1}\Big(\big((\tau^{-1}(X')^{t,\bullet} \nearrow (-D^{t,0})\big) \sqcap D^{\bullet,r} \sqcap G\Big)$

with

$$\tau : (\overbrace{\mathbf{A}\boldsymbol{x} + \mathbf{L}\boldsymbol{\xi} \leqslant \boldsymbol{b}}^{G_0(\boldsymbol{x},\boldsymbol{\xi})} \wedge \overbrace{\mathbf{J}\boldsymbol{\xi} \leqslant \boldsymbol{k}}^{G_1(\boldsymbol{\xi})}) \longrightarrow \boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \mathbf{T}\boldsymbol{\xi} + \boldsymbol{u} \qquad \mathbf{C} \text{ diagonal with 0 or 1 only}$$
$$G(\boldsymbol{x}) = \exists \boldsymbol{\xi} : G_0(\boldsymbol{x}, \boldsymbol{\xi}) \qquad D = \{\boldsymbol{d} \mid \exists \boldsymbol{\xi} : \boldsymbol{d} = \boldsymbol{T}\boldsymbol{\xi} + \boldsymbol{u} \wedge G_1(\boldsymbol{\xi}) \wedge \exists \boldsymbol{x} : G_0(\boldsymbol{x}, \boldsymbol{\xi})\}$$

Approximations:

In all cases	dense and convex approximation
$L \neq \mathbf{0}$	G and D are decoupled.
Translations/resets	D is approximated by the Cartesian product $D^t \times D^r$.

Table 5.1: Overview of abstract acceleration formulas

Hence, even though the derivative closure method elegantly deals with multiple loops by taking the convex union of the derivatives, it is also less precise than abstract acceleration for such programs. However, the main advantage of the derivative closure method is that it is more general than abstract acceleration, because it automatically approximates any kind of transformations. Moreover, since it computes abstract transformers, it is modular and can be used in the context of interprocedural analyses.

5.4 Conclusions and Perspectives

We have presented an extension of abstract acceleration to numerical inputs for forward and backward analysis. Table 5.1 shows a summary of the formulas. This extension is less straightforward than supposed – most notably due to the observation that inputs can be used to turn translations into arbitrary affine transformations; also, resetting variables to input values may cause some subtle behavior. Regarding approximations, Table 5.1 shows the cases where our method is precise in the sense that we perform only dense and convex approximations, and the more complex cases for which additional approximations are necessary to abstract away the number of iterations.

Abstract acceleration can be elegantly integrated into an abstract interpretation-based verification tool, where it is normally used in combination with widening: as pointed out in §5.3.2, it is possible to accelerate the innermost loops precisely while using widening for the outer loops in nested loop situations. Thus, better invariants can be computed for programs for which a lot of information is lost when using widening only (cf. experimental results in §8.4).

In comparison to other abstract interpretation-based transitive closure methods, for instance the affine derivative closure algorithm [ACI10], abstract acceleration deals only with some frequently occurring types of self-loop transitions and needs to resort to widening in the general case. However, the derivative closure may be less precise in nested loops since it ultimately relies on descending iterations to recover the information

about loop guards, whereas abstract acceleration can precisely accelerate the innermost loops.

Improving precision. Regarding integer (*e.g.*, divisibility) properties, our techniques based on convex polyhedra cannot express them and Rem. 4.1 discusses the effect of the induced dense approximation. To improve on this, we could combine our techniques with the linear congruence abstract domain introduced in [Gra91]. This domain satisfies the finite ascending chain condition, hence it does not require widening nor acceleration. By this means we could tighten the results.

For instance, when (x_1, x_2) is iteratively translated by $(4, 2)$ starting from $(1, 0)$, we know from the linear congruences domain that $x_1 = 1 \mod 4 \wedge x_2 = 0 \mod 2$. Then, assuming that the abstract acceleration results in a convex polyhedron $1 \leqslant x_1 \wedge 0 \leqslant x_2 \wedge x_1 + x_2 \leqslant 4$, we can tighten this polyhedron to $x_1 = 1 \wedge 0 \leqslant x_2 \leqslant 2$.

Compared to Presburger arithmetic, we still limit ourselves to convex sets with such a technique.

Periodic affine transformations with inputs. In this chapter we dealt only with translations (*e.g.*, $x' = x+2$) and resets (*e.g.*, $x' = 1$). Yet, our experiments showed that there are other kinds of transitions in synchronous programs that would be worthwhile to accelerate: *variable exchanges* (*e.g.*, $x'_1 = x_2;\ x'_2 = x_1$) and *delays* (*e.g.*, $x'_1 = x_2;\ x'_2 = x_3$) result in transitions that fall into the category of (ultimately) periodic transformations; hence, we know how to accelerate them by applying Thm. 4.6.

We can extend the abstract acceleration of these transformations to inputs based on Eq. (4.2) $\tau^*(X) = \bigcup_{0 \leqslant j \leqslant q-1} (\tau^q)^*(\tau^j(X))$, where, in the case of inputs, τ^q is defined as:

$$\bigwedge_{0 \leqslant i \leqslant q-1} \left(\mathbf{A}\mathbf{C}^i \boldsymbol{x} + \mathbf{L}\mathbf{C}^i \boldsymbol{\xi}_i + \sum_{0 \leqslant j \leqslant i-1} \mathbf{T}\mathbf{C}^j \boldsymbol{\xi}_j - \mathbf{C}^j \boldsymbol{u} \leqslant \boldsymbol{b} \right) \quad \rightarrow \quad \left(\boldsymbol{x}' = \mathbf{C}^q \boldsymbol{x} + \sum_{0 \leqslant j \leqslant q-1} \mathbf{T}\mathbf{C}^j \boldsymbol{\xi}_j - \mathbf{C}^j \boldsymbol{u} \right)$$

Then we can accelerate each τ^q by relaxing the guard according to §5.1.3 and applying respectively the theorems for translations with inputs (Thm. 5.1) and translations with resets and inputs (Thm. 5.2) in the eigenbasis of $\mathbf{C}^q$ according to Lem. 4.1. Mind that we need to duplicate the inputs q times.

Another interesting case of transitions we encountered are *dependencies on unmodified variables* (*e.g.*, $x'_1 = x_1 + x_2;\ x'_2 = x_2$) which we will deal with in the next chapter.

Chapter 6

Revisiting Acceleration

In chapter 5 we considered abstract accelerations of translations and translations with resets, and discussed shortly the general case of (ultimately) periodic transformations. Yet, in our experiments, we encountered transitions which involve *dependencies on unmodified variables* (*e.g.*, $x'_1 = x_1 + x_2$; $x'_2 = x_2$) that are not (ultimately) periodic and hence they are not considered accelerable in exact acceleration theory. Nonetheless, intuitively, we should be able to accelerate them. This observation led us to revisit the concept of linear accelerability.

Outline. We start with a motivating example below. Then we restate the definition of linear accelerability using a *characterization based on the Jordan form* (§6.1) of homogenized affine transformations and we show that it is more general than *finite monoid acceleration* (§6.2). In §6.3 we *generalize abstract acceleration* to our linear accelerability criterion. We conclude with a discussion about the practical implications of these results and further research directions in generalizing abstract acceleration (§6.4). Fig. 5.1 illustrates the organization of the chapter.

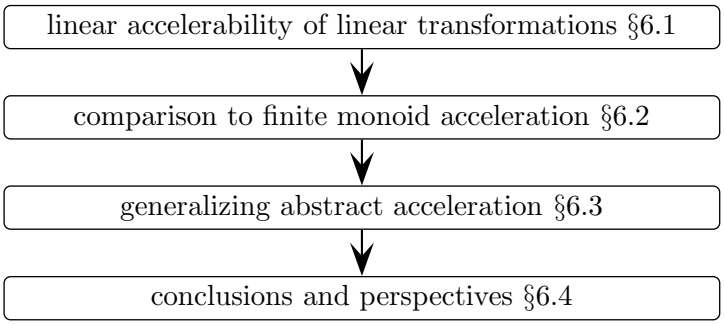


Figure 6.1: Chapter organization

Motivating example

We give a simple example that is not finite-monoidal:

Example 6.1 (Dependencies on unmodified variables I)

$$\tau : \mathsf{tt} \to \boldsymbol{x}' = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \boldsymbol{x} \text{ with } X_0 = (0 \leqslant x_1 = x_2)$$

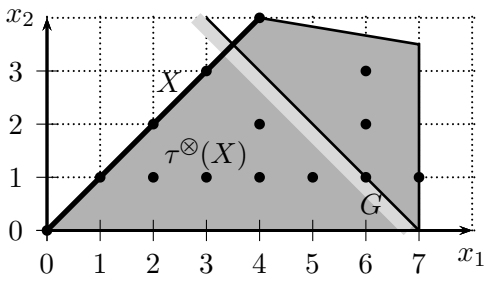


Figure 6.2: Dependencies on unmodified variables: $\tau : x_1{+}x_2 \leqslant 7 \;\rightarrow\; x'_1 = x_1{+}x_2 \wedge x'_2 = x_2$ with $X_0 = (0 \leqslant x_1 = x_2 \leqslant 4)$: reachable integer points and polyhedron computed by abstract acceleration.

We have $\mathbf{C}^k = \begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$*, which means that it is not finite-monoid (Def. 4.2), and thus not accelerable. And indeed, although the example seems trivial, its transitive closure is* not *Presburger-definable because of the multiplicative term* kx_2*:*

$$\tau^*(X_0) = \{\boldsymbol{x}' \mid \exists k \geqslant 0 : x'_1 = x_1{+}kx_2 \wedge x'_2 = x_2 \wedge X_0(x_1, x_2) \wedge \forall 0 \leqslant k' < k : G(x_1{+}k'x_2, x_2)\}$$

By substituting a dense coefficient α *for* k *(dense approximation) we obtain*

$$\begin{array}{rll} & \exists \alpha \in \mathbb{R}^{\geqslant 0} : & 0 \leqslant x_2 = x_1 \wedge x'_1 = x_1 + \alpha x_2 \wedge x'_2 = x_2 \\ = & \exists \alpha \in \mathbb{R}^{\geqslant 0} : & 0 \leqslant x'_2 \leqslant x'_1 \wedge x'_1 = x'_2 + \alpha x'_2 \\ = & \exists \alpha \in \mathbb{R}^{\geqslant 0} : & 0 \leqslant x'_2 \leqslant x'_1 \wedge x'_2 = \frac{1}{1+\alpha} x'_1 \\ = & & x'_1 = x'_2 = 0 \vee 0 < x'_2 \leqslant x'_1 \end{array}$$

We observe that the topological closure of this set, i.e., $0 \leqslant x'_2 \leqslant x'_1$*, is a convex polyhedron. Hence, a precise abstract acceleration should be possible.*

Fig. 6.2 illustrates the example with a guard and the result we obtain by abstract acceleration using Prop. 6.1 which we are going to formulate in §6.3.

6.1 Linear Accelerability of Linear Transformations

We propose the following definition of linear accelerability:

Definition 6.1 (Linear accelerability) *A transition* τ *is linearly accelerable iff its reflexive and transitive closure* τ^* *can be written as a finite union of sets*

$$\tau^* = \lambda X. \bigcup_l \{\mathbf{A}_l \boldsymbol{x} + k\boldsymbol{b}_l \mid k \geqslant 0, \boldsymbol{x} \in X\}$$

In Def. 4.2 we gave the accelerability criterion based on the matrix $\mathbf{C}$ of an affine transformation $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$, *i.e.*, $\{\mathbf{C}^k \mid k \geqslant 0\}$ is finite ("finite monoid"). This characterization is merely based on the matrix $\mathbf{C}$ and it does not takes into account possible correlations between the coefficients of $\mathbf{C}$ and $\boldsymbol{d}$, neither the initial set X.

In this chapter, we give an alternative characterization based on the *homogeneous* form of affine transformations: any affine transformation of dimension n can be written as a linear transformation

$$\begin{pmatrix} \boldsymbol{x}' \\ x'_{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{C} & \boldsymbol{d} \\ \mathbf{0} & 1 \end{pmatrix}}_{\mathbf{C}'} \begin{pmatrix} \boldsymbol{x} \\ x_{n+1} \end{pmatrix}$$

of dimension $n+1$ and with $x_{n+1}=1$ in the initial set X.

Our criterion considers the Jordan normal form $\overline{\mathbf{J}} \in \mathbb{C}^n$ of $\mathbf{C}' \in \mathbb{R}^n$ which can be obtained by a similarity transformation $\overline{\mathbf{J}} = \mathbf{Q}^{-1}\mathbf{C}'\mathbf{Q}$ with a nonsingular matrix $\mathbf{Q} \in \mathbb{C}^n$ as described in §4.1. $\overline{\mathbf{J}}$ is a block diagonal matrix consisting of Jordan blocks $\mathbf{J}_i$ associated with the eigenvalues $\lambda_i \in \mathbb{C}$ of $\mathbf{C}'$.

Furthermore, we have $\overline{\mathbf{J}}^k = \mathbf{Q}^{-1}\mathbf{C}'^k\mathbf{Q}$ and thus:

$$\overline{\mathbf{J}}^k = \begin{pmatrix} \mathbf{J}_1^k & \dots & \mathbf{0} \\ \dots & \ddots & \dots \\ \mathbf{0} & \dots & \mathbf{J}_j^k \end{pmatrix}$$

We will now examine the linear accelerability of a Jordan block w.r.t. its size and associated eigenvalue:

Lemma 6.1 (Jordan block of size 1) *A transition $\tau(X) : \boldsymbol{x}' = \mathbf{J}\boldsymbol{x}$ where $\mathbf{J}$ is a Jordan block of size 1 is linearly accelerable iff its associated eigenvalue is either zero or a complex root of unity,* i.e., *$\lambda \in \{0\} \cup \{e^{i2\pi\frac{q}{p}} \mid p, q \in \mathbb{N}\}$.*

Proof A Jordan block of size $m = 1$ consists of its associated eigenvalue: $\mathbf{J} = (\lambda)$. The linear acceleration for τ is thus

$$\tau^*(X) = \bigcup_{k \geqslant 0} \{\lambda^k x \mid x \in X\} = \bigcup_l \{a_l x + b_l k \mid k \geqslant 0, x \in X\}$$

This means that we have to look for values of λ such that there is a finite number of values for the coefficients a_l, b_l as solutions of the equation $\forall k \geqslant 0, x \in X : \lambda^k x = a_l x + b_l k$. We distiniguish cases according to k:

- $k = 0$: $\lambda^0 = 1$ for any λ, hence $a_0 = 1, b_0 = 0$.
- $k \geqslant 1$: By writing λ^k in polar form $\rho^k e^{i\theta k}$ we get the solutions

  $$\begin{cases} \lambda = 0, & a_0 = 0, & b_0 = 0 & (i.e.,\ 0^k e^{i\theta k} = 0x + 0k) \\ \lambda = e^{i2\pi\frac{q}{p}}, & a_k = e^{i2\pi\frac{q}{p}k}, & b_k = 0, p, q \in \mathbb{N} & (i.e.,\ 1^k e^{i2\pi\frac{q}{p}k} x = e^{i2\pi\frac{q}{p}k} x + 0k). \end{cases}$$

  For the second solution, the number of values for a_k is finite because they are periodic: $e^{i2\pi\frac{q}{p}} = e^{i(2\pi\frac{q}{p} + 2\pi q j)}, j \in \mathbb{Z}$. Hence, there are p distinct values for a_k with $0 \leqslant k \leqslant p-1$.

This yields:

$\lambda = 0$: $\quad \tau^* = \lambda X.X \cup \{0\}$

$\lambda = e^{i2\pi\frac{q}{p}}$: $\quad \tau^* = \lambda X. \bigcup_{0 \leqslant l \leqslant p-1} \{\left(e^{i2\pi\frac{q}{p}l} x\right) \mid x \in X\}$ ∎

Lemma 6.2 (Jordan block of size 2) *A transition $\tau(X) : \boldsymbol{x}' = \mathbf{J}\boldsymbol{x}$ where $\mathbf{J}$ is a Jordan block of size 2 is linearly accelerable iff its associated eigenvalue is*

– *either zero ($\lambda = 0$) or*
– *a complex root of unity ($\lambda \in \{e^{i2\pi\frac{q}{p}} \mid p,q \in \mathbb{N}\}$) and if in this case the variable associated to the second dimension of the block has only a finite number of values in X.*

Proof A Jordan block of size $m = 2$ has the form $\mathbf{J} = \begin{pmatrix} \lambda & 1 \\ 0 & \lambda \end{pmatrix}$. The linear acceleration for τ is

$$\begin{aligned} \tau^*(X) &= X \cup \bigcup_{k\geqslant 1} \left\{ \begin{pmatrix} \lambda^k & k\lambda^{k-1} \\ 0 & \lambda^k \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \mid \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in X \right\} \\ &= \bigcup_l \left\{ \begin{pmatrix} a_{l,1} & a_{l,2} \\ a_{l,3} & a_{l,4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + k \begin{pmatrix} b_{l,1} \\ b_{l,2} \end{pmatrix} \mid k\geqslant 0, \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in X \right\} \end{aligned}$$

The case $k = 0$ is the identity for any eigenvalue λ, thus, we concentrate on $k \geqslant 1$: We have to find values of λ such that there is a finite number of values for the coefficients $a_{l,\cdot}, b_{l,\cdot}$ in the equation

$$\forall k\geqslant 1, \boldsymbol{x} \in X : \left\{ \begin{array}{rcl} \lambda^k x_1 + k\lambda^{k-1} x_2 &=& a_{l,1}x_1 + a_{l,2}x_2 + b_{l,1}k \\ \lambda^k x_2 &=& a_{l,3}x_1 + a_{l,4}x_2 + b_{l,2}k \end{array} \right.$$

We distinguish cases by values of k:

- $k = 1$: The left-hand side of the first equation reduces to $\lambda x_1 + x_2$. Hence, we can match left and right-hand sides for any values of λ: $a_{0,1} = a_{0,4} = \lambda, a_{0,2} = 1$, all other coefficents are 0.
- $k \geqslant 2$: In this case we cannot match $k\lambda^{k-1}x_2$ with $a_{l,2}x_2$, but we can match it with $b_{l,1}k$ under the assumption that x_2 has a finite number of values in X, which gives us (besides $\lambda = 0$) the following solution ($p, q \in \mathbb{N}$):

$$\forall k\geqslant 2, \boldsymbol{x} \in X : \left\{ \begin{array}{rcl} \underbrace{e^{i2\pi\frac{q}{p}k}}_{\lambda^k} x_1 + k \underbrace{e^{i2\pi\frac{q}{p}(k-1)}}_{\lambda^{k-1}} x_2 &=& \underbrace{e^{i2\pi\frac{q}{p}k}}_{a_{kx_2,1}} x_1 + \underbrace{0}_{a_{kx_2,2}} \cdot x_2 + \underbrace{e^{i2\pi\frac{q}{p}(k-1)}x_2}_{b_{kx_2,1}} k \\ \underbrace{e^{i2\pi\frac{q}{p}k}}_{\lambda^k} x_2 &=& \underbrace{0}_{a_{kx_2,3}} \cdot x_1 + \underbrace{e^{i2\pi\frac{q}{p}k}}_{a_{kx_2,4}} x_2 + \underbrace{0}_{b_{kx_2,2}} \cdot k \end{array} \right.$$

  As in the proof for Lem. 6.1 there is a finite number of values for $e^{i2\pi\frac{q}{p}k}$ because it is periodic.

This yields:

$$\lambda = 0: \qquad \tau^* = \lambda X.X \cup \left\{ \begin{pmatrix} x_2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right\}$$

$$\lambda = e^{i2\pi\frac{q}{p}}: \quad \tau^* = \left\{ \begin{array}{l} \lambda X.X \cup \\ \bigcup_{1\leqslant l\leqslant p, x_2} \left\{ \begin{pmatrix} e^{i2\pi\frac{q}{p}l}x_1 + kle^{i2\pi\frac{q}{p}(l-1)}x_2 \\ e^{i2\pi\frac{q}{p}l}x_2 \end{pmatrix} \mid \boldsymbol{x} \in X, k\geqslant 0 \right\} \end{array} \right.$$

∎

Lemma 6.3 (Jordan block of size > 2) *A transition $\tau(X) : \boldsymbol{x}' = \mathbf{J}\boldsymbol{x}$ where $\mathbf{J}$ is a Jordan block of size > 2 is linearly accelerable iff its associated eigenvalue is zero.*

Proof Jordan blocks of size $m > 2$ have the form: $\mathbf{J} = \begin{pmatrix} \lambda & 1 & 0 & \dots & 0 \\ 0 & \lambda & 1 & \dots & 0 \\ \dots & \dots & \ddots & \ddots & \dots \\ 0 & \dots & 0 & \lambda & 1 \\ 0 & \dots & 0 & 0 & \lambda \end{pmatrix}$.

Their powers have the form:

$$\mathbf{J}^k = \begin{pmatrix} \lambda^k & C_1^k\lambda^{k-1} & C_2^k\lambda^{k-2} & \dots & C_{m-1}^k\lambda^{k-m+1} \\ 0 & \lambda^k & C_1^k\lambda^{k-1} & \dots & C_{m-2}^k\lambda^{k-m+2} \\ \dots & \dots & \ddots & \ddots & \dots \\ 0 & \dots & 0 & \lambda^k & C_1^k\lambda^{k-1} \\ 0 & \dots & 0 & 0 & \lambda^k \end{pmatrix}$$

where C_j^k are the binomial coefficients.

For any value of $\lambda \neq 0$ we have polynomials in k of order m, hence we cannot match the coefficients with a linear form in k.

Thus, we have only $\lambda = 0$: $\tau^* = \lambda X. \bigcup_{0 \leqslant k \leqslant m} \{\mathbf{J}^k \boldsymbol{x} \mid \boldsymbol{x} \in X\}$ ∎

We summarize these results:

Theorem 6.1 (Accelerable linear transformations) *A linear transformation $\tau(X) : \boldsymbol{x}' = \mathbf{C}'\boldsymbol{x}$ is linearly accelerable iff its Jordan form consists of Jordan blocks $\mathbf{J}$ satisfying the following criteria:*

- $\mathbf{J}$ *is of size 1 and its associated eigenvalue* $\lambda \in \{0\} \cup \{e^{i2\pi\frac{q}{p}} \mid p, q \in \mathbb{N}\}$.
- $\mathbf{J}$ *is of size 2 and its associated eigenvalue* $\lambda = 0$ *or* $\lambda \in \{e^{i2\pi\frac{q}{p}} \mid p, q \in \mathbb{N}\}$, *and in the latter case the variable associated with the second dimension of the block has only a finite number of values in X in the Jordan basis.*
- $\mathbf{J}$ *is of size greater than 2 and its associated eigenvalue* $\lambda = 0$.

6.2 Comparison with Finite Monoid Acceleration

The background of the "finite monoid" criterion of Def. 4.2 is the following characterization of Boigelot (Theorem 8.53, [Boi98]): $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d}$ is accelerable if $\exists q > 0$ such that $\mathbf{C}^q$ is diagonizable and all its eigenvalues are in $\{0, 1\}$. In other words: the eigenvalues are either zero or roots of unity and all Jordan blocks of non-zero eigenvalues have size 1.

Theorem 6.2 (Jordan form of finite monoid affine transformations) *The Jordan form of the homogeneous transformation matrix* $\begin{pmatrix} \mathbf{C} & \boldsymbol{d} \\ \mathbf{0} & 1 \end{pmatrix}$, *where* $\{\mathbf{C}^k \mid k \geqslant 0\}$ *is finite, consists of*

- *Jordan blocks of size 1 with eigenvalues which are complex roots of unity,*
- *at most one block of size 2 with eigenvalue* 1 *where the variable associated with the second dimension is a constant equal to* 1, *and*
- *blocks with eigenvalue* 0 *of any size.*

Proof We will show that

(1) extending $\mathbf{C}$ to the homogeneous form adds the eigenvalue 1 to the spectrum. Hence, only the Jordan blocks in the Jordan form of $\mathbf{C}$ associated with an eigenvalue 1 are affected by homogenization, and

(2) the Jordan form of the homogenized matrix has at most one Jordan block of size 2 associated to an eigenvalue 1.

(1) follows from the fact that the characteristic polynomial of the homogeneous form is the characteristic polynomial of $\mathbf{C}$ multiplied by $1-\lambda$:

$$det\begin{pmatrix} \mathbf{C}-\lambda\mathbf{I} & \boldsymbol{d} \\ \mathbf{0} & 1-\lambda \end{pmatrix} = (1-\lambda)\cdot det(\mathbf{C}-\lambda\mathbf{I})$$

(because the left-hand side matrix is triangular). The variable corresponding to the dimension added during homogenization is known to equal the constant 1.

(2) We show that the Jordan form of $\mathbf{C}'$ has at most one Jordan block of size 2 associated with an eigenvalue 1: Assume that the Jordan form of the $(n-1)$-dimensional matrix $\mathbf{C}$ has $m-1$ blocks of size 1 associated with eigenvalue 1. Then, the homogeneous form $\mathbf{C}'$ has

- exactly 1 Jordan block of size 2 (with eigenvalue 1) and $m-2$ blocks of size 1 (with eigenvalue 1) if $ker(\mathbf{C}'-\mathbf{I})$ has dimension $m-1$, *i.e.*, $rank(\mathbf{C}'-\mathbf{I}) = n-m+1$;
- no Jordan block of size 2 (with eigenvalue 1) and m blocks of size 1 (with eigenvalue 1) if $ker(\mathbf{C}'-\mathbf{I})$ has dimension m, *i.e.*, $rank(\mathbf{C}'-\mathbf{I}) = n-m$.

Since the eigenvalues 1 of $\mathbf{C}$ have all geometric multiplicity 1, $\mathbf{C}-\mathbf{I}$ has $n-m$ linearly independent column vectors. Hence, $\mathbf{C}'-\mathbf{I}$ has $n-m$ linearly independent column vectors iff the additional column vector $\begin{pmatrix} \boldsymbol{d} \\ 0 \end{pmatrix}$ is not linearly independent from the others; otherwise it has $n-m+1$ linearly independent column vectors. Hence, we have $n-m \leqslant rank(\mathbf{C}'-\mathbf{I}) \leqslant n-m+1$. ∎

Example 6.2 (Jordan form characterization) *Consider* $\mathbf{C} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ *and* $\boldsymbol{d} = (0,1,2)^T$. $\mathbf{C}$ *has the eigenvalues* $\{1, -\frac{1}{2} \pm \frac{1}{2}i\sqrt{3}\}$. *Hence, the homogeneous form* $\mathbf{C}'$ *has 4 dimensions* ($n=4$) *and an eigenvalue* 1 *with algebraic multiplicity* $m=2$.

The matrix $\mathbf{C}'-\mathbf{I} = \begin{pmatrix} -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & 1 \\ 1 & 0 & -1 & 2 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ *has* $n-m+1=3$ *linearly independent column vectors, hence the Jordan form has one block with size 2 associated with the eigenvalue* 1.

Assume that $\boldsymbol{d}=(-1,0,1)^T$, *then* $\mathbf{C}'-\mathbf{I}$ *would have only* $n-m=2$ *linearly independent column vectors, hence the Jordan form would have 2 blocks of size 1 associated with the eigenvalue* 1.

Table 6.1 gives the Jordan form characterization of the homogeneous transformation for the different types of transformations.

Non-finite monoid case.

The only Jordan blocks of size two with non-zero eigenvalue that are considered accelerable by the finite monoid characterization are those with eigenvalue 1 and the second dimension of the block equal to 1 in the initial set X (w.r.t. the Jordan basis).

The criterion of Thm. 6.1 is slightly *more general*: It identifies furthermore those transformations as accelerable where the eigenvalue is a root of unity and the second dimension of the block has a finite set of values in the initial set X.

We give two examples for such transformations and show how to accelerate them:

transition type	ult. per.	Jordan form
Translations $\boldsymbol{x}' = \mathbf{I}_{(n)}\boldsymbol{x} + \boldsymbol{d},\ \boldsymbol{d} \neq \mathbf{0}$	$p=0, l=1$	• 1 translation block • $n-1$ identity blocks
Translations with resets $\boldsymbol{x}' = \begin{pmatrix} \mathbf{0}_{(m)}\mathbf{0} \\ \mathbf{0} \quad \mathbf{I}_{(n-m)} \end{pmatrix} \boldsymbol{x} + \begin{pmatrix} \boldsymbol{d} \\ \boldsymbol{d}' \end{pmatrix},\ \boldsymbol{d}' \neq \mathbf{0}$	$p=1, l=1$	• m nilpotent blocks of size 1 • 1 translation block • $n-m-1$ identity blocks
Purely periodic transformations $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d},\ \mathbf{C}^l = \mathbf{I}$	$p=0$	• rotation (by $\theta = 2\pi\frac{q_i}{p_i}$) blocks • zero or one translation blocks (depending on $\boldsymbol{d}$)
Ultimately periodic transformations $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x} + \boldsymbol{d},\ \mathbf{C}^{p+l} = \mathbf{C}^p$	$p>0$	• rotation (by $\theta = 2\pi\frac{q_i}{p_i}$) blocks • nilpotent blocks with maximum size p • zero or one translation blocks (depending on $\boldsymbol{d}$)

Legend:

ult. per.: ultimate periodicity characterization $\mathbf{C}^{p+l} = \mathbf{C}^p$, $p \geqslant 0, l > 0$

Jordan form: direct product of
- identity block $\lambda = 1$, size 1
- translation block $\lambda = 1$, size $2, 2^{nd}$ dimension $\equiv 1$
- nilpotent block $\lambda = 0$, size m ($\mathbf{J}^m = \mathbf{0}$)
- rotation block $\lambda = e^{i2\pi\frac{q}{p}}$, size 1

$\mathbf{A}_{(k)}$ denotes a square matrix of size k.

Table 6.1: Jordan form characterization of finite monoid transformations

Example 6.3 (Dependencies on modified variables II)

$$\tau : x_1 + 2x_2 \leqslant 6 \rightarrow \boldsymbol{x}' = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \boldsymbol{x} \text{ with } X_0 = (x_1 = 0 \wedge x_2 \in \{2,3,4\})$$

We can exactly accelerate the loop τ by enumerating the values of x_2 in X_0 and translating x_1 by each of these values:

$$\begin{aligned} \tau^*(X_0) = \ & \{(x_1', 2) \mid \exists k \geqslant 0 : x_1' = x_1 + 2k \wedge X_0(\boldsymbol{x}) \wedge \forall 0 \leqslant k' < k : x_1' \leqslant 2\} \cup \\ & \{(x_1', 3) \mid \exists k \geqslant 0 : x_1' = x_1 + 3k \wedge X_0(\boldsymbol{x}) \wedge \forall 0 \leqslant k' < k : x_1' \leqslant 0\} \cup \\ & \{(x_1', 4) \mid \exists k \geqslant 0 : x_1' = x_1 + 4k \wedge X_0(\boldsymbol{x}) \wedge \forall 0 \leqslant k' < k : x_1' \leqslant -2\} \\ = \ & \{(0,2), (2,2), (4,2), (0,3), (3,3), (6,3), (0,4)\} \end{aligned}$$

Example 6.4 (Rotation with translation by rotated variables) *The linear transformation*

$$\mathbf{C} = \begin{pmatrix} 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

is a rotation by $\frac{\pi}{2}$ in dimensions 3 and 4, and it also rotates dimensions 1 and 2 by $\frac{\pi}{2}$ while translating them by dimensions 3 and 4 respectively.

$\mathbf{C}$ *has conjugate complex eigenvalues $(\pm i)^4 = 1$ with algebraic multiplicity 2 and geometric multiplicity* 1, i.e., *it is not finite-monoidal. The Jordan form is*

$$\overline{\mathbf{J}} = \begin{pmatrix} -i & 1 & 0 & 0 \\ 0 & -i & 0 & 0 \\ 0 & 0 & i & 1 \\ 0 & 0 & 0 & i \end{pmatrix}$$

Let us consider the loop $\tau : -9 \leqslant x_1 + x_2 \leqslant 9 \wedge -5 \leqslant x_1 \leqslant 10 \to \boldsymbol{x}' = \mathbf{C}\boldsymbol{x}$ with $X_0 = \{(0,0,1,2)\}$. Since the values for x_3 and x_4 are finite in X_0, the transition is accelerable according to Thm. 6.1.

The matrix $\mathbf{C}$ is of the form $\mathbf{C} = \begin{pmatrix} \mathbf{J} & \mathbf{I} \\ \mathbf{0} & \mathbf{J} \end{pmatrix}$, thus its powers have the form $\mathbf{C}^k = \begin{pmatrix} \mathbf{J}^k & k\mathbf{J}^{k-1} \\ \mathbf{0} & \mathbf{J}^k \end{pmatrix}$. Moreover, $\mathbf{J}$ has periodicity $p = 4$.

With this information, $\boldsymbol{x}' = \mathbf{C}\boldsymbol{x}$ can be accelerated by enumerating the powers of $\mathbf{J}$ and the values of x_3, x_4 in X_0 as follows (cf. proof of Lem. 6.2):

$$(\boldsymbol{x}' = \mathbf{C}\boldsymbol{x})^* = \lambda X.X \cup \bigcup_{1 \leqslant l \leqslant p, x_3, x_4} \left\{ \begin{pmatrix} \mathbf{J}^l \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + kl\mathbf{J}^{(l-1)} \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} \\ \mathbf{J}^l \begin{pmatrix} x_3 \\ x_4 \end{pmatrix} \end{pmatrix} \mid \boldsymbol{x} \in X, k \geqslant 0 \right\}$$

Using this formula and taking into account the guard G, τ^* is computed by

$$\begin{aligned}
\tau^*(X_0) = \quad & \{(x_1', x_2', 1, 2) \quad \mid \exists k \geqslant 0 : x_1' = x_1^k \wedge x_2' = x_2^k \wedge \\
& \quad x_1^k = x_1 + 2 \cdot 4k \wedge x_2^k = x_2 - 1 \cdot 4k \\
& \quad \wedge X(x_1, x_2, 1, 2) \wedge \forall 0 \leqslant k' < k : G(x_1^{k'}, x_2^{k'}, 1, 2)\} \cup \\
& \{(x_1', x_2', -2, 1) \quad \mid \exists k \geqslant 0 : x_1' = x_1^k \wedge x_2' = x_2^k \wedge \\
& \quad x_1^k = -x_2 + 1(4k+1) \wedge x_2^k = x_1 + 2(4k+1) \\
& \quad \wedge X(x_1, x_2, -2, 1) \wedge \forall 0 \leqslant k' < k : G(x_1^{k'}, x_2^{k'}, -2, 1)\} \cup \\
& \{(x_1', x_2', -1, -2) \quad \mid \exists k \geqslant 0 : x_1' = x_1^k \wedge x_2' = x_2^k \wedge \\
& \quad x_1^k = -x_1 - 2(4k+2) \wedge x_2^k = -x_2 + 1(4k+2) \\
& \quad \wedge X(x_1, x_2, -1, -2) \wedge \forall 0 \leqslant k' < k : G(x_1^{k'}, x_2^{k'}, -1, -2)\} \cup \\
& \{(x_1', x_2', 2, -1) \quad \mid \exists k \geqslant 0 : x_1' = x_1^k \wedge x_2' = x_2^k \wedge \\
& \quad x_1^k = x_2 - 1(4k+3) \wedge x_2^k = -x_1 - 2(4k+3) \\
& \quad \wedge X(x_1, x_2, 2, -1) \wedge \forall 0 \leqslant k' < k : G(x_1^{k'}, x_2^{k'}, 2, -1)\}
\end{aligned}$$

We obtain the result

$$\begin{aligned}
\tau^*(X_0) = \quad & \{(0,0,1,2), (8,-4,1,2), (1,2,-2,1), (5,10,-2,1), \\
& (-4,2,-1,-2), (-3,-6,2,-1)\}
\end{aligned}$$

See Fig. 6.3 for a plot of these points.

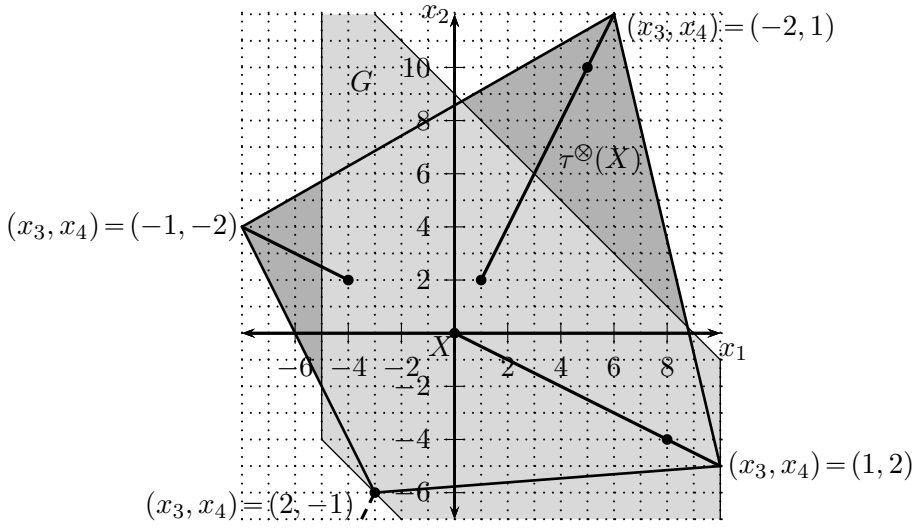


Figure 6.3: Rotation with translation by rotated variables: reachable integer points and polyhedron computed by abstract acceleration (dark gray, bold frame) projected on plane x_1, x_2; guard G (light gray), the four sets involved in the union (bold lines) with their associated values of x_3 and x_4.

6.3 Generalizing Abstract Acceleration

Thanks to the work of Gonnord et al. [GH06, Gon07], abstract acceleration can handle finite monoid transformations. In this section, we will generalize abstract acceleration to the additional case of linearly accelerable transformations we identified in the previous sections, *i.e.*, *Jordan blocks of size two* with eigenvalues that are roots of unity.

We consider transitions of the form $\tau : \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \to \boldsymbol{x}' = \underbrace{\begin{pmatrix} \mathbf{J} & \mathbf{I} \\ \mathbf{0} & \mathbf{J} \end{pmatrix}}_{\mathbf{C}} \boldsymbol{x}$ where the transformation matrix is in real Jordan form (§4.1), *i.e.*, $\mathbf{J} = \begin{pmatrix} \cos 2\pi\frac{q}{p} & -\sin 2\pi\frac{q}{p} \\ \sin 2\pi\frac{q}{p} & \cos 2\pi\frac{q}{p} \end{pmatrix}$ for conjugate complex eigenvalues $e^{2\pi\frac{q}{p}}$, $p, q \in \mathbb{N}$, and $\mathbf{J} = (1)$ for the eigenvalue 1.

We use notations similar to §5.1.2 where we partitioned the dimensions into translated and reset dimensions. Here, we distinguish the translated dimensions T from the non-translated ones N. In the case of conjugate complex eigenvalues, T is $\{1, 2\}$ and $N = \{3, 4\}$, for the eigenvalue 1 we have $T = \{1\}$ and $N = \{2\}$. Then, $X^{\bullet,N}$ for example, denotes the projection of the polyhedron X onto the non-translated dimensions. $\boldsymbol{x}^T$ and $\boldsymbol{x}^N$ denote the subvectors of translated and non-translated dimensions respectively.

Proposition 6.1 *Let* $\tau : \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b} \to \boldsymbol{x}' = \begin{pmatrix} \mathbf{J} & \mathbf{I} \\ \mathbf{0} & \mathbf{J} \end{pmatrix} \boldsymbol{x}$ *be a transition as explained above. Then*

$$\tau^{\otimes}(X) = X \sqcup \bigsqcup_{0 \leqslant l \leqslant p-1} \tau\big((\tau^l(X) \sqcap G) \nearrow D^{(l)}\big)$$

is a sound over-approximation of τ^**, where*

- τ^l *is expressed as in Eq. 4.3,* i.e.*:*

$$\tau^l = \bigwedge_{0\leqslant i\leqslant l-1} \left(\mathbf{A}\mathbf{C}^i\boldsymbol{x} + \sum_{0\leqslant j\leqslant i-1} \mathbf{C}^j\boldsymbol{d} \leqslant \boldsymbol{b} \right) \rightarrow \boldsymbol{x}' = \mathbf{C}^l\boldsymbol{x} + \sum_{0\leqslant j\leqslant l-1} \mathbf{C}^j\boldsymbol{d}$$

- $D^{(l)} = \begin{pmatrix} \mathbf{0} & \mathbf{J}^{l'} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \cdot \left((X \sqcap G)^{\bullet,N}\right)$ *with* $l' = (l-1) \mod p$.

This means that we enumerate the powers of $\mathbf{J}$ and translate the dimensions T by the polyhedron $D^{(l)}$ corresponding to the $(l\text{–}1)^{th}$ power of $\mathbf{J}$. The polyhedron $D^{(l)}$ originates from the right upper block in $\mathbf{C}^k\boldsymbol{x} = \begin{pmatrix} \mathbf{J}^k\boldsymbol{x}^T & k\mathbf{J}^{k-1}\boldsymbol{x}^N \\ \mathbf{0} & \mathbf{J}^k\boldsymbol{x}^N \end{pmatrix}$ where it over-approximates the set $\{\mathbf{J}^{l-1}\boldsymbol{x}^N \mid \boldsymbol{x} \in X\}$ by which the dimensions T are translated.

Proof
The formula is trivially correct for 0 iterations. It remains to show that our formula yields an over-approximation for $k \geqslant 1$ iterations:

$$\begin{array}{ll}
 & \boldsymbol{x}' \in \bigcup_{k\geqslant 1} \tau^k(X) \\
\iff & \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_k : \\
 & \boldsymbol{x}' \in \tau(\boldsymbol{x}_k) \wedge \boldsymbol{x}_k = \mathbf{C}^k\boldsymbol{x}_0 \wedge \forall k' \in [0,k] : G(\boldsymbol{x}_{k'}) \\
\iff & \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_{pk+l}, \exists l \in [0,p-1] : \\
 & \boldsymbol{x}' = \bigcup_l \tau(\boldsymbol{x}_{pk+l}) \wedge \boldsymbol{x}_{pk+l} = \mathbf{C}^{pk+l}\boldsymbol{x}_0 \wedge \forall k' \in [0,pk+l] : G(\boldsymbol{x}_{k'}) \\
\iff & \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_{pk+l}, \exists l \in [0,p-1] : \\
 & \boldsymbol{x}' = \bigcup_l \tau(\boldsymbol{x}_{pk+l}) \wedge \boldsymbol{x}^T_{pk+l} = \mathbf{J}^l\boldsymbol{x}^T_0 + (pk+l)\mathbf{J}^{l'}\boldsymbol{x}^N_0 \wedge \boldsymbol{x}^N_{pk+l} = \mathbf{J}^l\boldsymbol{x}^N_0 \wedge \\
 & \forall k' \in [0,pk+l] : G(\boldsymbol{x}_{k'}) \\
 & (\text{with } l' = (l-1) \mod p) \\
\implies & \exists k \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_{pk+l}, \exists l \in [0,p-1], \exists \boldsymbol{d}^{(l)T} \in D^{(l)T} = \mathbf{J}^{l'}(X \sqcap G)^N : \\
 & \boldsymbol{x}' = \bigcup_l \tau(\boldsymbol{x}_{pk+l}) \wedge \boldsymbol{x}_l = \tau^l(\boldsymbol{x}_0) \wedge \boldsymbol{x}^T_{pk+l} = \boldsymbol{x}^T_l + (pk+l)\boldsymbol{d}^{(l)T} \wedge \boldsymbol{x}^N_{pk+l} = \boldsymbol{x}^N_l \wedge \\
 & \forall k' \in [0,pk+l] : G(\boldsymbol{x}_{k'}) \\
 & (\text{over-approximation of } \mathbf{J}^{l'}\boldsymbol{x}^N_l \text{ by } \boldsymbol{d}^{(l)T}) \\
\implies & \exists \alpha \geqslant 0, \exists \boldsymbol{x}_0 \in X, \exists \boldsymbol{x}_{pk+l}, \exists l \in [0,p-1], \exists \boldsymbol{d}^{(l)T} \in D^{(l)T} = \mathbf{J}^{l-1}(X \sqcap G)^N : \\
 & \boldsymbol{x}' = \bigsqcup_l \tau(\boldsymbol{x}_{pk+l}) \wedge \boldsymbol{x}_l = \tau^l(\boldsymbol{x}_0) \wedge \boldsymbol{x}^T_{pk+l} = \boldsymbol{x}^T_l + \alpha\boldsymbol{d}^{(l)T} \wedge \boldsymbol{x}^N_{pk+l} = \boldsymbol{x}^N_l \wedge \\
 & \bigwedge_l G(\boldsymbol{x}_l) \wedge G(\boldsymbol{x}_{pk+l}) \\
 & (\text{dense and convex approximations}) \\
\iff & \boldsymbol{x}' \in \bigsqcup_{0\leqslant l\leqslant p-1} \tau\left((\tau^l(X) \sqcap G) \nearrow D^{(l)}\right)
\end{array}$$

∎

The first approximation ($\implies$) in the proof due to the projection on the non-translated dimensions occurs only if the non-translated dimensions are not independent from the translated dimensions in the initial set X.

Mind that we do not require as in Thm. 6.1 that the variables corresponding to the non-translated dimensions have a finite number of values in the initial set: this generalization is justified by the dense approximation that we perform (second $\implies$ in the proof).

Figures 6.2 and 6.3 depict the result of the application of Prop. 6.1 to Examples 6.1 and 6.4 respectively.

6.4 Conclusions and Perspectives

In this chapter we have seen that the Jordan normal form is a powerful theoretical tool which made us gain more insight into the accelerability of linear transformations. However, there are two challenges in exploiting these results in a practical abstract acceleration approach:

First, we have to *build the Jordan form*, *i.e.*, we have to compute the eigenvalues of the transformation matrix and their algebraic and geometric multiplicities. Determining the eigenvalues requires finding the roots of the characteristic polynomial, which is a hard problem: by the theorem of Abel-Ruffini there is no solution based on radicals for polynomials of degree 5 and higher. Hence, in general only numerical approximations can be computed (see, *e.g.*, [Wil88]). However, for our characterization it is essential to know whether an eigenvalue (exactly) equals zero or one. Nevertheless, a solution might be possible, because we are only interested in eigenvalues which are 0 or roots of unity.

Second, we have to find and apply an abstract acceleration formula. We have to remark that our theorems and those of Gonnord et al. for abstract acceleration – in contrast to exact acceleration – require to *perform a basis change* in order to reduce the acceleration problem to the simple cases of translations, resets and the new case introduced in §6.3. However, for above-mentioned reasons this basis change is hard to implement: for instance also the tool Aspic [Gon] does not fully implement Thm. 4.6, but only the case $\mathbf{C}^2 = \mathbf{C}$, where no change of basis is necessary. It has not been considered so far, whether there is an abstract acceleration formula of periodic affine transformations for instance which can do without a basis change.

Detecting accelerable sub-transformations. Another direction for future research could be to detect the maximal linearly accelerable sub-transformations within a general linear transformation or even in a general transition function. Then, the transition function can be decomposed w.r.t. the accelerable dimensions which are treated using abstract acceleration and the non-accelerable dimensions which can be handled with widening or the derivative closure technique.

We will use a similar idea in §8 for accelerating transition functions involving numerical and Boolean variables.

Acceleration of not linearly accelerable transformations. At last, it would be of great interest to generalize the abstract acceleration concept to not linearly accelerable linear transformations: for example, to compute a reasonably simple, but precise, convex, polyhedral over-approximation of the transitive closure of the self-loop $x \leqslant 10 \rightarrow x' = 2x$ which has an exponential trajectory as a function of time. Computing precise approximations of such behavior is also of high importance in the analysis of hybrid systems (see §10), of which the time-continuous behavior is often specified by linear dynamics $\dot{\boldsymbol{x}} = \mathbf{C}\boldsymbol{x}$.

Part II

Verification of Logico-Numerical Systems

Chapter 7

Logico-Numerical Program Analysis: Our Approach

We call programs with numerical and Boolean variables *logico-numerical programs*. Synchronous data-flow programs, *e.g.*, LUSTRE programs, count among such programs. We model them as logico-numerical discrete transition systems (§7.1).

Symbolic representations (§7.2) for such programs and abstract domains for their analysis, as *e.g.*, implemented in the library BDDAPRON [Jea], make use of binary decision diagrams. Such logico-numerical abstract domains are constructed by combining Boolean and numerical abstract values so as to build product or power domains.

State space partitioning (§7.3) allows CFG representations of discrete dynamical systems to be generated. The overall abstract domain associated with a CFG is a power domain. Hence, the choice of the partition determines the abstract domain, and thus, influences precision and efficiency of the analysis.

Finally, we discuss related work w.r.t. the analysis of logico-numerical programs (§7.4).

7.1 Logico-Numerical Programs

We model logico-numerical programs as deterministic discrete transition systems (see §2.1.1). We include the assertion observer $\mathcal{A}$ (see §2.3) in the definition:

Definition 7.1 (Logico-numerical discrete transition system) *A logico-numerical discrete transition system* $\langle \Sigma, \Upsilon, \boldsymbol{f}, \mathcal{A}, \mathcal{I} \rangle$ *over the state space* Σ *and the input space* Υ *is defined as* $\begin{cases} \mathcal{I}(\boldsymbol{s}) \\ \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \end{cases}$ *where*
- $\boldsymbol{f} : \Sigma \times \Upsilon \rightarrow \Sigma$ *is the vector of transition functions,*
- $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \subseteq (\Sigma \rightarrow \wp(\Upsilon))$ *is an* assertion *constraining the inputs depending on the current state, and*
- $\mathcal{I}(\boldsymbol{s}) \subseteq \Sigma$ *defines the initial states.*

We use the following notations:

$\boldsymbol{s} = (\boldsymbol{b}, \boldsymbol{x})$	:	state variable vector, with $\boldsymbol{b}$ Boolean and $\boldsymbol{x}$ numerical subvectors
$\boldsymbol{i} = (\boldsymbol{\beta}, \boldsymbol{\xi})$	:	input variable vector, with $\boldsymbol{\beta}$ Boolean and $\boldsymbol{\xi}$ numerical subvectors
$\mathcal{C}(\boldsymbol{x}, \boldsymbol{\xi})$	:	vector of constraints over numerical variables (for short $\mathcal{C}$)

The semantics of such a system is defined in the style of Def. 2.2:

Definition 7.2 (Semantics) *An execution of such a system is a sequence*

$$\boldsymbol{s}_0 \xrightarrow{\boldsymbol{i}_0} \boldsymbol{s}_1 \xrightarrow{\boldsymbol{i}_1} \ldots \boldsymbol{s}_k \xrightarrow{\boldsymbol{i}_k} \ldots$$

such that $\mathcal{I}(\boldsymbol{s}_0)$ *and for any* $k \geqslant 0$*:* $\mathcal{A}(\boldsymbol{s}_k, \boldsymbol{i}_k) \wedge \big(\boldsymbol{s}_{k+1} = \boldsymbol{f}(\boldsymbol{s}_k, \boldsymbol{i}_k)\big)$.

Example 7.1 (Logico-numerical program) *The following program computes in* x_1 *the sum over the input* ξ *for 10 time steps after receiving an input* $\beta = \mathtt{tt}$ *provided that the input* ξ *is between* 1 *and* 3*.*

$$\left\{ \begin{array}{l} \mathcal{I}(b, x_1, x_2) = \neg b \wedge (x_1 = x_2 = 0) \\ \\ 1 \leqslant \xi \leqslant 3 \rightarrow \left(\begin{array}{c} b' \\ \\ x_1' \\ \\ x_2' \end{array} \right) = \left(\begin{array}{l} \neg b \wedge \beta \ \vee \ b \wedge x_2 < 9 \\ \left\{ \begin{array}{ll} x_1 + \xi & \textit{if } b \\ x_1 & \textit{else} \end{array} \right. \\ \left\{ \begin{array}{ll} 0 & \textit{if } \beta \\ x_2 + 1 & \textit{else} \end{array} \right. \end{array} \right) \end{array} \right.$$

We distinguish the Boolean and numerical components of the transition function:

$$\left(\begin{array}{c} \boldsymbol{b}' \\ \boldsymbol{x}' \end{array} \right) = \left(\begin{array}{c} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{array} \right)$$

A Boolean transition function is written as a Boolean formula φ involving Boolean variables and numerical constraints:

$$f^b(\boldsymbol{s}, \boldsymbol{i}) = \varphi(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})$$

A numerical transition function is written as a disjunction of guarded actions:

$$f^x(\boldsymbol{s}, \boldsymbol{i}) = \bigvee_j \big(a_j(\boldsymbol{x}, \boldsymbol{\xi}) \text{ if } g_j(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})\big)$$

with $\bigvee_j g_j$ and $\neg(g_i \wedge g_j)$ for $i \neq j$. The guards g_j are Boolean formulas involving Boolean variables and numerical constraints and the actions a_j are arithmetic expressions without tests. The program in Ex. 7.1 conforms to these notations.

7.2 Symbolic Representations

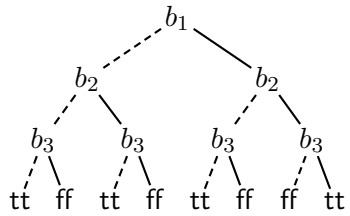
The library BddApron [Jea] provides symbolic representations for logico-numerical programs and abstract domains for their analysis.

It exploits the well-known efficiency of Bdds (§7.2.1) for representing logico-numerical formulas, functions (§7.2.2) and abstract values, (§7.2.3).

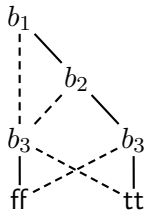
7.2.1 Introduction to Binary Decision Diagrams

Binary decision diagrams (Bdds) were introduced by Bryant [Bry86, Bry92] as a representation of Boolean functions $\mathbb{B}^m \to \mathbb{B}$. In this section we briefly summarize the most important concepts. For further details we refer to textbooks, *e.g.*, [MT98].

The basic idea is to recursively apply the Shannon expansion $f = f[b \leftarrow \mathtt{tt}] \vee f[b \leftarrow \mathtt{ff}]$ to the variables b occurring in f in order to build a decision tree (Fig. 7.1a): the



(a) Shannon decision tree



(b) Bdd (reduced, ordered binary decision diagram)

Figure 7.1: Representations of the Boolean function $f(b_1, b_2, b_3) = b_1 \wedge (b_2 \Leftrightarrow b_3) \vee \neg b_1 \wedge \neg b_3$: solid edge = tt, dashed edge = ff (cf. [Jea00]).

inner nodes of the tree are labeled with the variables b, the leaves (terminal nodes) are labeled with the truth values tt, ff, and the edges are labeled with the truth value of the decision on the parent node tt, ff. Such a decision tree can be structurally represented as a formula with the help of the if-then-else operator $\mathsf{ite}(b, f[b \leftarrow \mathsf{tt}], f[b \leftarrow \mathsf{ff}])$.

If the Shannon expansion is applied using the same order of variables on all branches, then the binary decision diagram is called *ordered* (OBDD).

In order to obtain a Bdd in the narrow sense, it must be compressed to a *reduced*, ordered binary decision diagram (ROBDD, Fig. 7.1b): (1) Identical subgraphs are merged, and (2) nodes of which both children are identical subtrees are removed.

The result is a directed, acyclic graph such that (a) all non-terminal nodes have two children, (b) there is a single root (which has no parent), and (c) there are two terminal nodes (without children).

Variable ordering. Given a variable ordering, a Bdd is a *canonical* representation of a Boolean function. However, the size of the Bdd depends on the ordering of the variables.

Example 7.2 (Variable ordering) *For example (cf. [Bry86]) the function*

$$f(b_1, \ldots, b_{2n}) = b_1 \wedge b_2 \vee b_3 \wedge b_4 \vee \ldots \vee b_{2n-1} \wedge b_{2n}$$

has
- $\Theta(2^n)$ *nodes with the variable ordering* $b_1 < b_3 < \ldots < b_{2n-1} < b_2 < b_4 < \ldots < b_{2n}$, *and*
- $\Theta(n)$ *nodes for* $b_1 < b_2 < \ldots < b_{2n-1} < b_{2n}$.

A rule of thumb is that variables related by conjunctions should be close to each other. Finding the best order, though, is an NP-complete problem [BW96]. In practice heuristics, *e.g.*, [Rud93], are used to find a variable ordering minimizing the size of a set of Bdds. Nonetheless, some functions, *e.g.*, the binary encoding of integer multiplication [Bry91], have always an exponential Bdd representation regardless of the variable ordering.

Operations. The operations are implemented by recursive traversal of the operand Bdds and by using hashtables to store and reuse already computed subgraphs (see [Bry86]). The basic operations are the following (the computational complexity is given in terms of the number of nodes n_i of the operands):
- apply_2 lifts any binary operator $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ like $\wedge$, $\vee$ or $\Rightarrow$ to Bdds: $(\mathbb{B}^m \to \mathbb{B}) \times (\mathbb{B}^m \to \mathbb{B}) \to (\mathbb{B}^m \to \mathbb{B})$. Complexity: $\mathcal{O}(n_1 n_2)$.

– apply_1 applies any unary operator $\mathbb{B} \to \mathbb{B}$, *e.g.*, $\neg$, to the terminal nodes of the BDD. Complexity: $\mathcal{O}(t)$, where t is the number of terminal nodes.
– $\mathsf{cofactor}\ f[b \leftarrow c]$ substitutes the constant c for the variable b. Complexity: $\mathcal{O}(n)$.

We can compose these operators to define:

– *Function composition*: $f_1[b \leftarrow f_2] = f_1[b \leftarrow \mathsf{tt}] \wedge f_2 \ \vee \ f_1[b \leftarrow \mathsf{ff}] \wedge \neg f_2$. Complexity: $\mathcal{O}(n_1^2 n_2)$.
– *Existential quantification* $\exists b.f = f[b \leftarrow \mathsf{tt}] \vee f[b \leftarrow \mathsf{ff}]$. Complexity: $\mathcal{O}(n^2)$.

Some other operations are trivial:

– *Equality* $f_1 \Leftrightarrow f_2$: since a BDD is a canonical representation of a Boolean function, it suffices to compare physical equality in memory (pointer equality). Complexity: $\mathcal{O}(1)$.
– *Validity* $f \Leftrightarrow \mathsf{tt}$ and *unsatisfiability* $f \Leftrightarrow \mathsf{ff}$ amounts to comparing with the constant BDDs for tt and ff respectively. Complexity: $\mathcal{O}(1)$.

Partial evaluation. An important operation in our context is the partial evaluation (see *e.g.*, [JGS93]) of a Boolean formula f by a Boolean formula g, denoted $f \uparrow g$. This can be seen as the generalization of the *cofactor* operation ($f[b \leftarrow c] = f \uparrow (b \Leftrightarrow c)$) to formulas.

Partial evaluation is implemented by a generalized cofactor operator [CBM89]:

Definition 7.3 (Generalized cofactor) *A generalized cofactor $f \uparrow g$ of a formula f w.r.t. a formula g is a formula h such that*

$$g \Rightarrow (f \Leftrightarrow h)$$

and h is a "smaller" formula than f.

We use the variant of the operator proposed by Raymond [Ray91].

Example 7.3 (Generalized cofactor) *Using this operator we have for example:*

$$(b_1 \wedge b_2) \uparrow (b_1 \vee \neg b_2) = b_2$$

We can easily verify the validity:
$$\left\{ \begin{array}{ll} & (b_1 \vee \neg b_2) \Rightarrow ((b_1 \wedge b_2) \Leftrightarrow b_2) \\ \iff & \neg b_1 \wedge b_2 \ \vee \ (b_1 \wedge b_2 \ \vee \ \neg b_1 \wedge \neg b_2 \ \vee \ \neg b_2) \\ \iff & \mathsf{tt} \end{array} \right.$$

Multi-Terminal Binary Decision Diagrams. These are BDDs with more than two terminal nodes [CMZ+93], *i.e.*, they represent functions $\mathbb{B}^m \to E$. E can be any type, but in order to preserve canonicity and for implementation reasons it must be equipped with an equality operator and a hash function.

Most operations can be transferred directly from BDDs to MTBDDs: for example, apply_2 lifts a binary operator $E_1 \times E_2 \to E$ to $(\mathbb{B}^m \to E_1) \times (\mathbb{B}^m \to E_2) \to (\mathbb{B}^m \to E)$.

However, some operations need to be adapted to be meaningful: *Existential quantification* for BDDs combines terminal nodes using $\vee$ (see above). In the case of MTBDDs we need to use an operator adapted to the type E: A generic solution for combining terminal nodes is the set union $\cup$: then, existential quantification becomes an operator transforming an MTBDD ($\mathbb{B}^m \to E$) into an MTBDD ($\mathbb{B}^m \to 2^E$). In the case where E is an abstract domain, for example, then the adequate operation is $\sqcup$.

Boolean expressions:

$\langle Bexpr \rangle ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle Bvar \rangle \mid \neg\langle Bexpr \rangle \mid \langle Bexpr \rangle\ (\wedge \mid \vee \mid \dots)\ \langle Bexpr \rangle$
$\quad \mid \quad \langle expr \rangle\ \mathtt{=}\ \langle expr \rangle \mid \langle Iexpr \rangle\ (\mathtt{<} \mid \mathtt{<=})\ \langle Iexpr \rangle \mid \langle Acons \rangle$

Arithmetic expressions:

$\langle Aexpr \rangle ::= cst \mid \langle Avar \rangle \mid (\mathtt{-}|\sqrt{})\langle Aexpr \rangle \mid \langle Aexpr \rangle\ (\mathtt{+}|\mathtt{-}|\mathtt{*}|\mathtt{/}|\mathtt{\%})\ \langle Aexpr \rangle$
$\quad \mid \quad \text{if } \langle Bexpr \rangle \text{ then } \langle Aexpr \rangle \text{ else } \langle Aexpr \rangle$

Arithmetic conditions:

$\langle Acons \rangle ::= \langle Aexpr \rangle\ (\mathtt{<} \mid \mathtt{<=})\ \langle Aexpr \rangle$

Enumerated types:

$\langle Eexpr \rangle ::= label \mid \langle Evar \rangle \mid \text{if } \langle Bexpr \rangle \text{ then } \langle Eexpr \rangle \text{ else } \langle Eexpr \rangle$

Bounded integers:

$\langle Iexpr \rangle ::= \langle cst \rangle \mid \langle Ivar \rangle \mid \langle Iexpr \rangle\ (\mathtt{+} \mid \mathtt{-} \mid \mathtt{*})\ \langle Iexpr \rangle \mid \langle Iexpr \rangle\ (\mathtt{<<} \mid \mathtt{>>})\ \mathrm{n}$
$\quad \mid \quad \text{if } \langle Bexpr \rangle \text{ then } \langle Iexpr \rangle \text{ else } \langle Iexpr \rangle$

Expressions:

$\langle expr \rangle ::= \langle Bexpr \rangle \mid \langle Eexpr \rangle \mid \langle Iexpr \rangle \mid \langle Aexpr \rangle$

Table 7.1: Expressions available in BddApron (subset).

Another operation which we are going to use is the *product* of MtBdds $(\mathbb{B}^m \to E_1) \times (\mathbb{B}^m \to E_2) \to (\mathbb{B}^m \to (E_1 \times E_2))$, which can again be implemented using the generic apply_2 operator.

The library CUDD. The library Cudd[1] implements Bdds, MtBdds and other variants of binary decision diagrams and many operations on them. The actual implementation of Bdds uses so-called *typed decision graphs* [Bil87] which allow Bdds to be compressed even further by merging subgraphs that are isomorphic up to negation. Moreover, isomorphic subgraphs are shared among the set of Bdds being manipulated.

7.2.2 Formulas and Functions

In this section we explain the representations and operations for logico-numerical formulas and functions offered by the BddApron library.

Variables of the following types are supported: Boolean, enumerated types, bounded integers and numerical types (integer, rationals, ...). Tab. 7.1 lists the (simplified) grammar of expressions of these types. Enumerated types ($Etype = label_1 | \dots | label_n$) and bounded integers are only syntactic sugar: they are encoded as bit arrays and thus treated as vectors of Booleans – in the following we will employ the term "Boolean" to refer to all these finite data types.

Hence, we can concentrate on four types of expressions:

[1] `http://vlsi.colorado.edu/~fabio/CUDD/`

– purely arithmetic expressions $\langle aAexpr \rangle$, *e.g.*, $x+1$;
– purely Boolean expressions $\langle bBexpr \rangle$, *e.g.*, $b_1 \wedge \neg b_2$;
– Boolean expressions $\langle Bexpr \rangle$ with arithmetic constraints $\langle Acons \rangle$, *e.g.*, $b \vee x \geqslant 0$;
– arithmetic expressions $\langle Aexpr \rangle$ with tests on Booleans and arithmetic constraints, *e.g.*, if $b \vee x \geqslant 0$ then $x+1$ else 0.

Purely arithmetic expressions.

$\langle aAexpr \rangle ::= cst \mid \langle Avar \rangle \mid (\text{-}|\sqrt{})\langle Aexpr \rangle \mid \langle aAexpr \rangle\ (\texttt{+}|\texttt{-}|\texttt{*}|\texttt{/}|\texttt{\%})\ \langle aAexpr \rangle$

Affine arithmetic expressions, *i.e.*, of the form $\sum_i a_i x_i + b$, are canonically represented by the array of the coefficients $(a_1, \ldots, a_n, b)$. Other expressions are represented as operator trees.

Purely Boolean expressions.

$\langle bBexpr \rangle ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle Bvar \rangle \mid \neg\langle bBexpr \rangle \mid \langle bBexpr \rangle\ (\wedge \mid \vee \mid \ldots)\ \langle bBexpr \rangle$

Purely Boolean expressions are represented by a BDD ($\mathbb{B}^m \to \mathbb{B}$).

Boolean expressions with arithmetic constraints. $\langle Bexpr \rangle$ are represented by mixed (or interpreted) BDDs ($\mathbb{B}^m \times \langle Acons \rangle^* \to \mathbb{B}$).

This means that the numerical constraints $\langle Acons \rangle$ are mapped to additional (*interpreted*) Boolean variables: for example the constraint $x \geqslant 0$ will be mapped to a new variable $\widehat{b}$; this implies that its negation $x < 0$ is mapped to $\neg\widehat{b}$. See Fig. 7.5 for an example of a mixed BDD.

Operations on such expressions are simply those defined for BDDs. Mind that existential quantification cannot be applied to a numerical variable but only to a numerical constraint.

However, the problem is that these interpreted Booleans are not semantically independent, *i.e.*, they may be redundant resp. contradictory (see Fig. 7.2). Removing such decisions from such a mixed BDD is expensive, because it requires a decision procedure w.r.t. the arithmetic theory corresponding to the constraints.

Jeannet [Jea00] proposes a more light-weight solution using a "care set", *i.e.*, a Boolean formula that is the disjunction of some valid implications between the numerical constraints occurring in the expressions. This formula is computed on demand using a decision procedure. Then, certain redundant/contradictory decisions can be removed from a mixed BDD with the help of the generalized cofactor operator.

Arithmetic expressions with tests. Analogously to $\langle Bexpr \rangle$, $\langle Aexpr \rangle$ are represented by mixed MTBDDs with arithmetic expressions as terminal nodes: $\mathbb{B}^m \times \langle Acons \rangle^* \to \langle aAexpr \rangle$.

With the help of the operators apply_1 and apply_2 operations are lifted from purely arithmetic expressions to arithmetic expressions with tests. Fig. 7.3 shows as an example the addition of two arithmetic expressions with tests $f_1 + f_2$.

Fig. 7.4 shows an example of creating the Boolean expression $f > 0$ of an arithmetic expression with tests f.

Transition functions. Boolean transition functions $f^b(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})$ are represented as Boolean expressions $\langle Bexpr \rangle$. Numerical transition functions $f^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi})$ are represented as arithmetic expressions with tests $\langle Aexpr \rangle$. A guard g is a Boolean expression $\langle Bexpr \rangle$, and an action a is a purely arithmetic expression $\langle aAexpr \rangle$. Initial states $\mathcal{I}$ and error states $\mathcal{E}$ are also represented by $\langle Bexpr \rangle$.

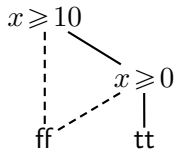


Figure 7.2: In this mixed BDD, the decision $x \geqslant 0$ is redundant (w.r.t. tt branch) and contradictory (w.r.t. ff branch).

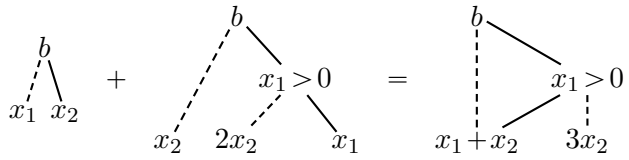


Figure 7.3: Addition of arithmetic expressions with tests $f_1 + f_2$ represented as mixed MTBDDs.

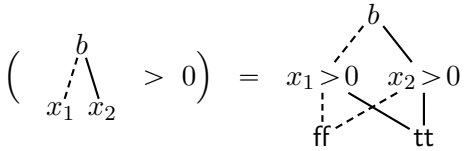


Figure 7.4: Boolean expression $f > 0$ (represented as a mixed BDD) of an arithmetic expression with tests f (represented as a mixed MTBDD).

Further operations on formulas and functions

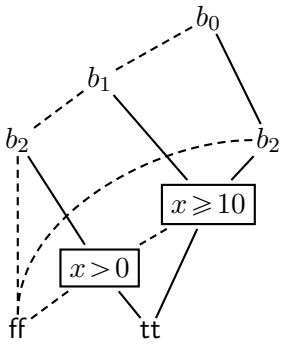
Vectorization of transition functions with factorization of their guards. In some cases we need to vectorize the actions a_j of the numerical transition functions and factorize their common guards, *i.e.*, to transform the system of transition functions

$$\begin{cases} x'_1 &= f_1^x(\boldsymbol{s}, \boldsymbol{i}) \\ \dots & \\ x'_n &= f_n^x(\boldsymbol{s}, \boldsymbol{i}) \end{cases} \quad \text{into a vector transition function } \boldsymbol{x}' = \bigvee_j \left(\boldsymbol{a}_j(\boldsymbol{x}, \boldsymbol{\xi}) \text{ if } g_j(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \right)$$

This operation is in fact the MTBDD product $\times_j f_j^x$ (§7.2.1) of the numerical transition functions.

Numerically convex formulas. Sometimes we need to decompose a Boolean formula $\varphi(\boldsymbol{b}, \mathcal{C})$ into a disjunction of numerically convex formulas: $\mathsf{decomp\_convex}(\varphi) = \bigvee_j \varphi_j^b(\boldsymbol{b}) \wedge \varphi_j^x(\mathcal{C})$ where φ_j^b are general purely Boolean formulas and φ_j^x are conjunctions of numerical constraints. This operation can be implemented efficiently by imposing a variable ordering of the BDD such that the Boolean and numerical variables are separated: for example with a variable order $b_1 < \ldots < b_m < \mathcal{C}_1 < \ldots < \mathcal{C}_p$ (Boolean variables towards the root and numerical constraints towards the leaves), the formulas φ_j^x are simply the formulas represented by the paths leading from the nodes with the lowest constraint indices to terminal node tt, and φ_j^b are the formulas guarding these nodes. See Fig. 7.5.

Partial evaluation of a formula by a polyhedron $\varphi \uparrow X$. This operation (cf. [Jea00]) simplifies a BDD or MTBDD by evaluating the numerical decisions (interpreted Booleans) over the polyhedron X along the paths of the decision diagram. Decisions that are valid or unsatisfiable are removed and their predecessor edges are redirected to the respective child nodes.



$$= \begin{cases} (b_0 \wedge b_2 \ \vee \ \neg b_0 \wedge b_1) \wedge x \geqslant 10 \ \vee \\ (b_0 \wedge b_2 \ \vee \ \neg b_0 \wedge b_1) \wedge 0 < x < 10 \ \vee \\ b_1 \wedge \neg b_2 \wedge x > 0 \end{cases}$$

Figure 7.5: Decomposition of a Boolean expression into a disjunction of numerically convex formulas: framed nodes are nodes with the lowest numerical constraint indices.

7.2.3 Abstract Domains

We abstract sets $\wp(\mathbb{B}^m \times \mathbb{R}^n)$ by an abstract domain that combines Boolean formulas, which exactly represent sets $\wp(\mathbb{B}^m)$, and a numerical abstract domain $\mathcal{N}$ abstracting $\wp(\mathbb{R}^n)$. We will consider the following two combinations (cf. §3.3.5):
– the power domain $\mathcal{N}^{(\mathbb{B}^m)}$ (also written $\mathbb{B}^m \to \mathcal{N}$), and
– the product domain $\wp(\mathbb{B}^m) \times \mathcal{N}$.

Power Domain $\mathbb{B}^m \to \mathcal{N}$

Values of this domain have the natural representation as MTBDDs ($\mathbb{B}^m \to \mathcal{N}$) (cf. [BCC+03]).

The definitions of the domain operations are summarized in Table 7.2. Operations inherited from the numerical domain are marked with the superscript x.

Our programs use mixed Boolean formulas with numerical constraints $\varphi(\boldsymbol{b}, \mathcal{C}) \in \langle Bexpr \rangle$ to specify the concrete values of initial and error states. Hence, abstraction and concretization operators allow us to convert abstract values into such formulas and vice versa:
– $\alpha : \langle Bexpr \rangle \to (\mathbb{B}^m \to \mathcal{N})$
– $\gamma : (\mathbb{B}^m \to \mathcal{N}) \to \langle Bexpr \rangle$
The γ operator converts the MTBDD into a BDD by "replacing" the numerical abstract value in the terminal nodes of the MTBDD by the Boolean formula representing the numerical abstract value: $\gamma(S) = \bigvee_{(g_j^b \to X_j) \in S} g_j^b \wedge \mathsf{to\_constraints}(X_j)$, where $\mathsf{to\_constraints} : \mathcal{N} \to \langle Bexpr \rangle$ turns a numerical abstract value into a Boolean expression: for a convex polyhedron, for instance, this is the conjunction of the constraints of its constraint representation.

The *binary operators* and predicates ($\sqsubseteq^x$, $\sqcup^x$, $\sqcap^x$, ∇^x) on numerical abstract domains are lifted to MTBDDs with the help of the apply_2 operator, *e.g.*, $S_1 \sqcup S_2 := \mathsf{apply}_2(\lambda X_1, X_2 . X_1 \sqcup^x X_2)(S_1, S_2)$. Complexity is $\mathcal{O}(n_1 n_2)$ for the BDD operations and $\mathcal{O}(t_1 t_2)$ joins of numerical abstract values (where n_i are the number of nodes and t_i are the number of terminal nodes).

Existential quantification of a numerical variable $\exists x : S$ corresponds to applying existential quantification to the numerical abstract values of the terminal nodes:

Inclusion	$S_1 \sqsubseteq S_2$	$=$	$\mathsf{apply}_2(\lambda(X_1, X_2).X_1 \sqsubseteq^x X_2)(S_1, S_2)$
Abstraction	$\alpha(\varphi(\boldsymbol{b}, \mathcal{C}))$	$=$	$\top \sqcap^g \varphi(\boldsymbol{b}, \mathcal{C})$
Concretization	$\gamma(S)$		see text
Emptyness	$(S \Leftrightarrow \bot)$	$=$	$(S \Leftrightarrow \lambda \boldsymbol{b}.\bot^x)$
Union	$S_1 \sqcup S_2$	$=$	$\mathsf{apply}_2(\lambda(X_1, X_2).X_1 \sqcup^x X_2)(S_1, S_2)$
Intersection	$S_1 \sqcap S_2$	$=$	$\mathsf{apply}_2(\lambda(X_1, X_2).X_1 \sqcap^x X_2)(S_1, S_2)$
Guard intersection	$S \sqcap^g g$		see text
Transformation	$\begin{pmatrix} \boldsymbol{f}^b \\ \boldsymbol{f}^x \end{pmatrix}(S), \begin{pmatrix} \boldsymbol{f}^b \\ \boldsymbol{f}^x \end{pmatrix}^{-1}(S)$		see text
Projection	$\exists x : S$	$=$	$\mathsf{apply}_1(\lambda X.\exists x : X)(S)$
	$\exists b : S$	$=$	$\mathsf{apply}_2(\lambda(X_1, X_2).X_1 \sqcup^x X_2)(S[b \to \mathsf{tt}], S[b \to \mathsf{ff}])$
Widening	$S_1 \nabla S_2$	$=$	$\mathsf{apply}_2(\lambda(X_1, X_2).X_1 \nabla^x X_2)(S_1, S_2)$

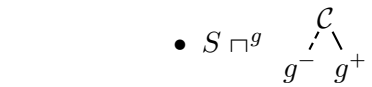
Table 7.2: Operations of the logico-numerical power domain $\mathbb{B}^m \to \mathcal{N}$.

$\mathsf{apply}_1(\lambda X.\exists x : X)(S)$. Mind that we can existentially quantify a numerical variable in a mixed Boolean formula (approximately w.r.t. an abstract domain) by computing $\gamma(\exists x.\alpha(\varphi(\boldsymbol{b}, \mathcal{C})))$.

Existential quantification of a Boolean variable is almost identical to existential quantification in BDDs, except that we have to use $\sqcup^x$ (instead of $\vee$) to combine the terminal nodes.

Intersection with a guard (a Boolean formula with numerical constraints) $S \sqcap^g g(\boldsymbol{b}, \mathcal{C})$ is implemented by the following recursion:

– Base cases:
  - $S \sqcap^g \mathsf{tt} = S$
  - $S \sqcap^g \mathsf{ff} = \bot$
  - No constraint in g: $S \sqcap^g g(\boldsymbol{b}) = S \sqcap \mathsf{ite}(g, \top, \bot)$
  - Single constraint in g: $S \sqcap^g \mathcal{C} = \mathsf{apply}_1(\lambda S.S \sqcap^x \alpha(\mathcal{C}))(S)$

– Recursion:
  -  $= S^- \sqcap^g g^- \;\; S^+ \sqcap^g g^+$
  - $S \sqcap^g$  $= (S \sqcap^g \neg\mathcal{C}) \sqcap^g g^- \sqcup (S \sqcap^g \mathcal{C}) \sqcap^g g^+$

Transformations (image and pre-image computation) work in a similar manner as the guard intersection.

The *widening* operator of the numerical abstract domain is applied to the terminal nodes using apply_1. The Boolean component does not actually need to be widened because the corresponding lattice has finite height $\mathcal{O}(2^m)$, thus we simply take the disjunction of the formulas. Moreover, it is difficult to define a reasonable widening operator: as already mentioned, widening supposes that there is some regularity in the operations as, *e.g.*, translations, but there is no such regularity in Boolean operations. Moreover, Boolean extrapolation operators on BDDs are highly dependent on the variable ordering. Mauborgne [Mau98] proposes an operator that over-approximates a BDD

Inclusion	$(B_1, X_1) \sqsubseteq (B_2, X_2)$	$:=$	$B_1 \Rightarrow B_2 \wedge X_1 \sqsubseteq^x X_2$
Abstraction	$\alpha(\varphi(\boldsymbol{b}, \mathcal{C}))$	$:=$	$\mathsf{cvx}(\top \sqcap^g \varphi(\boldsymbol{b}, \mathcal{C}))$
Concretization	$\gamma(B, X)$	$:=$	$B \wedge \mathsf{to\_constraints}(X)$
Canonicalization	$\mathsf{can}(B, X)$	$:=$	$\begin{cases} \bot & \text{if } B \Leftrightarrow \mathsf{ff} \vee X = \bot^x \\ (B, \mathsf{can}^x(X)) & \text{else} \end{cases}$
Emptiness	$(B, X) \Leftrightarrow \bot$	$:=$	$(B, X) \Leftrightarrow \bot$
Union	$(B_1, X_1) \sqcup (B_2, X_2)$	$:=$	$(B_1 \vee B_2, X_1 \sqcup^x X_2)$
Intersection	$(B_1, X_1) \sqcap (B_2, X_2)$	$:=$	$\mathsf{can}(B_1 \wedge B_2, X_1 \sqcap^x X_2)$
Guard intersection	$S \sqcap^g g$	$:=$	$\mathsf{cvx}(S \sqcap^g g)$
Transformation	$\begin{pmatrix} \boldsymbol{f}^b \\ \boldsymbol{f}^x \end{pmatrix} \begin{pmatrix} B \\ X \end{pmatrix}$	$:=$	$\begin{pmatrix} (\boldsymbol{f}^b \uparrow X) \uparrow B \\ \bigsqcup_j [\![\boldsymbol{a}_j]\!]^\sharp (X \sqcap (g_j \uparrow B)) \end{pmatrix}$
	$\begin{pmatrix} \boldsymbol{f}^b \\ \boldsymbol{f}^x \end{pmatrix}^{-1} \begin{pmatrix} B \\ X \end{pmatrix}$	$:=$	$\begin{pmatrix} B[\boldsymbol{b} \leftarrow (\boldsymbol{f}^b \uparrow X)] \\ \bigsqcup_j [\![\boldsymbol{a}_j]\!]^{\sharp -1} (X) \sqcap (g_j \uparrow B) \end{pmatrix}$
Projection	$\exists b, \exists x : (B, X)$	$:=$	$(\exists b : B, \exists x : X)$
Widening	$(B_1, X_1) \nabla (B_2, X_2)$	$:=$	$(B_1 \vee B_2, X_1 \nabla^x X_2)$

Table 7.3: Operations of the logico-numerical product domain $\wp(\mathbb{B}^m) \times \mathcal{N}$.

by another Bdd that has at most a given number of nodes.

Product domain $\wp(\mathbb{B}^m) \times \mathcal{N}$

This domain approximates a set of states coarsely by a conjunction of a Boolean formula and a single abstract value. For example, Jeannet [Jea00] considers the product domain $\wp(\mathbb{B}^m) \times Pol(\mathbb{R}^n)$. In such a domain the formula $(b \wedge x \leqslant 2) \vee (\neg b \wedge x \leqslant 4)$ is abstracted by $\mathsf{tt} \wedge x \leqslant 4$. This domain is not implemented in BddApron, but its operations can be easily derived from those of the power domain.

The definitions of the domain operations are summarized in Table 7.3. The operations for union and intersection are applied component-wise. Inclusion must hold for both components. The value $\bot$ requires a canonicalization.

Guard intersection by a Boolean formula with numerical constraints can be implemented by the corresponding operator of the power domain followed by a convexification of the result (cvx): All non-$\bot$ terminal nodes are joined ($\sqcup^x$) resulting in an MtBdd of the form $\mathsf{ite}(\varphi(\boldsymbol{b}), X, \bot^x)$, which is then transformed into the pair (B, X) with $B = \mathsf{ite}(\varphi(\boldsymbol{b}), \mathsf{tt}, \mathsf{ff})$ in constant time.

Transformations can be composed from the previously defined operators using guard intersection and partial evaluation: The postcondition operators, for example, are computed as follows: the numerical constraints in the Boolean transition function are evaluated over the numerical abstract value X and its Boolean variables are evaluated over the Boolean states B; the numerical image is obtained by the convex union of applying the numerical actions $\boldsymbol{a}_j$ associated with each case of the if-then-else to the numerical abstract value X constrained by the associated guard g_j evaluated over the Boolean states B.

We will mainly use the product domain. Since this abstract domain provides only

relatively coarse approximations, we follow the approach of building a power domain by partitioning the program as explained in the next section §7.3.

7.3 State Space Partitioning

Static analysis is usually performed over the CFG of a program. This allows us to assign an abstract value from an abstract domain A to each location $\ell \in L$ of the graph, inducing the power domain A^L. However, the basic CFG of a data-flow program, *e.g.*, a LUSTRE program, consists of a single location.

In order to conduct a classical analysis of such programs using numerical abstract domains, it is necessary to explicitly unfold the Boolean control structure by *enumerating* the Boolean state space and encoding the Boolean states in locations of a CFG. In addition, by interpreting Boolean input variables as non-deterministic choices, one obtains a CFG that has only numerical transition relations (Fig. 7.6b).

The problem is that this enumeration becomes intractable with larger programs because the number of locations and arcs grows exponentially with the number of Boolean state and input variables respectively (state space explosion problem).

Since we perform a logico-numerical analysis, such an enumeration is not required. Nevertheless, when using the logico-numerical product domain, we would like to benefit from the power domain induced by a CFG in order to enable more precise analyses.

Considering a partition of the state space of a system allows us to generate a CFG (§7.3.1) by associating to each partition element (equivalence class) to a location. *Defining a partition* (§7.3.2) is usually based on heuristics. §7.3.3 describes the *partitioning process* in practice.

7.3.1 Control Flow Graphs Induced by State Space Partitioning

We extend the definition of a control flow graph (Def. 2.3) by labeling each location ℓ with a *location definition* (or location invariant) φ_ℓ. We view φ_ℓ alternatively as the predicate $\varphi_\ell(\boldsymbol{s})$ or the set $\{\varphi_\ell(\boldsymbol{s}) \mid \boldsymbol{s} \in \Sigma\}$.

Definition 7.4 (Partition-induced CFG) *A partition-induced CFG* $\langle \Sigma, \Upsilon, L, \rightsquigarrow, \Sigma^0 \rangle$ *is a directed graph where*
- Σ *and* Υ *are the state and input spaces,*
- L *is the set of locations; each location* $\ell \in L$ *is labeled with its location definition* $\varphi_\ell(\boldsymbol{s})$ *such that the sets* $\varphi_\ell(\boldsymbol{s})$ *form a partition of* Σ, i.e., $\varphi_\ell \cap \varphi_{\ell'} = \varnothing$ *for* $\ell \neq \ell'$ *and* $\bigcup_\ell \varphi_\ell = \Sigma$.
- $\rightsquigarrow \subseteq L \times \mathcal{R} \times L$ *defines arcs between the locations. The arcs are labeled with a transition relation* $R \in \mathcal{R} = (\Sigma \times \Upsilon \times \Sigma)$.
- $\Sigma^0 : L \to \Sigma$ *defines, for each location* ℓ*, the set of initial states, such that* $\Sigma^0(\ell) \subseteq \varphi_\ell(\boldsymbol{s})$.

We take into account this location definition in the semantics of Def. 2.3:

Definition 7.5 (Semantics) *An* execution *of a partitioned CFG is a sequence*

$$(\ell_0, \boldsymbol{s}_0) \xrightarrow{\boldsymbol{i}_0} (\ell_1, \boldsymbol{s}_1) \xrightarrow{\boldsymbol{i}_1} \dots (\ell_k, \boldsymbol{s}_k) \xrightarrow{\boldsymbol{i}_k} \dots$$

such that for any $k \geqslant 0 : \exists (\ell_k, R, \ell_{k+1}) \in \rightsquigarrow : R(\boldsymbol{s}_k, \boldsymbol{i}_k, \boldsymbol{s}_{k+1}) \wedge \varphi_{\ell_{k+1}}(\boldsymbol{s}_{k+1})$ *where* $\ell \in L$ *and* $\boldsymbol{s} \in \Sigma$.

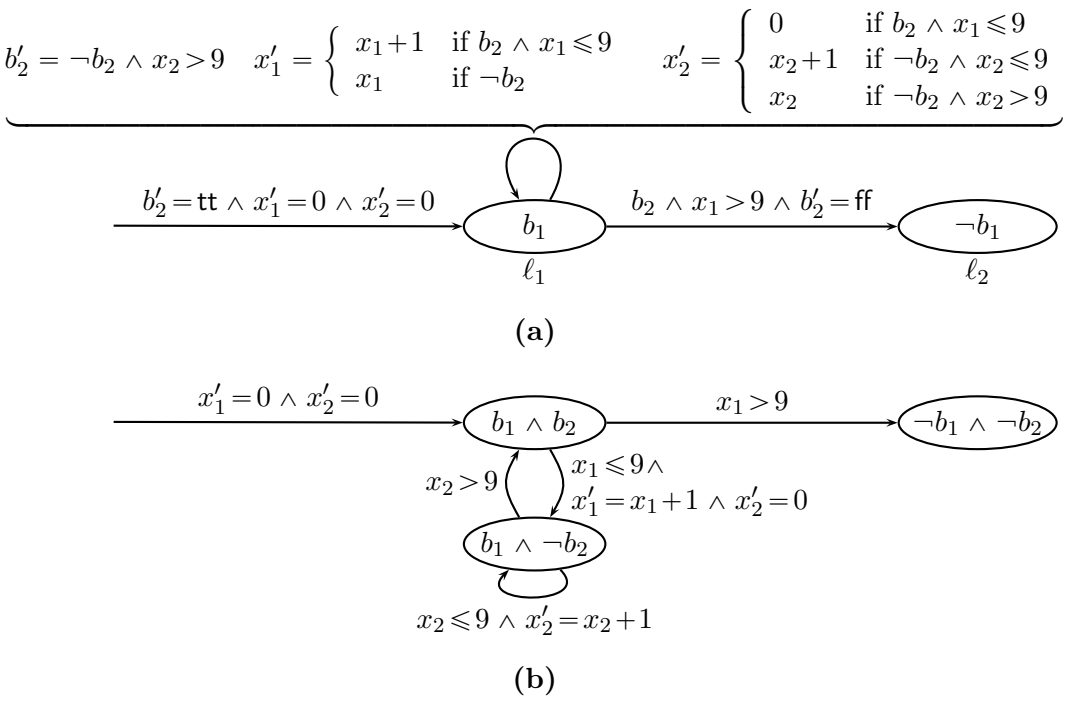


Figure 7.6: Two CFGs of the logico-numerical program in Ex. 7.4 obtained by state space partitioning.

Example 7.4 (Partition-induced CFG) *Let us consider the following logico-numerical program:*

$$\begin{cases} \mathcal{I}(b_1, b_2, x_1, x_2) = (b_1 \wedge b_2 \wedge x_1 = 0 \wedge x_2 = 0) \\ b_1' = b_1 \wedge \neg b_2 \ \vee \ b_1 \wedge b_2 \wedge x_1 \leqslant 9 \\ b_2' = b_1 \wedge \neg b_2 \wedge x_2 > 9 \\ x_1' = \begin{cases} x_1+1 & \textit{if } b_1 \wedge b_2 \wedge x_1 \leqslant 9 \\ x_1 & \textit{else} \end{cases} \\ x_2' = \begin{cases} x_2+1 & \textit{if } b_1 \wedge \neg b_2 \wedge x_2 \leqslant 9 \\ 0 & \textit{if } b_1 \wedge b_2 \wedge x_2 \leqslant 9 \\ x_2 & \textit{else} \end{cases} \end{cases}$$

Fig. 7.6a shows a CFG of this program induced by the partition $\{b_1, \neg b_1\}$*. Fig. 7.6b depicts the CFG obtained by the enumeration of the (reachable) Boolean state space,* i.e.*, induced by the partition* $\{b_1 \wedge b_2, b_1 \wedge \neg b_2, \neg b_1 \wedge \neg b_2\}$ *(*$\neg b_1 \wedge b_2$ *is trivially not reachable, see §7.3.3).*

A partition-induced CFG can be transformed into an equivalent discrete transition system (Def. 2.1.1) $\langle \Sigma, \Upsilon, R', \mathcal{I} \rangle$ with $R'(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}') = \bigcup_{(\ell, R, \ell') \in \rightsquigarrow} R(\boldsymbol{s}, \boldsymbol{i}, \boldsymbol{s}') \cap \varphi_{\ell'}(\boldsymbol{s}')$ and $\mathcal{I} = \bigcup_{\ell \in L} \Sigma^0(\ell)$.

Analysis. The use of a CFG implies the power abstract domain $(L \to A)$ where the concrete states S are connected to their abstract counterparts $S^\sharp$ by the Galois connection:

$$S^\sharp = \alpha(S) = \lambda \ell \,.\, \alpha(S \cap \varphi_\ell) \qquad\qquad S = \gamma(S^\sharp) = \bigcup_{\ell \in L} \gamma(S^\sharp_\ell)$$

$\mathsf{reach}(S^{\sharp 0})$ is the least fixed point of

$$S^\sharp = S^{\sharp 0} \sqcup \lambda \ell \,.\, \bigsqcup_{\ell' \in L} \Big(\mathsf{post}^\sharp(S^\sharp_{\ell'}) \sqcap \varphi_\ell \Big)$$

where $S^\sharp, S^{\sharp 0} \in (L \to A)$.

Example 7.5 (Analysis) *We analyze the CFG in Fig. 7.6a (Ex. 7.4) using the logico-numerical power domain (with intervals) $\mathbb{B}^m \to \mathit{Int}(\mathbb{R}^n)$ with Kleene iteration and widening ($N=0$): We start with the initial state:*
$S_0 = \Big(\ell_1 \to (b_1 \wedge b_2 \to x_1 \in [0,0] \wedge x_2 \in [0,0]), \ell_2 \to \bot\Big)$
The widening sequence converges with
$S'_5 = \Big(\ell_1 \to \left\{ \begin{array}{l} (b_1 \wedge b_2 \to x_1 \in [0,\infty[\wedge x_2 \in [0,\infty[) \ \vee \\ (b_1 \wedge \neg b_2 \to x_1 \in [1,\infty[\wedge x_2 \in [0,\infty[) \end{array} \right. , \ell_2 \to (b_1 \wedge \neg b_2 \to x_1 \in [10,\infty[\wedge x_2 \in [0,\infty[)\Big)$
We get the final result after the descending iterations:
$S''_3 = \Big(\ell_1 \to \left\{ \begin{array}{l} (b_1 \wedge b_2 \to x_1 \in [0,\infty[\wedge x_2 \in [0,10]) \ \vee \\ (b_1 \wedge \neg b_2 \to x_1 \in [1,\infty[\wedge x_2 \in [0,10])) \end{array} \right. , \ell_2 \to (\neg b_1 \wedge \neg b_2 \to x_1 \in [10,\infty[\wedge x_2 \in [0,10])\Big)$
The analysis using the logico-numerical product domain $\wp(\mathbb{B}^m) \times \mathit{Int}(\mathbb{R}^n)$ yields the weaker result: $\Big(\ell_1 \to (b_1 \wedge x_1 \in [0,\infty[\wedge x_2 \in [0,\infty[), \ell_2 \to (\neg b_1 \wedge \neg b_2 \wedge x_1 \in [10,\infty[\wedge x_2 \in [0,\infty[)\Big)$

7.3.2 Defining Partitions

Techniques for defining partitions were developed in the context of model checking (cf. §3.1) for minimizing and abstracting systems.

Our approach for generating a partition follows the idea of predicate abstraction [GS97, FQ02], which considers the truth values of a finite set of predicates Ψ usually obtained by heuristics. Hence, the generated partition consists of $\mathcal{O}(2^{|\Psi|})$ equivalence classes, *i.e.*, the possible combinations of conjunctions of the predicates and their negations.

The basic partition we use for verification [Jea00] divides the system into initial and error states and those that are neither initial nor error states. The set of predicates is $\Psi = \bigcup_j \{\mathcal{I}_j\} \cup \bigcup_j \{\mathcal{E}_j\}$ where the initial states $\mathcal{I}$ and error states $\mathcal{E}$ are decomposed into (disjoint) convex formulas $\mathcal{I}_j$ and $\mathcal{E}_j$ respectively in order to represent them exactly – taking simply $\alpha(\mathcal{I})$ and $\alpha(\mathcal{E})$ would lead to a coarse approximation if $\mathcal{I}$ and $\mathcal{E}$ have non-convex numerical parts. Fig. 7.7 shows the schema of such a CFG.

Then, we can further partition the location $\neg\mathcal{I} \wedge \neg\mathcal{E}$ by another set of predicates obtained by a partitioning heuristics. We will propose such heuristics for discrete systems in §8.3 and then for hybrid systems in §13.2.

It can be considered to alternate analysis and partitioning so as to iteratively *refine the partition* in case the analysis was inconclusive (see §7.4). Such techniques are orthogonal to the one-shot partitioning/analysis approach considered in this thesis.

7.3.3 Partitioning Process

In this section we describe how we implement state space partition of a logico-numerical program.

Boolean analysis. Before starting the actual partitioning process, we perform a cheap reachability analysis of the Boolean abstraction of the system in order to restrict the state space Σ that is going to be partitioned. This analysis ignores the numerical

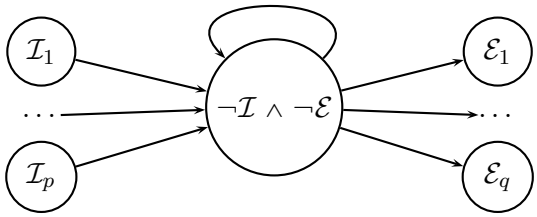


Figure 7.7: Partitioning by initial and error states.

transition functions, but it takes into account the numerical constraints represented by interpreted Booleans in the BDDs:

$$\begin{array}{rcl}
\mathsf{post}^b(\varphi(\boldsymbol{b},\mathcal{C})) & = & \exists \boldsymbol{b},\boldsymbol{\beta},\mathcal{C} : (\boldsymbol{b}' \Leftrightarrow \boldsymbol{f}^b(\boldsymbol{b},\mathcal{C})) \wedge \varphi(\boldsymbol{b},\mathcal{C}) \wedge \mathcal{A}(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C}) \\
\mathsf{pre}^b(\varphi(\boldsymbol{b}',\mathcal{C})) & = & \exists \boldsymbol{b}',\boldsymbol{\beta},\mathcal{C} : (\boldsymbol{b}' \Leftrightarrow \boldsymbol{f}^b(\boldsymbol{b},\mathcal{C})) \wedge \varphi(\boldsymbol{b}',\mathcal{C}) \wedge \mathcal{A}(\boldsymbol{b},\boldsymbol{\beta},\mathcal{C}) \\
\mathsf{reach}^b(\mathcal{I}) & = & lfp\lambda\varphi.\mathcal{I} \vee \mathsf{post}^b(\varphi) \\
\mathsf{co\text{-}reach}^b(\mathcal{E}) & = & lfp\lambda\varphi.\mathcal{E} \vee \mathsf{pre}^b(\varphi)
\end{array}$$

The reachability computation is guaranteed to converge in a finite number of steps because the state space is finite.

Example 7.6 (Boolean analysis) *We analyze the following program:*

$$\left\{\begin{array}{l}
\mathcal{I}(b_1,b_2,x) = (\neg b_1 \wedge b_2 \wedge x = 0) \\
b_1' = \neg b_1 \wedge x \geqslant 0 \\
b_2' = b_2 \wedge x < 0 \\
x' = \ldots
\end{array}\right.$$

After two iterations we reach the fixed point $\neg b_1 \vee \neg b_2$. Hence, the system has at most three reachable Boolean states.

Incremental partitioning. Partitioning is done by incrementally dividing the locations of the CFG. For all predicates $\psi \in \Psi$, each location ℓ is tried to be split by ψ. A location can be divided only if ψ splits the location definition φ_ℓ into two formulas that are neither φ_ℓ itself nor ff, *i.e.*, $(\varphi_\ell \wedge \psi \Rightarrow \varphi_\ell) \wedge (\varphi_\ell \wedge \neg\psi \Rightarrow \varphi_\ell)$. The splitting algorithm is listed in Fig. 7.8 and illustrated in Fig. 7.9.

Mind that this splitting algorithm results in a fully connected graph. However, many transitions are not feasible. Hence, the CFG is simplified during incremental partitioning:

Simplification of transition functions by origin location. We simplify the transition function f associated to an arc $\ell \rightsquigarrow \ell'$ by partial evaluation: $\boldsymbol{f} \uparrow \varphi_\ell$.

Example 7.7 (Simplification by origin location) *We simplify the following transition function associated with an outgoing arc of location ℓ with $\varphi_\ell = \neg b_1$:*

$$\left(\begin{array}{l}
x' = \left\{\begin{array}{ll} x+1 & \text{if } b_1 \wedge x \leqslant 9 \\ 0 & \text{if } \neg b_1 \vee x > 9 \end{array}\right. \\
b_1' = b_1 \wedge \neg b_2 \wedge x > 5 \\
b_2' = \neg b_2
\end{array}\right) \uparrow \neg b_1 \Leftrightarrow \left(\begin{array}{l} x' = 0 \\ b_1' = false \\ b_2' = \neg b_2 \end{array}\right)$$

split_location(CFG,ℓ,ψ):
  if $(\varphi_\ell \wedge \psi \Rightarrow \varphi_\ell) \wedge (\varphi_\ell \wedge \neg\psi \Rightarrow \varphi_\ell)$ then
  begin
    add locations ℓ_1 with $\varphi_{\ell_1} = \varphi_\ell \wedge \psi$ and ℓ_2 with $\varphi_{\ell_2} = \varphi_\ell \wedge \neg\psi$
    duplicate arcs:
      for all incoming (non-loop) arcs (ℓ_p, R, ℓ): add (ℓ_p, R, ℓ_1) and (ℓ_p, R, ℓ_2)
      for all outcoming (non-loop) arcs (ℓ, R, ℓ_s): add (ℓ_1, R, ℓ_s) and (ℓ_2, R, ℓ_s)
      for all loop arcs (ℓ, R, ℓ): add (ℓ_1, R, ℓ_1), (ℓ_2, R, ℓ_2), (ℓ_1, R, ℓ_2) and (ℓ_2, R, ℓ_1)
    remove ℓ
  end
  return CFG

Figure 7.8: Incremental partitioning: algorithm for splitting a location ℓ by a predicate ψ.

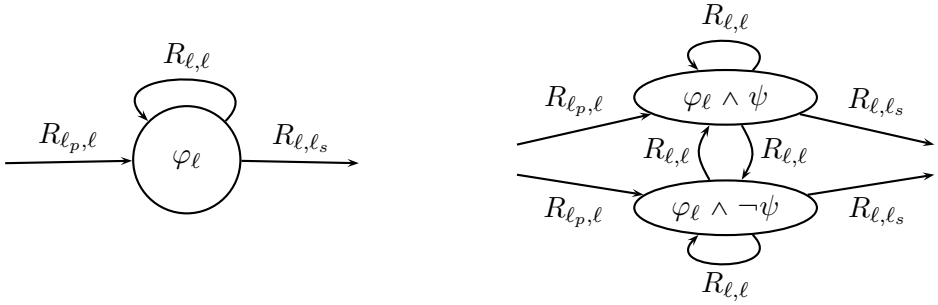


Figure 7.9: Incremental partitioning: splitting a location (a) by a predicate ψ into two locations (b).

Removal of infeasible arcs. A transition $\ell \rightsquigarrow \ell'$ is *feasible* iff $\exists \boldsymbol{x}, \boldsymbol{i} : \varphi_\ell(\boldsymbol{s}) \wedge \boldsymbol{s}' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{i}) \wedge \varphi_{\ell'}(\boldsymbol{s}')$. If an arc is proved to be *infeasible* it is removed. This can be done by checking the satisfiability of the above formula using

– an SMT solver,
– the abstract post-condition $\mathsf{post}^\sharp$, or
– the Boolean post-condition post^b.

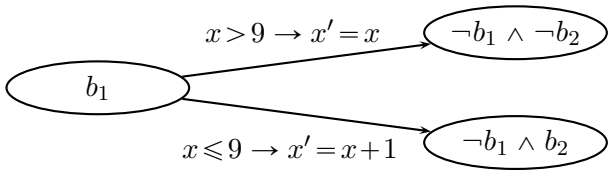
The latter method removes the least number of arcs, but it is also the least expensive. Observe that the infeasible arcs have been removed in the CFGs in Fig. 7.6.

Arc assertions. An arc assertion is the condition that must be satisfied in order to reach the destination location definition $\varphi_{\ell'}$ of a transition. We can over-approximate this condition by computing the Boolean pre-image of the destination location by an arc: $g(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) = \exists \boldsymbol{b}' : (\boldsymbol{b}' \Leftrightarrow \boldsymbol{f}^b(\boldsymbol{b}, \mathcal{C})) \wedge \varphi_{\ell'}(\boldsymbol{b}', \mathcal{C}) \wedge \mathcal{A}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})$.

The arc assertion can also be viewed as a factorization of the guards in the transition function: $g \rightarrow (\boldsymbol{s}' = \boldsymbol{f} \uparrow g)$. Fig. 7.10 gives an example for the computation of arc assertions.

$$R = \begin{cases} b_1' = \neg b_1 \\ b_2' = x \leqslant 9 \\ x' = \begin{cases} x+1 & \text{if } x \leqslant 9 \\ x & \text{if } x > 9 \end{cases} \end{cases}$$

(a)



(b)

Figure 7.10: Arc assertions and refinement of the location definition by the destination location: (a) original transition relation associated to the two outgoing arcs of location b_1 in (b); (b) refinement of the transitions by arc assertions.

7.4 Alternative Approaches

In this section we discuss related work w.r.t. alternative logico-numerical abstract domains, partition refinement techniques and alternative (not abstract-interpretation-based) approaches to the analysis of logico-numerical programs.

Logico-numerical abstract domains

Mixed BDD domain. Mauras [Mau96] proposes mixed BDDs (Boolean formulas with numerical constraints) as a logico-numerical abstract domain. The problem is that the representation is *exact*, *i.e.* there is no abstraction. Moreover, mixed BDDs are not well-suited as abstract domain representation, because they have no canonical form and their manipulation is less efficient than the convex representations used by numerical abstract domains. At last, it is difficult to define a widening operator.

Presburger arithmetic and BDDs. Another approach originates from model checking logico-numerical systems. Bultan et al. [BGP97, BGL00, YKTB01] propose a combination of BDDs and Presburger arithmetic as implemented by the library OMEGA. OMEGA uses disjunctions of convex polyhedra and divisibility constraints to represent Presburger formulas. However, in order to contain the explosion in number of polyhedra they bound the number of disjunctions and they force the convex hull when the bound is exceeded.

Partition Refinement

The goal of partition refinement is to improve precision in case the analysis result is inconclusive.

Avoiding convex unions. Apart from widening, the main reason for losing precision is the convex approximation in the abstract domain.

Identifying in the CFG the convex unions that lose most precision requires an efficiently computable quantitative measure for precision loss. Jeannet [Jea00], for instance, suggests to count the number of equalities lost in the constraint representation of a polyhedron.

Trace partitioning [HT98, RM07] avoids convex unions by distinguishing program paths according to the history of program states and control flow.

Property-based refinement. The prevalent partition refinement methods try to identify the locations where the decisive information to prove the property is lost by the abstraction.

The classical method in model checking for such a property-based, systematic partition refinement uses spurious counterexamples, *i.e.*, behavior that is only present in the abstraction but not in the concrete program. The abstraction is refined by eliminating this behavior.

The well-known CEGAR-method ([CGJ+00], cf. §3.1) is based on computing concrete counterexamples traces corresponding to a spurious abstract counterexample in order to determine the first location where the abstract trace starts to exhibit spurious behavior. Then this location is split such that the spurious counterexample is eliminated. Since finding the coarsest refinement is an NP-hard problem (reduction to partition-into-cliques problem) [CGJ+00], they propose a heuristic algorithm for deriving an appropriate splitting predicate.

In the abstract interpretation context, such a refinement has already been proposed by Wong-Toi [WT94]. However, his method makes use of an under-approximated backward analysis – which is difficult to compute – in order to check whether the a counterexample is spurious or not.

Jeannet [JHR99, Jea00, Jea03] proposes heuristic methods for property-based partition refinement (*dynamic partitioning*) implemented in the tool Nbac. Again, the idea is to split locations based on abstract counterexamples such that potentially "dangerous" paths from initial to error states are cut: the locations privileged are those which allow strongly connected components to be cut in the graph. Candidates for splitting predicates are the arc assertions. Since the only numerical constraints considered for refinement are those occurring in the program source, there is a "finest" partition with a finite number of locations. Thus, in contrast to CEGAR, these heuristics guarantee the termination of the partitioning process.

SMT-based approaches

Alternative approaches for verifying properties about discrete logico-numerical data-flow programs rely on bounded model-checking (cf. [CBRZ01]) or k-induction (cf. [SSS00]) techniques, which both exploit the efficiency of modern SMT solvers (see [BHvMW09]).

Hagen and Tinelli [HT08] describe the application of these two approaches to the verification of Lustre programs. The transition relation $R(n) = (\boldsymbol{s}_n = \boldsymbol{f}(\boldsymbol{s}_{n-1}, \boldsymbol{i}_n))$ and the observers $\mathcal{A}(n) = \mathcal{A}(\boldsymbol{s}_n, \boldsymbol{i}_n)$ and $\mathcal{G}(n) = \mathcal{G}(\boldsymbol{s}_n)$ are unrolled k times and then a k-induction-based proof is tried using an SMT-solver:

– Base case (*i.e.*, bounded model checking up to k):

$$\bigwedge_{n=0\ldots k} R(n) \wedge \mathcal{A}(n) \quad \Longrightarrow \quad \bigwedge_{n=0\ldots k} \mathcal{G}(n)$$

– Induction step $m \to (m+1)$:

$$\bigwedge_{n=m\ldots m+k+1} R(n) \wedge \mathcal{A}(n) \wedge \bigwedge_{n=m\ldots m+k} \mathcal{G}(n) \quad \Longrightarrow \quad \mathcal{G}(m+k+1)$$

If the proof fails, it is retried with increasing values of k.

This technique has been implemented in the tool KIND[2] and has proved powerful [HT08, Hag08, DHKR11].

Like abstract interpretation, k-induction provides unbounded verification. However, there are several differences: k-induction does not explicitly compute the reachable state space, which on the one hand can be an advantage in terms of performance, but on the other hand, the method cannot be used for computing invariants, but only for verifying them. Moreover, there is in general no guarantee for termination. Yet, in combination with bounded model checking, one can falsify properties and obtain counterexamples.

[2] `http://clc.cs.uiowa.edu/Kind/`

Chapter 8

Logico-Numerical Abstract Acceleration

In this chapter we extend abstract acceleration from purely numerical programs to logico-numerical ones.

The classical approach to applying the abstract acceleration concepts of Gonnord et al. [GH06, Gon07] to logico-numerical programs relies on the enumeration of the Boolean state space (see §8.1). However, this technique suffers from the state space explosion. In contrast, our approach alleviates this problem with the help of an analysis in a *logico-numerical abstract domain* (§7.2.3) and *state space partitioning* (§7.3).

For this purpose we present an efficient method for (i) building an appropriate CFG without resorting to Boolean state space enumeration, and (ii) analyzing it using abstract acceleration. Our methods allow us to treat these two problems independently of each other. Our contributions can be summarized as follows (Fig. 8.1):

1. We propose methods for *accelerating self-loops* in the CFG of *logico-numerical* data-flow programs (§8.2).
2. We define *Boolean partitioning heuristics*, (§8.3) which favor the applicability of abstract acceleration and enable a reasonably precise reachability analysis.
3. We provide *experimental results* (§8.4) on the use of abstract acceleration enhancing the analysis of logico-numerical programs.

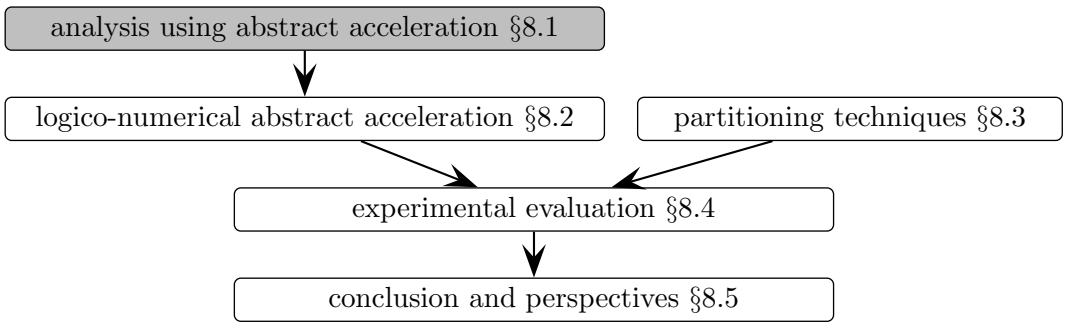


Figure 8.1: Chapter organization

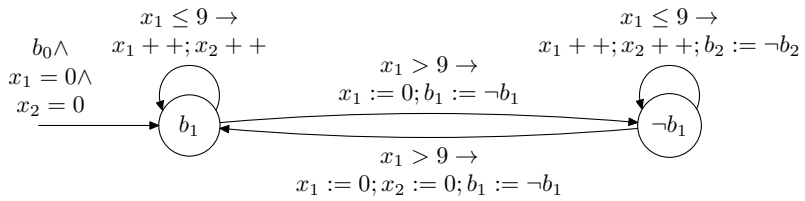


Figure 8.2: Self-loop ready to be accelerated (location b_1). Acceleration not applicable (location $\neg b_1$).

8.1 Analysis Using Abstract Acceleration

In this section, we describe the classical approach to applying abstract acceleration to the analysis of logico-numerical programs, as followed by the tool ASPIC [Gon], for which we propose major enhancements.

Example 8.1 *We will try to infer invariants on the following running example:*

$$\mathcal{I}(\boldsymbol{b},\boldsymbol{x}) = \neg b_0 \wedge \neg b_1 \wedge x_0 = 0 \wedge x_1 = 0 \wedge x_2 = 0$$

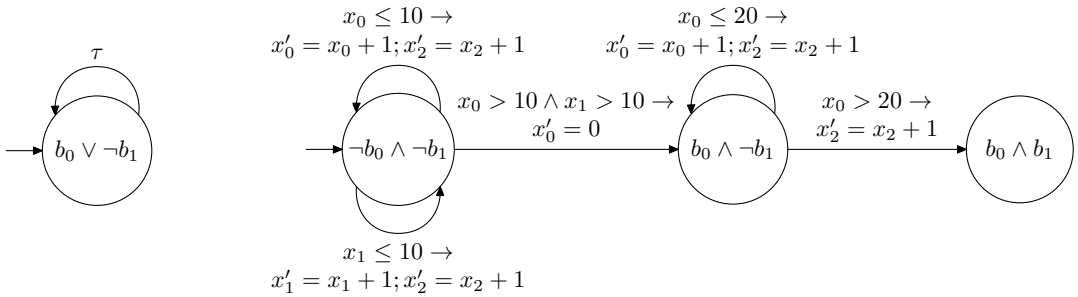
$$\mathtt{tt} \rightarrow \begin{cases} b_0' &=& b_0 \ \vee \ (\neg b_0 \wedge x_0 > 10 \wedge x_1 > 10) \\ b_1' &=& b_1 \ \vee \ (\neg b_1 \wedge x_0 > 20) \\ x_0' &=& \begin{cases} x_0 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \leqslant 10 \wedge \beta) \ \vee \ (b_0 \wedge \neg b_1 \wedge x_0 \leqslant 20) \\ 0 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ x_0 & \text{otherwise} \end{cases} \\ x_1' &=& \begin{cases} x_1 + 1 & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \leqslant 10 \wedge \neg\beta \\ x_1 & \text{otherwise} \end{cases} \\ x_2' &=& \begin{cases} x_2 + 1 & \text{if } (\neg b_0 \wedge \neg b_1 \wedge (x_0 \leqslant 10 \wedge \beta \vee x_1 \leqslant 10 \wedge \neg\beta)) \ \vee \ (b_0 \wedge \neg b_1) \\ x_2 & \text{otherwise} \end{cases} \end{cases}$$

Numerical acceleration can be applied to self-loops where the numerical state evolves while the Boolean state does not, *i.e.*, the Boolean part of the transition function is the identity (see Fig. 8.2 for an example and a counterexample):

Definition 8.1 (Accelerable logico-numerical transition) *A transition τ is accelerable if it has the form $g^b(\boldsymbol{b},\boldsymbol{\beta}) \wedge g^x(\mathcal{C}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix}$, where $g^x(\mathcal{C}) \rightarrow \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi})$ is accelerable.*

First, Boolean states are enumerated using the techniques described in §7.3, which trivially induce a CFG in which the Boolean part of the transition functions of the self-loops is the identity. However, the guards of the loops might still be non-convex. Transforming the guard into a minimal disjunctive normal form (DNF) and splitting the transition into several transitions, one for each disjunct, yields a CFG with self-loops accelerable according to Def. 8.1. A single self-loop, like in location $b_0 \wedge \neg b_1$ in Fig. 8.3b can now be "flattened" into a transitive closure transition as done in Fig. 4.1b.

In case of multiple self-loops, like in location $\neg b_0 \wedge \neg b_1$ in Fig. 8.3b, a simple "flattening" of the loop is not possible: For the fixed point computation, we must take into account all sequences of self-loop transitions in this location. For the two accelerable loops we have to compute: $\tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \tau_1^{\otimes} \circ \tau_2^{\otimes} \circ$



(a) Initial CFG **(b)** CFG after Boolean enumeration and removal of the Boolean inputs. (Identity transition functions are implicit.)

Figure 8.3: Transformation of the program of Example 8.1. τ is the global transition. The guards are already convex in the obtained CFG.

$\tau_1^{\otimes}(X) \sqcup \tau_2^{\otimes} \circ \tau_1^{\otimes} \circ \tau_2^{\otimes}(X) \sqcup \ldots$ This infinite sequence may not converge, thus in general, widening is necessary to guarantee termination. However, in practice the sequence often converges after the first few elements.

The technique implemented in Aspic involves expanding multiple self-loops into a graph of which the paths represent these sequences, as shown in Fig. 8.4 in the case of three self-loops, and to solve iteratively the fixed point equations induced by the CFG, using widening if necessary. Moreover, there are also techniques [BFLP08, Gon07, MG11] for detecting circuits of length greater than one and accelerating them by concatenating their transitions. We focus here on self-loops.

8.2 Logico-Numerical Abstract Acceleration

Our goal is to exploit abstract acceleration techniques *without resorting to a Boolean state space enumeration* in order to overcome the limitations of current tools (*e.g.*, [Gon]) w.r.t. the analysis of logico-numerical programs.

In this section, we will first discuss some related issues in order to motivate our approach, before presenting methods that make abstract acceleration applicable to a CFG that now may contain loops with operations on both, Boolean and numerical, variables.

8.2.1 Motivations for Our Approach

A first observation is that identifying self-loops is more complex when Boolean state variables are not fully encoded in the CFG. Indeed, if a CFG contains a *"syntactic"* self-loop (ℓ, τ, ℓ) with $\tau : g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \rightarrow (\boldsymbol{b}', \boldsymbol{x}') = \boldsymbol{f}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$, there is an *"effective"* self-loop only for those Boolean states $\boldsymbol{b} \in \varphi_\ell$ in location ℓ such that $g(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi}) \wedge \boldsymbol{b} = \boldsymbol{f}^b(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{\xi})$ is satisfiable[1]. For instance, the self-loop around location $\neg b_1$ in Fig. 8.2 is not an "effective" self-loop.

This observation also applies to circuits, where, moreover, *numerical inputs have to be duplicated*: If there is a circuit (ℓ, τ_1, ℓ') and (ℓ', τ_2, ℓ) with $\tau_i : g_i(\boldsymbol{s}, \boldsymbol{\xi}) \rightarrow \boldsymbol{s}' = \boldsymbol{f_i}(\boldsymbol{s}, \boldsymbol{\xi})$

[1] We assume here that Boolean inputs β have been encoded by non-determinism (cf. §7.3).

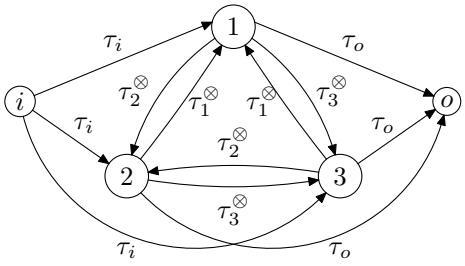


Figure 8.4: Computation of three accelerable self-loops τ_1, τ_2 and τ_3. τ_i and τ_o are the incoming resp. outgoing transitions of the location.

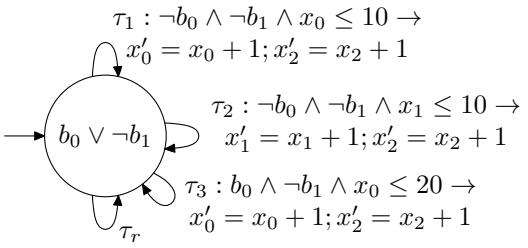


Figure 8.5: Acceleration of Ex. 8.1 in a CFG with a single location: The upper three self-loops are accelerable. The rest of the system is summarized in the transition τ_r where the Boolean equations are not the identity.

for $i = 1, 2$, the composed transition has the form $\tau : g(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}') \rightarrow \boldsymbol{s}'' = \boldsymbol{f}(\boldsymbol{s}, \boldsymbol{\xi}, \boldsymbol{\xi}')$. This limits in practice the length of circuits that can be accelerated. Here, we will not deal with such circuits and we focus on self-loops.

A naive approach to our problem could be to partition the system into sufficiently many locations, until we get self-loops that correspond to Def. 8.1. This approach is simple-minded for two reasons: (i) There might be no such Boolean states in the program at all; (ii) in the case of Fig. 8.2, simply ignoring the Boolean variable b_2 would make the (syntactic) self-loop accelerable without impacting the precision. More generally, it may pay off to slightly abstract the behavior of self-loops in order to benefit from precise acceleration techniques rather than relying on widening which may lose much more information in the end.

Another important remark is that we do not necessarily need to partition the system into locations to apply acceleration: it is sufficient to decompose the self-loops: Starting from the basic CFG with a single location and a single self-loop, we could split the loop into self-loops where the numerical transition function can be accelerated and the Boolean transition is the identity and a last self-loop where this is not the case. Fig. 8.5 shows the result of the application of this idea to our running example of Fig. 8.3.

This allows us to *separate the issue of accelerating self-loops in a symbolic CFG*, addressed in this section, *from the issue of finding a suitable CFG*, addressed in §8.3.

8.2.2 Decoupling Numerical and Boolean Transition Functions

We consider self-loops (ℓ, τ, ℓ) with $\tau : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}$ without any restriction on $\boldsymbol{f}^b$. We use the abstraction $\wp(E) = \wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\pi}{\overset{id}{\leftrightarrows}} \wp(\mathbb{B}^m) \times \wp(\mathbb{R}^n)$, where π is the function that approximates a set $S \in E$ by a Cartesian product, *e.g.*, $\pi((B_1 \times X_1) \cup (B_2 \times X_2)) = (B_1 \cup B_2) \times (X_1 \cup X_2)$. If τ is accelerable in the sense of abstract acceleration, then $\pi \circ \tau^* \subseteq \tau^\otimes$.

Our logico-numerical abstract acceleration method relies on *decoupling* the numerical and Boolean parts of the transition function τ with

$$\tau_b : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s}, \boldsymbol{i}) \\ \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{x} \end{pmatrix} \text{ and } \tau_x : g(\boldsymbol{s}, \boldsymbol{i}) \rightarrow \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}' \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s}, \boldsymbol{i}).\, \boldsymbol{b} \\ \boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) \end{pmatrix}.$$

We can approximate τ^* as follows (Fig. 8.6):

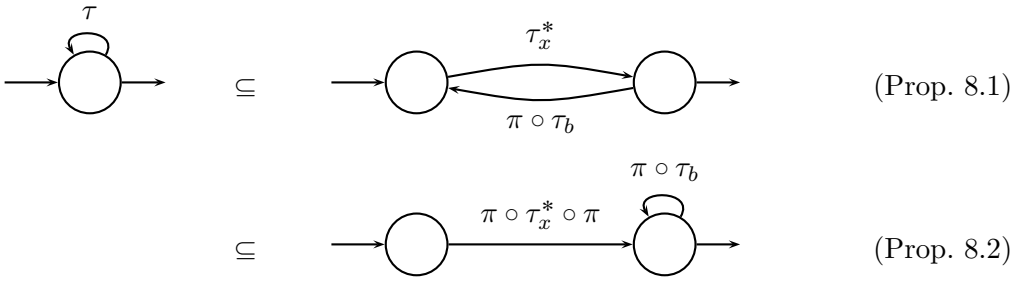


Figure 8.6: Decoupling numerical and Boolean transitions

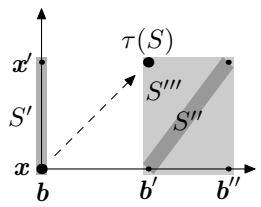


Figure 8.7: Illustration of the proof of Prop. 8.1

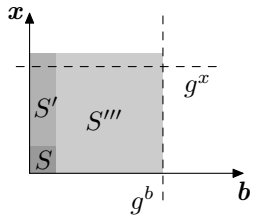


Figure 8.8: Illustration of the proof of Prop. 8.2 ($S \subseteq S' \subseteq S'''$)

Proposition 8.1 $\tau^* \subseteq (\pi \circ \tau_b \circ \tau_x^*)^*$.

Proof We prove first $\tau \subseteq \pi \circ \tau_b \circ (id \cup \tau_x)$:
Let $S = \{(\boldsymbol{b}, \boldsymbol{x})\}$, then $\tau_b(S) = \{(\boldsymbol{b'}, \boldsymbol{x})\}$, $\tau_x(S) = \{(\boldsymbol{b}, \boldsymbol{x'})\}$, and $\tau(S) = \{(\boldsymbol{b'}, \boldsymbol{x'})\}$:

$$\begin{array}{lllll} & \{(\boldsymbol{b},\boldsymbol{x}),(\boldsymbol{b},\boldsymbol{x'})\} & \subseteq & (id \cup \tau_x)(S) & = & S' \\ \Rightarrow & \{(\boldsymbol{b'},\boldsymbol{x}),(\boldsymbol{b''},\boldsymbol{x'})\} & \subseteq & \tau^b(S') & = & S'' \quad \text{with } \{(\boldsymbol{b''},\boldsymbol{x'})\} = \tau^b(\{(\boldsymbol{b},\boldsymbol{x'})\}) \\ \Rightarrow & \{(\boldsymbol{b'},\boldsymbol{x'})\} & \subseteq & \pi(S'') & = & S''' \end{array}$$

The graphical intuition of these steps is depicted in Fig. 8.7.
We conclude by

$$\begin{array}{lllll} & \tau & \subseteq & \pi \circ \tau_b \circ (id \cup \tau_x) & \\ \Rightarrow & \tau & \subseteq & \pi \circ \tau_b \circ \tau_x^* & \text{(because of } (id \cup \tau_x) \subseteq \tau_x^*) \\ \Rightarrow & \tau^* & \subseteq & (\pi \circ \tau_b \circ \tau_x^*)^* & \end{array}$$

∎

Now, we assume that τ_x is accelerable in the sense of Def. 8.1, which implies that $g(\boldsymbol{s}, \boldsymbol{i}) = g^b(\boldsymbol{b}, \boldsymbol{\beta}) \wedge g^x(\boldsymbol{x}, \boldsymbol{\xi})$ and $\boldsymbol{f}^x(\boldsymbol{s}, \boldsymbol{i}) = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$. By applying Prop. 8.1, we obtain that $(\pi \circ \tau_b \circ \tau_x^{\otimes})^*$ is a sound over-approximation of τ^*. However, this formula still involves Kleene iteration and thus widening is required (cf. §5.3.1).

But, there exists an alternative in which numerical and Boolean parts are computed in sequence, so that numerical acceleration is applied only once (Fig. 8.6):

Proposition 8.2 (Decoupling Boolean and numerical transition functions) *If τ_x is accelerable, then*

$$\begin{array}{ll} \textit{(1)} & (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi \textit{ is idempotent, and} \\ \textit{(2)} & \tau^* \subseteq (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi \end{array}$$

Proof The intuition for (1) is the following: If the guard $g^x \wedge g^b$ is satisfied, *i.e.*, the transition can be taken, then we compute first the transitive closure w.r.t. the numerical states before saturating the Boolean states. The application of τ_b does not enable "more"

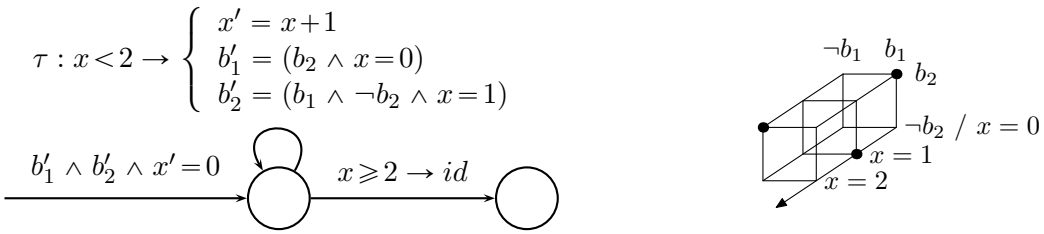


Figure 8.9: CFG of a counterexample (left-hand side) to show why the Boolean iterations cannot be computed exactly: the state $(\mathsf{ff}, \mathsf{tt}, 2)$ contained in $\tau^*(\{(\mathsf{tt}, \mathsf{tt}, 0)\}) = \{(\mathsf{tt}, \mathsf{tt}, 0), (\mathsf{tt}, \mathsf{ff}, 1), (\mathsf{ff}, \mathsf{tt}, 2)\}$ (dots in the right-hand side figure) is not part of $\pi \circ \tau_b^* \circ \pi \circ \tau_x^* \circ \pi(\{(\mathsf{tt}, \mathsf{tt}, 0)\}) = \{(\mathsf{tt}, \mathsf{tt}), (\mathsf{tt}, \mathsf{ff}), (\mathsf{ff}, \mathsf{ff})\} \times \{0, 1, 2\}$.

behavior of the numerical variables; hence, re-applying the τ_x^* has no effect.
We first compute $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S)$:

$$\begin{aligned}
\pi(S) &= B \times X \\
\tau_x^* \circ \pi(S) &= (B \times X) \cup \big((B \cap (\exists \boldsymbol{\beta} : g^b)) \times X'\big) \\
&\quad \text{with } X' \text{ s.t. } (\exists \boldsymbol{\xi} : \boldsymbol{a}((X \cup X') \cap g^x)) \subseteq X' \quad \text{(i)} \\
S' = \pi \circ \tau_x^* \circ \pi(S) &= B \times (X \cup X') \\
(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) &= S' \cup \overbrace{B' \times \big((X \cup X') \cap (\exists \boldsymbol{\xi} : g^x)\big)}^{S''} \\
&\quad \text{where } B' = \textstyle\bigcup_{k \geqslant 1} \tau_b^k(B, (X \cup X') \cap (\exists \boldsymbol{\xi} : g^x)) \\
&\quad \text{and with the property } (\pi \circ \tau_b)(S' \cup S'') \subseteq S'' \quad \text{(ii)} \\
S''' = (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) &= (B \cup B') \times (X \cup X')
\end{aligned}$$

Fig. 8.8 illustrates the sets S, S', and S'''. S''' is obviously stable by application of π. We show that it is also stable by application of τ_x and $\pi \circ \tau_b$, which allows to conclude that $(\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S''') = S'''$, hence the idempotency of the function:

$$\begin{aligned}
\tau_x(S''') &= ((B \cup B') \cap (\exists \boldsymbol{\beta} : g^b)) \times X'' \\
&\quad \text{with } X'' \subseteq X' \text{ because of property (i) above, hence} \\
\tau_x(S''') &\subseteq S''', \text{ and} \\
\pi \circ \tau_x^*(S''') &= S''' \\
\pi \circ \tau_b(S''') &= \pi \circ \tau_b(S''' \cap (\mathbb{B}^m \times (\exists \boldsymbol{\xi} : g^x))) \\
&= \pi \circ \tau_b((B \cup B') \times ((X \cup X') \cap (\exists \boldsymbol{\xi} : g^x))) \\
&\subseteq \pi \circ \tau_b(S' \cup S'') \\
&\subseteq S''' \text{ according to property (ii).}
\end{aligned}$$

Now, we can prove (2): from Prop. 8.1 follows

$$\begin{aligned}
\tau^* &\subseteq (\pi \circ \tau_b \circ \tau_x^*)^* \\
&= ((\pi \circ \tau_b)^* \circ \tau_x^*)^* \\
&\subseteq ((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)^* \\
&= (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi.
\end{aligned}$$

For the last step, we use the idempotency of the function and the fact that it includes the identity. ∎

Remark 8.1 (Why not τ_b^*?) *We cannot compute Boolean iterations exactly using τ_b^* instead of $(\pi \circ \tau_b)^*$. Fig. 8.9 gives a counterexample where $\tau^* \nsubseteq \pi \circ \tau_b^* \circ \pi \circ \tau_x^* \circ \pi$, which shows that using exact iterations τ_b^* would not give a sound decoupling.*

The following theorem implements Prop. 8.2 in the logico-numerical product domain A with the Galois connection $\wp(\mathbb{B}^m \times \mathbb{R}^n) \underset{\alpha}{\overset{id}{\leftrightarrows}} \wp(\mathbb{B}^m) \times Pol(\mathbb{R}^n)$ (see §7.2.3).

Theorem 8.1 (Logico-numerical abstract acceleration) *If a transition τ is such that τ_x is accelerable, then τ^* can be approximated in the logico-numerical product domain A with*

$$\tau^{\otimes}(B, X) = \left(\left(\tau_b^b \left[X^{\otimes} \right] \right)^* (B) \ , \ X^{\otimes} \right)$$

where

- $X^{\otimes} = (\tau_x^x)^{\otimes}(X)$
- $(\tau_x^x)^{\otimes}$ *is the abstract acceleration of* $\tau_x^x : g^x(\boldsymbol{x}, \boldsymbol{\xi}) \rightarrow \boldsymbol{x}' = \boldsymbol{a}(\boldsymbol{x}, \boldsymbol{\xi})$
- $\tau_b^b[X](B) = \tau_b(B, X^{\otimes} \sqcap^g (\exists \boldsymbol{\xi} : g^x))$
- $(\tau_b^b[X])^*(B) = lfp(\lambda B' \,.\, B \cup \tau_b^b[X](B'))$.

$(\tau_b^b[X])^*$ *converges in a finite number of iterations as it is the least fixed point of a monotonic function in the finite lattice* $\wp(\mathbb{B}^m)$.

In other words, we compute the reflexive and transitive closure $X^{\otimes}$ of τ_x using numerical abstract acceleration and saturate τ_b partially evaluated over $X^{\otimes}$.

Proof We prove that $\tau^{\otimes}(S)$ over-approximates the result of the formula of Prop. 8.2, *i.e.*, $((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)(S) \subseteq \tau^{\otimes} \circ \alpha(S)$ with $S \subseteq \wp(\mathbb{B}^m \times \mathbb{R}^n)$. This over-approximation is due to the convex approximations by the numerical abstract domain $Pol(\mathbb{R}^n)$.

$$\begin{array}{rl}
 & (\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi(S) \\
= & (\bigcup_{k \geqslant 0} \tau_b^k(B, (X \cup X') \cap (\exists \boldsymbol{\xi} : g^x)) \ , \ X \cup X') \\
 & \qquad \text{with } \pi(S) = (B, X) \\
 & \qquad \text{and } X' \text{ s.t. } (\exists \boldsymbol{\xi} : \boldsymbol{a}((X \cup X') \cap g^x)) \subseteq X' \text{ (see proof of Prop. 8.2)} \\
\subseteq & \lambda(B, X).\, \left(\bigcup_{k \geqslant 0} \tau_b^k(B, X^{\otimes} \sqcap^g (\exists \boldsymbol{\xi} : g^x)) \ , \ X^{\otimes} \right) \circ \alpha(S) \\
 & \qquad \text{with } \tau_x^x = \lambda X. \exists \boldsymbol{\xi} : \boldsymbol{a}(X \cap g^x) \\
 & \qquad \text{due to the soundness of numerical abstract acceleration:} \\
 & \qquad (X \cup X') \subseteq (\tau_x^x)^{\otimes} \circ \alpha(X) = X^{\otimes} \\
= & \lambda(B, X).((\tau_b^b[X^{\otimes}](B))^* \ , \ X^{\otimes}) \circ \alpha(S) \\
 & \qquad \text{with the notation } \tau_b^b[X](B) = \tau_b(B, X \sqcap^g (\exists \boldsymbol{\xi} : g^x))
\end{array}$$

Then, by $\tau^*(S) \subseteq ((\pi \circ \tau_b)^* \circ \pi \circ \tau_x^* \circ \pi)(S)$ (Prop. 8.2), we conclude $\tau^*(S) \subseteq \tau^{\otimes} \circ \alpha(S)$. ∎

Mind that, due to the convex approximation of the numerical sets, $\tau^{\otimes}$ is not idempotent in general (cf. §5.3.1).

8.2.3 Discussion

At the first glance, the approximations induced by this partial decoupling seem to be rather coarse. However, this is not severe in our context for two reasons:

1. The relations between Boolean and numerical variables that are lost by our method are mostly not representable in the abstract domain A anyway. For example, consider the loop $x \leqslant 4 \rightarrow (b' = \neg b; x' = x + 1)$, where b could be the least significant bit of a binary counter for instance: starting from $(b, x) \in \{(\mathtt{tt}, 0)\}$ the exact reachable set is $\{\mathtt{tt}\} \times \{0, 2, 4\} \cup \{\mathtt{ff}\} \times \{1, 3, 5\}$; its abstraction in A is

$\{\top\} \times \{0 \leqslant x \leqslant 5\}$. Hence, these relations will also be lost in a standard analysis merely relying on widening. Yet, due to numerical acceleration, we can even expect a better precision with our method.

2. We will apply this method to CFGs (see §8.3) in which the Boolean states defining a location exhibit the same numerical behavior and thus, decoupling is supposed not to seriously affect the precision.

Until now we studied the case of a single self-loop. In the presence of multiple self-loops, we expand the graph in the same way as with purely numerical transitions, *e.g.*, as shown in Fig. 8.4, and we apply Thm. 8.1 to each loop. As in the purely numerical case, widening must be applied in order to guarantee convergence.

Example 8.2 *We give the results obtained for Ex. 8.1: Analyzing the enumerated CFG in Fig. 8.3b using abstract acceleration gives* $0 \leqslant x_0 \leqslant 21 \wedge\ 0 \leqslant x_1 \leqslant 11 \wedge\ x_0 + x_1 \leqslant x_2 \leqslant 44$ *bounding all variables*[2]*. Analyzing the system on a CFG* with a single location *using decoupling and abstract acceleration still bounds two variables (*$0 \leqslant x_0 \leqslant 21 \wedge 0 \leqslant x_1 \leqslant 11 \wedge x_0 + x_1 \leqslant x_2$*), whereas, even on the enumerated CFG a standard analysis does not find any upper bound at all:* $0 \leqslant x_0 \wedge 0 \leqslant x_1 \wedge x_0 + x_1 \leqslant x_2$.

8.2.4 Variants

Decoupling accelerable from non-accelerable and Boolean transition functions. Theorem 8.1 applies only if the numerical transition functions are accelerable. If this is not the case, we can reuse the idea of Prop. 8.1, but now by decoupling the *accelerable numerical* functions (marked by the sub/superscript a) from *Boolean and non-accelerable numerical* functions (sub/superscripts b and n respectively):

$$\tau_a : g(\boldsymbol{s},\boldsymbol{i}) \to \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \lambda(\boldsymbol{s},\boldsymbol{i}).\,\boldsymbol{b} \\ \lambda(\boldsymbol{s},\boldsymbol{i}).\,\boldsymbol{x}_n \\ \boldsymbol{a}(\boldsymbol{x},\boldsymbol{\xi}) \end{pmatrix}\ ,\ \tau_{n,b} : g(\boldsymbol{s},\boldsymbol{i}) \to \begin{pmatrix} \boldsymbol{b}' \\ \boldsymbol{x}'_n \\ \boldsymbol{x}'_a \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}^b(\boldsymbol{s},\boldsymbol{i}) \\ \boldsymbol{f}^n(\boldsymbol{s},\boldsymbol{i}) \\ \lambda(\boldsymbol{s},\boldsymbol{i}).\,\boldsymbol{x}_a \end{pmatrix}$$

Proposition 8.3 (Decoupling accelerable and non-accelerable transitions)

$$\tau^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^*)^* \subseteq (\pi \circ \tau_{n,b} \circ \tau_a^{\otimes})^*$$

However, we cannot remove the Kleene iteration as in Prop. 8.2, because the function τ_a depends on non-accelerated numerical variables updated by $\tau_{n,b}$, and hence, widening is needed.

Using inputization techniques. Inputization (see [BBBS02], for instance) is a technique that treats some state variables as input variables. This method is useful to cut dependencies between the state variables, and thus, to remove loops. For example, it can be employed to reduce $\big((\pi \circ \tau_b)^* \circ \pi\big)$ to $\big(\pi \circ \tau'_b \circ \pi\big)$ in Prop. 8.2, where τ'_b is obtained by inputizing in τ_b those Boolean state variables that have a transition function that is neither the identity nor constant.

Example 8.3 (Inputization) *The loop* τ_b *can be approximated by the transition* τ'_b *where* β_0 *and* β_2 *correspond to* b_0 *and* b_2 *manipulated as Boolean inputs:*

$$\tau_b : \left| \begin{array}{l} b'_0 = \neg b_0 \\ b'_1 = b_1 \\ b'_2 = b_2 \wedge x \geqslant 0 \end{array} \right. \qquad \tau'_b : \left| \begin{array}{l} b'_0 = \beta_0 \\ b'_1 = b_1 \\ b'_2 = \beta_2 \wedge x \geqslant 0 \end{array} \right.$$

[2]This is an over-approximated result: the actual polyhedron has more constraints.

In our experiments (§8.4) we observed that the speed-up gained often pays off in comparison to the approximations it brings about.

8.3 Partitioning Techniques

The logico-numerical acceleration method described in the previous section can be applied to any CFG. However, in order to make it effective, we apply it to a CFG obtained by a partitioning technique that aims at alleviating the impact of decoupling Boolean and numerical transition functions on the precision. This section proposes such partitioning techniques that generate CFGs in which those Boolean states that exhibit the same numerical behavior ("numerical modes") are grouped in the same locations.

Basic technique. In order to implement this idea, we generate a CFG that is characterized by the following equivalence relation:

Definition 8.2 (Numerical modes)

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \left\{ \begin{array}{ll} \forall \boldsymbol{\beta}_1, \mathcal{C}: & \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \Rightarrow \\ & \exists \boldsymbol{\beta}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \\ \wedge \forall \boldsymbol{\beta}_2, \mathcal{C}: & \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \Rightarrow \\ & \exists \boldsymbol{\beta}_1 : \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \end{array} \right.$$

The intuition of this heuristics is to make equivalent the Boolean states that can execute the same set of *numerical actions*, guarded by the same numerical constraints.

Example 8.4 (Numerical modes) *We illustrate the application of this method to Example 8.1. We first factorize the numerical transition functions by actions:*

$$(x_0', x_1', x_2') = \begin{cases} (x_0{+}1\,, \quad x_1 \quad , x_2{+}1) & \text{if } (\neg b_0 \wedge \neg b_1 \wedge x_0 \leqslant 10) \ \vee \ (b_0 \wedge \neg b_1 \wedge x_0 \leqslant 20) \\ (\ \ x_0 \quad , x_1{+}1\,, x_2{+}1) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_1 \leqslant 10 \\ (\ \ 0 \quad , \ \ x_1 \quad , \ \ x_2 \ \ ) & \text{if } \neg b_0 \wedge \neg b_1 \wedge x_0 > 10 \wedge x_1 > 10 \\ (\ \ x_0 \quad , \ \ x_1 \quad , x_2{+}1) & \text{if } b_0 \wedge \neg b_1 \wedge x_0 > 20 \\ (\ \ x_0 \quad , \ \ x_1 \quad , \ \ x_2 \ \ ) & \text{otherwise} \end{cases}$$

Then by applying Def. 8.2, we get the equivalence classes $\{\neg b_0 \wedge \neg b_1, b_0 \wedge \neg b_1, b_0 \wedge b_1\}$*: the obtained CFG is the one of Fig. 8.3b.*

In the worst case, as in Ex. 8.4 above, a different set of actions can be executed in each Boolean state, thus the Boolean states will be enumerated. In the other extreme case, in all Boolean states the same set of actions can be executed, which induces a single equivalence class. Both cases are unlikely to occur in larger, real systems.

From an algorithmic point, we vectorize the numerical actions and factorize their common guards, which is equivalent to computing the product of the MtBdds representing the numerical transition functions (see §7.2.2):

$$\boldsymbol{f}^x(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_{1 \leqslant i \leqslant m} \big(g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}) \rightarrow \boldsymbol{a}_i(\boldsymbol{x}, \boldsymbol{\xi}) \big)$$

Then we eliminate the Boolean inputs β, and we decompose the results as follows

$$(\exists \boldsymbol{\beta} : g_i(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})) = \bigvee_{1 \leqslant j \leqslant n_i} g_{ij}^b(\boldsymbol{b}) \wedge g_{ij}^x(\mathcal{C})$$

where $g^x_{ij}(\mathcal{C})$ may be non-convex. The equivalence relation $\sim$ of Def. 8.2 can be reformulated as

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Longleftrightarrow \forall i \forall j : g^b_{ij}(\boldsymbol{b}_1) \Leftrightarrow g^b_{ij}(\boldsymbol{b}_2).$$

This last formulation reflects the fact that, in the resulting CFG, the numerical function $\boldsymbol{f}^x$ specialized on a location ℓ does not depend any more on $\boldsymbol{b}$, and hence, the precision loss is supposed to be limited.

Reducing the size of the partition. An option for having a less discriminating equivalence relation is to make equivalent the Boolean states that can execute the same set of numerical actions *regardless of the numerical constraints guarding them.*

Definition 8.3 (Numerical modes (forgetting numerical guards))

$$\boldsymbol{b}_1 \approx \boldsymbol{b}_2 \Leftrightarrow \left\{ \begin{array}{ll} \forall \boldsymbol{\beta}_1, \mathcal{C}_1 : & \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \Rightarrow \\ & \exists \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{f}^x(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) = \boldsymbol{f}^x(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \\ \quad \textit{and} & \textit{vice versa} \end{array} \right.$$

We clearly have $\sim \subseteq \approx$. For example, if we have two guarded actions $b \wedge x \leqslant 10 \rightarrow x' = x{+}1$ and $\neg b \wedge x \leqslant 20 \rightarrow x' = x{+}1$, $\sim$ will separate the Boolean states satisfying resp. b and $\neg b$, whereas $\approx$ will keep them together.

Another option is to consider only a subset of the numerical actions, that is, we ignore the transition functions of some numerical variables in Defs. 8.2 or 8.3. One can typically focus only on variables involved in the property. According to our experiments, this method is very efficient, but it relies on manual intervention.

Refining the partition by backward bisimulation. Given a partition, it can be refined by *Boolean backward bisimulation* (cf. [BFH91]).

Definition 8.4 (Boolean backward bisimulation) *Given an equivalence relation $\sim$, its backward Boolean bisimulation equivalence $\sim_\infty$ is defined by*

$$\boldsymbol{b}_1 \sim_0 \boldsymbol{b}_2 \Leftrightarrow \boldsymbol{b}_1 \sim \boldsymbol{b}_2 \wedge (\mathcal{I}(\boldsymbol{b}_1) = \mathcal{I}(\boldsymbol{b}_2))$$

$$\boldsymbol{b}_1 \sim_{k+1} \boldsymbol{b}_2 \Leftrightarrow \left\{ \begin{array}{l} \forall \boldsymbol{\beta}_1, \mathcal{C}_1, \boldsymbol{b}'_1 \textit{ such that } \mathcal{A}(\boldsymbol{b}'_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \wedge \boldsymbol{b}_1 = \boldsymbol{f}^b(\boldsymbol{b}'_1, \boldsymbol{\beta}_1, \mathcal{C}_1) : \\ \quad \exists \boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{b}_2 = \boldsymbol{f}^b(\boldsymbol{b}'_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{b}'_1 \sim_k \boldsymbol{b}'_2 \\ \textit{and vice versa} \end{array} \right.$$

The rationale behind this refinement is that partitioning the state space with Def. 8.2 ($\sim$) and stabilizing it by backward bisimulation yields a CFG with locations that group together states that are reachable (in the graph sense) by the same sequence of numerical actions from an initial state.

8.4 Experimental Evaluation

We have implemented the proposed methods in our experimentation tool ReaVer[3] on the basis of the logico-numerical abstract domain library BddApron.

Benchmarks. Besides some small, but difficult benchmarks, we used primarily benchmarks that are simulations of *production lines* (see Fig. 8.10), as modeled with the

[3] A first version of the tool which implemented only logico-numerical abstract acceleration methods was called nbAccel [SJ11].

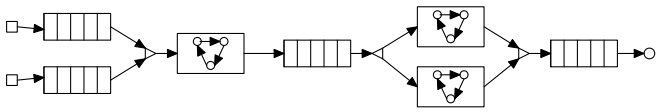


Figure 8.10: Schematic example of a production line with buffers, machines, and splitting and combining material flows.

library QUEST for the LCM language[4], for evaluating scalability. These models consist of building blocks like sources, buffers, machines, routers for splitting and combining flows of material and sinks, that synchronize via handshakes. The properties we want to prove depend on numerical variables, *e.g.*, (1) maximal throughput time of the first element passing the production line, or (2) minimal throughput of the production line. Inputs could model non-deterministic processing and arrival times, but we did not choose benchmarks with numerical inputs in order to enable a comparison with ASPIC [Gon].

Results. We compared our tool REAVER with NBAC [JHR99, Jea00, Jea03] and ASPIC. The results are summarized in Table 8.1. The tools where launched with the default options; for REAVER we use the partitioning heuristics of Def. 8.3 and the inputization technique of §8.2.4. We do not need the decoupling technique of Prop. 8.3 for our examples.

Discussion. The experimental comparison gives evidence about the advantages of abstract acceleration, but also some potential for future improvement:

- REAVER can prove a lot of examples where NBAC fails: this is due to the fact that abstract acceleration improves precision, especially in nested loops where the innermost loop can be "flattened", which makes it possible to recover more information in descending iterations.
- REAVER seems to scale better than NBAC: First, abstract acceleration reduces the number of iterations and fixed point checks. Second, our heuristics generates a partition that is well-suited for analysis – though, for some of the larger benchmarks, *e.g.*, LCM quest 4-1, the dynamic partitioning of NBAC starts to pay off, whereas our static partition is more fine-grained than necessary, which makes us waste time during analysis.
- Once provided with an enumerated CFG, ASPIC is very fast on the smaller benchmarks. However, the current version (3.1) cannot deal with CFGs larger than a few hundred locations. We were surprised that some of the small examples were not proved by ASPIC. We found out that this is due to our fixed point iteration strategy that uses a higher widening delay in strongly connected components resulting from the unfolding of multiple self-loops.
- The analysis using logico-numerical acceleration proved twice as many benchmarks and turned out to be 20% faster than a standard analysis of the same CFG with widening with delay 2 and two descending iterations.
- Applying the more refined partition of Def. 8.2 to our benchmarks had only a minor influence on performance and precision, and not applying inputization had no impact on the verification of properties, but it slowed down the analysis by 25% on average.
- Generally, for the benchmarks LCM quest 1 to 4, property 2 was not proved by the tools. Here, the combination of our heuristics with dynamic partitioning for further

[4] `http://www.3ds.com`

refining the critical parts of the CFG could help.

8.5 Conclusions and Perspectives

We proposed techniques for accelerating logico-numerical transitions, that allow us to benefit from the precision gain by numerical abstract acceleration as used in the tool Aspic, while tackling the Boolean state space explosion problem encountered when analyzing logico-numerical programs.

Experimentally, our tool ReaVer is often able to prove properties for the larger benchmarks, unlike the two other tools we tested – and this on CFGs that are ten times smaller than the CFGs obtained by enumeration of the reachable Boolean state space.

Although our method is based on the partial decoupling of the Boolean and numerical transitions, the experiments confirm our intuition that our method generally improves the precision. We attribute this to the following observations: first, numerical abstract acceleration reduces the need for widening; second, the information that we might lose by decoupling would often not be captured by the abstract domain anyway; and at last, the CFG obtained by our partitioning method particularly favors the application of our logico-numerical acceleration method.

Perspectives. Regarding abstract acceleration, the acceleration of multiple self-loops deserves additional investigation: We explained that, in case of nested loops, we have to resort to widening in order to guarantee convergence. Multiple self-loops, *i.e.*, several self-loops around one location, are a special case of nested loops. Gonnord et al [GH06, Gon07] developed abstract acceleration formulas for some special cases of multiple self-loops. In the general case, they apply a graph transformation method based on a partial unfolding (see §8.1) in order to compute more precise fixed points of multiple self-loops. However, this graph transformation is costly since it transforms 1 location with n self-loops into n locations with $n(n-1)$ arcs.

In the context of applying Presburger-based acceleration to program parallelization, Beletska et al. [BBBC09] propose an interesting approach which avoids this problem: for multiple self-loops $\tau_i = g_i \rightarrow \boldsymbol{a}_i$, they do not compute $(\bigcup_i \tau_i(X))^*$, but they use the more efficient, though less precise formula: $\bigcup_i \left(\tau_i \left(\bigcup_j \boldsymbol{a}_j(X)\right)^*\right)$. This computation scheme resembles the one of the derivative closure method of Ancourt et al. [ACI10] (cf. §5.3.3). This approach could be considered to improve the abstract acceleration of multiple self-loops.

Concerning partition refinement, the combination of our approach with dynamic partitioning *à la* [Jea03] seems to be worth pursuing. In particular, partitioning according to numerical constraints is mandatory for proving properties relying on non-convex invariants – our partitioning techniques partition only the Boolean state space. Such improvements should allow a wider range of benchmarks to be tackled.

		Aspic		ReaVer		Nbac	
	vars	size	time	size	time	size	time
Gate 1	4/4/2	7	?	5	**0.73**	24	?
Escalator 1	5/4/2	12	**0.14 (0.04)**	9	0.49	22	?
Traffic 1	4/6/0	18	**0.14 (0.01)**	16	0.19	5	3.49
Traffic 2	4/8/0	18	?	16	**0.35**	28	?
LCM Quest 0a-1	7/2/0	7	**0.04 (0.01)**	5	**0.04**	5	0.05
LCM Quest 0a-2	7/3/0	6	**0.05 (0.01)**	4	**0.05**	8	0.19
LCM Quest 0b-1	10/3/0	19	**0.08 (0.01)**	12	**0.08**	9	?
LCM Quest 0b-2	10/4/0	17	**0.09 (0.01)**	11	0.20	33	?
LCM Quest 0c-1	15/4/0	28	0.17 (0.01)	16	**0.16**	8	0.86
LCM Quest 0c-2	15/5/0	25	**0.20 (0.05)**	14	0.24	50	14.8
LCM Quest 1-1	16/5/0	114	1.99 (0.48)	42	**0.92**	6	2.45
LCM Quest 1-2	16/6/0	100	?	34	?	>156	>
LCM Quest 1b-1	16/5/0	55	0.92 (0.04)	29	**0.37**	15	?
LCM Quest 1b-2	16/5/0	45	0.76 (0.12)	23	**0.47**	61	?
LCM Quest 2-1	17/6/0	247	c	82	**7.84**	9	12.8
LCM Quest 2-2	17/7/0	198	>	62	?	>76	>
LCM Quest 3-1	25/5/0	483	26.5 (14.4)	58	8.49	12	**3.76**
LCM Quest 3-2	25/6/0	481	c	54	?	>1173	>
LCM Quest 3b-1	26/6/0	1724	>	170	43.8	14	**19.1**
LCM Quest 3b-2	26/7/0	1710	>	162	>	>32	>
LCM Quest 3c-1	26/6/0	1319	>	130	**34.2**	9	?
LCM Quest 3c-2	26/7/0	1056	c	98	>	>70	>
LCM Quest 3d-1	26/6/0	281	>	81	**5.43**	49	?
LCM Quest 3d-2	26/7/0	266	c	73	?	446	?
LCM Quest 3e-1	27/7/0	638	>	140	**20.6**	49	?
LCM Quest 3e-2	27/8/0	514	>	110	**6.46**	>28	>
LCM Quest 4-1	27/7/0	4482	>	386	186	9	**50.1**
LCM Quest 4-2	27/8/0	3586	>	290	>	>6	>

vars : Boolean state variables / numerical state variables / Boolean inputs
size : number of locations of the CFG
time : in seconds (Aspic: total time (time for analysis))
? : "don't know" (property not proved)
> : timed out after 600*s*
c : out of memory or crashed

Table 8.1: Experimental comparison between Aspic, ReaVer and Nbac.

Chapter 9

Logico-Numerical Max-Strategy Iteration

In this chapter we present a method for applying *max-strategy iteration* to the analysis of logico-numerical programs.

Strategy iteration methods (see §3.4.3) are able to solve the fixed point equation associated with the reachability analysis *without* the need for a widening operator. They can be applied to template domains, *i.e.*, abstract domains with *a priori* fixed constraints for which constant bounds are determined during the analysis. However, these techniques are limited to numerical programs. In order to avoid the state space explosion involved in transforming a logico-numerical program into a numerical one, we will use the approach of *state space partitioning* and abstract interpretation with *logico-numerical abstract domains* (see §7). Our method is based on max-strategy iteration, to which we will give an introduction in §9.1.

Outline. Our contributions can be summarized as follows (see Fig. 9.1):
1. We describe a method for computing the set of reachable states of a logico-numerical program based on max-strategy iteration (§9.2). The technique interleaves truncated Kleene iterations over a logico-numerical abstract domain with numerical max-strategy iterations. The method is optimal, *i.e.*, it computes the *least* fixed point w.r.t. the abstract semantics.
2. We give the results of an experimental evaluation (§9.3) examining various aspects concerning the efficiency and precision of the approach.

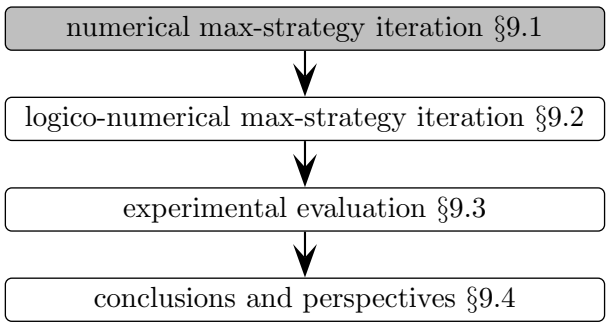


Figure 9.1: Chapter organization

9.1 Numerical Max-Strategy Iteration

Preliminaries. We consider programs modeled as *control flow graphs* (§2.1.2) over a state space Σ with transition relations $R \subseteq \Sigma^2$. In the case of affine programs $\Sigma = \mathbb{R}^n$ and the relations R are convex polyhedra.

We use the abstract domain of *template polyhedra* $Pol^{\mathbf{T}}$ (§3.3.3). A template polyhedron ($\mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d}$) is defined by a template constraint matrix $\mathbf{T} \in \mathbb{R}^{m\times n}$; an abstract value is represented by the vector of bounds $\boldsymbol{d} \in \overline{\mathbb{R}}^m$ where $\overline{\mathbb{R}}^m = \mathbb{R} \cup \{-\infty, \infty\}$. The analysis tries to find the smallest values of $\boldsymbol{d}$ representing a fixed point of the semantic equations. $\top$ and $\bot$ are naturally represented by the bound vectors ∞ and $-\infty$ respectively. The domain operations can be performed efficiently with the help of linear programming (LP) solvers.

Max-Strategy Iteration

Max-strategy iteration [GS07a, GS07b, GS08, GS10, GS11] is a method for computing the least solution of a system of equations $\mathcal{M}$ of the form $\boldsymbol{\delta} = \boldsymbol{F}(\boldsymbol{\delta})$, where $\boldsymbol{\delta}$ are the template bounds, and F_i, $0 \leqslant i \leqslant n$ is a finite maximum of monotonic and concave operators $\mathbb{R}^n \to \mathbb{R}$; in our case they are affine functions. The max-strategy improvement algorithm for affine programs is guaranteed to compute the least fixed point of $\boldsymbol{F}$, and it has to perform at most exponentially many improvement steps, each of which takes polynomial time.

Semantic equations. The equation system $\mathcal{M}$ is constructed from the abstract semantics of the program's transitions:

$$\text{for each } \ell' \in L : \boldsymbol{\delta}_{\ell'} = \max\Big(\{\boldsymbol{d}^0_{\ell'}\} \cup \{[\![R]\!]^\sharp(\boldsymbol{\delta}_\ell) \mid (\ell, R, \ell') \in \mathcal{R}\}\Big)$$

where $\boldsymbol{d}^0 = \lambda\ell. \begin{cases} \infty & \text{for } \ell = \ell_0 \\ -\infty & \text{for } \ell \neq \ell_0 \end{cases}$ denotes the initial values of the bounds, and

$$[\![R]\!]^\sharp(\boldsymbol{d}_\ell) = \sup\{\mathbf{T}_{\ell'}\boldsymbol{x}' \mid \mathbf{T}_\ell\boldsymbol{x} \leqslant \boldsymbol{d}_\ell \ \wedge \ R(\boldsymbol{x}, \boldsymbol{x}')\}.$$

We will view $\boldsymbol{d}$ alternatively as the concatenated vector of the bound vectors of all locations and the map $L \to \overline{\mathbb{R}}^m$ that assigns a vector of bounds $\boldsymbol{d}_\ell$ to each location ℓ: $\boldsymbol{d}(\ell) = \boldsymbol{d}_\ell$. Note, also, that we use $\boldsymbol{\delta}$ for denoting the vector of bound variables appearing in syntactic expressions, and $\boldsymbol{d}$ for the vector carrying the actual bounds. $\delta_{\ell,i}$ is the bound variable corresponding to the i^{th} line of the template in location ℓ.

Example 9.1 (Semantic equations) *Using the template constraints* $\begin{pmatrix} 1 \\ -1 \end{pmatrix} x \leqslant \begin{pmatrix} d_{\ell,1} \\ d_{\ell,2} \end{pmatrix}$ *in locations* ℓ*, the equation system for location* ℓ_1 *of the example in Fig. 9.2 consists of one equation for each template bound variable of which the arguments of the* max *operator are the initial value* $-\infty$ *and one expression* $[\![R]\!]^\sharp$ *for each of the three incoming arcs:*

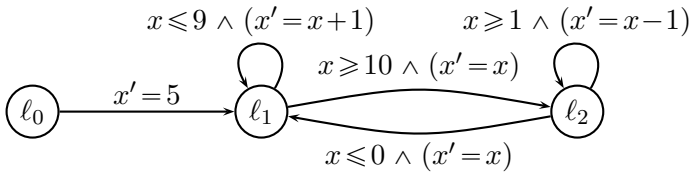


Figure 9.2: CFG of an affine program

$$\delta_{1,1} = \max \left\{ \begin{array}{llllllllll} -\infty, & & & & & & & & & \\ \sup\{ & x' & \mid & x \leqslant \delta_{0,1} & \wedge & -x \leqslant \delta_{0,2} & \wedge & x' = 5 & & \}, \\ \sup\{ & x' & \mid & x \leqslant \delta_{1,1} & \wedge & -x \leqslant \delta_{1,2} & \wedge & x \leqslant 9 & \wedge \quad x' = x+1 & \}, \\ \sup\{ & x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x \leqslant 0 & \wedge \quad x' = x & \} \end{array} \right\}$$

$$\delta_{1,2} = \max \left\{ \begin{array}{llllllllll} -\infty, & & & & & & & & & \\ \sup\{ & -x' & \mid & x \leqslant \delta_{0,1} & \wedge & -x \leqslant \delta_{0,2} & \wedge & x' = 5 & & \}, \\ \sup\{ & -x' & \mid & x \leqslant \delta_{1,1} & \wedge & -x \leqslant \delta_{1,2} & \wedge & x \leqslant 9 & \wedge \quad x' = x+1 & \}, \\ \sup\{ & -x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x \leqslant 0 & \wedge \quad x' = x & \} \end{array} \right\}$$

Strategies. A *strategy* μ induces a "subsystem" $\boldsymbol{\delta} = \widehat{\boldsymbol{F}}(\boldsymbol{\delta})$ of $\mathcal{M}$ in the sense that exactly one argument $\widehat{F}_i$ of the max operator on the right-hand side of each equation $\delta_i = \max(\ldots, \widehat{F}_i(\boldsymbol{\delta}), \ldots)$ is chosen. Intuitively, this means that a strategy selects exactly one "incoming transition" for each template bound variable in each location ℓ'.

Example 9.2 (Strategy) *A strategy in the example in Fig. 9.2 corresponds for instance the following system of equations:*

$$\begin{array}{lllllllllll} \delta_{0,1} & = \infty & & & & & & & & & \\ \delta_{0,2} & = \infty & & & & & & & & & \\ \delta_{1,1} & = \sup\{ & x' & \mid & x \leqslant \delta_{1,1} & \wedge & -x \leqslant \delta_{1,2} & \wedge & x \leqslant 9 & \wedge \quad x' = x+1 & \} \\ \delta_{1,2} & = \sup\{ & -x' & \mid & x \leqslant \delta_{0,1} & \wedge & -x \leqslant \delta_{0,2} & \wedge & x' = 5 & & \} \\ \delta_{2,1} & = -\infty & & & & & & & & & \\ \delta_{2,2} & = -\infty & & & & & & & & & \end{array}$$

We see that for $\delta_{1,1}$ we have chosen the third and for $\delta_{1,2}$ the second argument of the max *operators in the equations of Example 9.1.*

One has to compute the least solution $\mathit{lfp}[\![\mathcal{M}]\!]$ of the system $\mathcal{M}$, where $[\![\mathcal{M}]\!]$ is defined as

$$[\![\mathcal{M}]\!](\boldsymbol{d}) = \max_{\mu \text{ in } \mathcal{M}} [\![\mu]\!](\boldsymbol{d})$$

and with $[\![\mu]\!](\boldsymbol{d}) = [\![\boldsymbol{\delta} = \widehat{\boldsymbol{F}}(\boldsymbol{\delta})]\!](\boldsymbol{d}) = \widehat{\boldsymbol{F}}(\boldsymbol{d})$.

Max-strategy improvement. $\mathit{lfp}[\![\mathcal{M}]\!]$ is computed with the help of the max-strategy improvement algorithm (see Fig. 9.3) which iteratively improves strategies μ until the least fixed point $\mathit{lfp}[\![\mu]\!]$ of a strategy equals $\mathit{lfp}[\![\mathcal{M}]\!]$.

The least fixed point $\mathit{lfp}[\![\mu]\!]$ of a strategy μ can be computed by solving the LP problem with the constraint system

$$\text{for each } \left(\boldsymbol{\delta}_{\ell'} = [\![R]\!]^\sharp(\boldsymbol{\delta}_\ell)\right) \text{ in } \mu : \boldsymbol{d}_{\ell'} \leqslant \mathbf{T}_{\ell'} \boldsymbol{x}' \ \wedge \ \mathbf{T}_\ell \boldsymbol{x} \leqslant \boldsymbol{d}_\ell \ \wedge \ R(\boldsymbol{x}, \boldsymbol{x}')$$

initial strategy: $\mu := \{\boldsymbol{\delta}_{\ell_0} = \infty,\ \boldsymbol{\delta}_\ell = -\infty \text{ for all } \ell \neq \ell_0\}$
initial bounds: $\boldsymbol{d} := \lambda\ell.\boldsymbol{\delta}_\ell \to \begin{cases} \infty & \text{for } \ell = \ell_0 \\ -\infty & \text{for } \ell \neq \ell_0 \end{cases}$
while not $\boldsymbol{d}$ is a solution of $\mathcal{M}$ do
  $\mu := \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d})$
  $\boldsymbol{d} := \mathit{lfp}[\![\mu]\!]$
done
return $\boldsymbol{d}$

Figure 9.3: Max-strategy iteration algorithm

(where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are auxiliary variables) and the objective function $\max \sum_i d_i$, *i.e.*, the sum of all bounds $\boldsymbol{d}$.

μ' is called an *improvement* of μ w.r.t. $\boldsymbol{d}$, *i.e.*, $\mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu, \boldsymbol{d})$ iff

1. μ' is "at least as good" as μ: $[\![\mu']\!](\boldsymbol{d}) \geqslant [\![\mu]\!](\boldsymbol{d})$, and
2. μ' is "strictly better for the changed equations": if $(\delta_i = \widehat{F}_i(\boldsymbol{\delta}))$ in μ and $(\delta_i \geqslant \widehat{F}'_i(\boldsymbol{\delta}))$ in μ' and $\widehat{F}_i \neq \widehat{F}'_i$, then $\widehat{F}'(\boldsymbol{d}) > \widehat{F}(\boldsymbol{d})$.

Example 9.3 (Max-strategy iteration) *We illustrate some steps of the analysis of the example in Fig. 9.2. Assume the current strategy is:*

$$\mu_1 = \left\{ \begin{array}{l} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = \sup\{\ \ x' \mid x \leqslant \delta_{0,1} \wedge -x \leqslant \delta_{0,2} \wedge x' = 5\ \} \\ \delta_{1,2} = \sup\{ -x' \mid x \leqslant \delta_{0,1} \wedge -x \leqslant \delta_{0,2} \wedge x' = 5\ \} \\ \delta_{2,1} = -\infty \\ \delta_{2,2} = -\infty \end{array} \right\}$$

and the current template bounds are:

$$\boldsymbol{d}_1 = \left\{ \begin{array}{llll} \delta_{0,1} & \to \infty & \delta_{0,2} & \to \infty \\ \delta_{1,1} & \to 5 & \delta_{1,2} & \to -5 \\ \delta_{2,1} & \to -\infty & \delta_{2,2} & \to -\infty \end{array} \right\}$$

The strategy can only be improved w.r.t. $\delta_{1,1}$:

$$\mu_2 = \left\{ \begin{array}{l} \delta_{0,1} = \infty \\ \delta_{0,2} = \infty \\ \delta_{1,1} = \sup\{\ \ x' \mid x \leqslant \delta_{1,1} \wedge -x \leqslant \delta_{1,2} \wedge x \leqslant 9 \wedge x' = x+1\ \} \\ \delta_{1,2} = \sup\{ -x' \mid x \leqslant \delta_{0,1} \wedge -x \leqslant \delta_{0,2} \wedge x' = 5\ \} \\ \delta_{2,1} = -\infty \\ \delta_{2,2} = -\infty \end{array} \right\}$$

We compute the new fixed point w.r.t. μ_2:

$$\boldsymbol{d}_2 = \mathit{lfp}[\![\mu_2]\!] = \left\{ \begin{array}{llll} \delta_{0,1} & \to \infty & \delta_{0,2} & \to \infty \\ \delta_{1,1} & \to 10 & \delta_{1,2} & \to -5 \\ \delta_{2,1} & \to -\infty & \delta_{2,2} & \to -\infty \end{array} \right\}$$

In the next step we can improve the strategy w.r.t. $\delta_{2,1}$ and $\delta_{2,2}$, a.s.o.

An improving strategy is selected by testing for each equation whether an argument of its max-operator leads to a greater bound. Since the arguments of the max-operator

are required to be monotonic, the bounds are always monotonically increasing and, thus, arguments that have already been selected in previous strategies need not be considered again.

9.2 Logico-Numerical Max-Strategy Iteration

We will present an algorithm which enables the use of max-strategy iteration in a logico-numerical context, *i.e.*, programs with a state space $\mathbb{B}^p \times \mathbb{R}^n$.

Example 9.4 *An example for such a logico-numerical program is the following C program:*

```
b1=true; b2=true; x=0;
while(true)
{
  while(x<=19) { x = (b1 ? x+1 : x-1); }
  while(x<=99) { x = (b2 ? x+1 : x); b2 = !b2; }
  if (x>=100) { b1 = (x<=100); x = x-100; }
}
```

Fig. 9.5 in §9.2.2 shows a CFG corresponding to this program. Note that we allow numerical constraints in assignments to Boolean variables. A program property we want to prove is for instance the invariant $0 \leqslant x \leqslant 100$.

9.2.1 Abstract Domain

We consider the logico-numerical abstract domain $A = \wp(\mathbb{B}^p) \times \overline{\mathbb{R}}^m$ which combines Boolean formulas with template polyhedra. An abstract value $S^\sharp = (B, \boldsymbol{d})$ consists of the cartesian product of valuations of the Boolean variables B (represented as Boolean formulas using BDDs for example) and the template bounds $\boldsymbol{d}$. We define the domain operations:

– Abstraction: $\alpha_{\mathbf{T}}(S) = \begin{pmatrix} \{\boldsymbol{b} \mid \exists \boldsymbol{x} : (\boldsymbol{b}, \boldsymbol{x}) \in S\} \\ \min\{\boldsymbol{d} \mid (\boldsymbol{b}, \gamma^{\boldsymbol{x}}_{\mathbf{T}}(\boldsymbol{d})) \in S\} \end{pmatrix}$

– Concretization: $\gamma_{\mathbf{T}}(S^\sharp) = B \wedge \gamma^{\boldsymbol{x}}_{\mathbf{T}}(\boldsymbol{d})$

– Join: $\begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix} \sqcup_{\mathbf{T}} \begin{pmatrix} B' \\ \boldsymbol{d}' \end{pmatrix} = \begin{pmatrix} B \vee B' \\ \boldsymbol{d} \sqcup^{\boldsymbol{x}}_{\mathbf{T}} \boldsymbol{d}' \end{pmatrix}$

– Image: $[\![R_{\ell,\ell'}]\!]^\sharp \begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix} = \begin{pmatrix} \{\boldsymbol{b}' \mid \mathbf{T}_\ell \boldsymbol{x} \leqslant \boldsymbol{d} \wedge \boldsymbol{b} \in B \wedge R(\boldsymbol{b}, \boldsymbol{b}', \boldsymbol{x}, \boldsymbol{x}')\} \\ \sup \{\mathbf{T}_{\ell'} \boldsymbol{x}' \mid \mathbf{T}_\ell \boldsymbol{x} \leqslant \boldsymbol{d} \wedge \boldsymbol{b} \in B \wedge R(\boldsymbol{b}, \boldsymbol{b}', \boldsymbol{x}, \boldsymbol{x}')\} \end{pmatrix}$

$\top$ and $\bot$ are defined as $\begin{pmatrix} \mathsf{tt} \\ \infty \end{pmatrix}$ and $\begin{pmatrix} \mathsf{ff} \\ -\infty \end{pmatrix}$ respectively.

Since we are analyzing a CFG with locations L, we have the overall abstract domain $\Sigma^\sharp = L \to A$. An abstract value $S^\sharp = \lambda\ell.(B_\ell, \boldsymbol{d}_\ell) \in \Sigma^\sharp$ assigns to each location a value of the above logico-numerical domain. Note that the dimension m of $\boldsymbol{d}_\ell$ may depend on ℓ if the templates differ from location to location.

```
1   S := S^0
2   S' = post(S)
3   while ¬stable(S, S') do
4     while ¬p_stable(S, S') do                    ⎫
5       S := S'                                    ⎪
6       S' = post(S)                               ⎪  phase (1): truncated logico-numerical Kleene iteration
7     done                                         ⎭
8     S := S'
9     M = generate(S)
10    μ := (δ = d)
11    μ' = max_improve_M(μ, d)
12    while μ' ≠ μ do                              ⎫
13      μ := μ'                                    ⎪
14      d := lfp⟦μ⟧                                ⎬  phase (2): numerical max-strategy iteration
15      μ' = max_improve_M(μ, d)                   ⎪
16    done                                         ⎭
17    S' = post(S)
18  done
19  return S
```

Figure 9.4: Logico-numerical max-strategy iteration algorithm

9.2.2 Algorithm

Our analysis is based on alternating (1) a truncated Kleene iteration over the logico-numerical abstract domain and (2) numerical max-strategy iteration, see Fig. 9.4.

The truncated Kleene iteration (phase (1)) explores the system until a certain criterion is satisfied; we say that the system *preliminarily stable*. We use the following criterion: we stop Kleene iteration if for all locations the set of reachable Boolean states does not change whatever transition we take. The underlying idea is to discover a subsystem, in which Boolean variables are stable during a presumably larger number of iterations. In such a subsystem numerical variables evolve, while Boolean transitions switch only within the system, *i.e.*, they do not "discover" new Boolean states, until numerical conditions are satisfied that make Boolean variables leave the subsystem.

We use max-strategy iteration (phase (2)) to compute the fixed point for the numerical variables for such a subsystem. Then Kleene iteration (phase (1)) continues exploring the next preliminary stable subsystem. The algorithm terminates in a finite number of steps, as soon as the Kleene iteration of phase (1) has reached a fixed point.

Formal description. See Fig. 9.4. Since we only manipulate abstract quantities, we will omit the superscript $^\sharp$ in the sequel in order to improve readability.

For phase (1) we use the definitions:

– Initial abstract value: $S^0 = \lambda\ell.\begin{cases} \top & \text{for } \ell = \ell_0 \\ \bot & \text{for } \ell \neq \ell_0 \end{cases}$

– Post-condition: $\mathsf{post}(S) = \lambda\ell'.S(\ell') \sqcup \bigsqcup_\ell [\![R_{\ell,\ell'}]\!](S(\ell))$

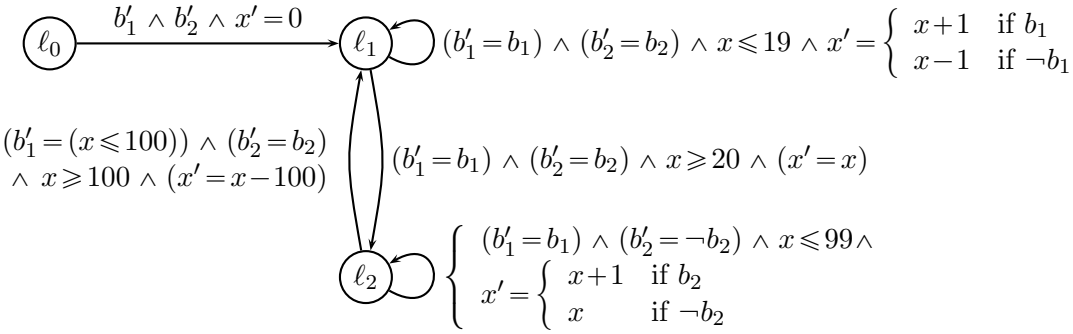


Figure 9.5: CFG of the program in Example 9.4

– Condition for preliminary stability: $\mathsf{p\_stable}(S, S') = (\forall \ell : B_\ell = B'_\ell)$

– Condition for stability (global convergence): $\mathsf{stable}(S, S') = (S = S')$

For phase (2) we define the following:

– The max-strategy improvement operator $\mathsf{max\_improve}_{\mathcal{M}}$ is defined as described in §9.1.

– The operator $\mathsf{generate}$ dynamically derives the system of equations for the current preliminary stable subsystem: this means that we restrict the system to those transitions that stay within the subsystem defined by the current Boolean states $\lambda\ell.B_\ell$. For this purpose we conjoin the term $\boldsymbol{b} \in B_\ell \wedge \boldsymbol{b'} \in B_{\ell'}$ to the transition relation in the definition below. Strategies may only contain convex constraints: thus, we transform the relation into disjunctive normal form and generate one strategy per disjunct:

$$\mathsf{generate}(S) = \bigcup_{\ell',\ell} \mathsf{decomp\_convex}(\exists \boldsymbol{b}, \boldsymbol{b'} : R_{\ell,\ell'}(\boldsymbol{b}, \boldsymbol{x}, \boldsymbol{b'}, \boldsymbol{x'}) \wedge \boldsymbol{b} \in B_\ell \wedge \boldsymbol{b'} \in B_{\ell'})$$

where $\mathsf{decomp\_convex}(R_{\ell,\ell'}) = \bigcup_j \left(\boldsymbol{\delta}_{\ell'} = [\![R^j_{\ell,\ell'}]\!](\boldsymbol{\delta}_\ell)\right)$ with $R_{\ell,\ell'} = \bigvee_j R^j_{\ell,\ell'}$ and $R^j_{\ell,\ell'}$ convex.

Remark 9.1 *Since the bounds $\boldsymbol{d}$ are monotonically increasing, we use the constant strategy $\boldsymbol{\delta} = \boldsymbol{d}$ (where $\boldsymbol{d}$ are the previously obtained bounds) as initial strategy for phase (2) (see line 10 in Fig. 9.4). This prevents the numerical max-strategy improvement from restarting with $\boldsymbol{\delta} = -\infty$ each time.*

We illustrate this algorithm by applying it to the CFG in Fig. 9.5:

Example 9.5 *We use the template constraint matrix* $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ *which corresponds to an interval analysis. In order to make the presentation more concise, we will write states* $(\ell \to \begin{pmatrix} B \\ \boldsymbol{d} \end{pmatrix}) \in \Sigma$ *as* $\ell \to \begin{pmatrix} \varphi(b_1, b_2) \\ [-\delta_{\ell,2}, \delta_{\ell,1}] \end{pmatrix}$ *where φ is a Boolean formula.*

The initial state in ℓ_0 is $\begin{pmatrix} \mathsf{tt} \\ [-\infty, \infty] \end{pmatrix}$. *We start exploring the system by taking transition (ℓ_0, R, ℓ_1):* $\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0, 0] \end{pmatrix}$. *We continue propagating through (ℓ_1, R, ℓ_1):*

$\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0,1] \end{pmatrix}$. *Now, we have reached preliminary stability because none of the transitions makes us discover new Boolean states in the next iteration ((ℓ_0, R, ℓ_1) and (ℓ_1, R, ℓ_1) yield nothing new w.r.t. Boolean states and the other transitions are infeasible,* i.e.*, they give $\bot$). Hence, we go ahead to phase (2) and extract the numerical equation system for each (ℓ, R, ℓ').* E.g. *for (ℓ_1, R, ℓ_1), we compute:*

$$\exists \boldsymbol{b}, \boldsymbol{b'} : R \wedge b_1 \wedge b_2 \wedge b'_1 \wedge b'_2 = (x' = x+1 \wedge x \leqslant 19)$$

which gives us the partial equations:

$$\begin{array}{lllllllllll} \delta_{1,1} = \sup\{ & x' & \mid & x \leqslant \delta_{1,1} & \wedge & -x \leqslant \delta_{1,2} & \wedge & x' = x+1 & \wedge & x \leqslant 19 & \} \\ \delta_{1,2} = \sup\{ & -x' & \mid & x \leqslant \delta_{1,1} & \wedge & -x \leqslant \delta_{1,2} & \wedge & x' = x+1 & \wedge & x \leqslant 19 & \} \end{array}$$

which have to be completed by the other incoming arcs of ℓ_1. We start the max-strategy iteration with the strategy corresponding to the values obtained in phase (1):

$$\mu = \{\boldsymbol{\delta}_0 = \infty, \qquad \delta_{1,1} = 1, \qquad \delta_{1,2} = 0, \qquad \boldsymbol{\delta}_2 = -\infty\}$$

We observe that we can improve this strategy w.r.t. $\delta_{1,1}$:

$$\mu' = \left\{ \begin{array}{l} \boldsymbol{\delta}_1 = \infty \\ \delta_{1,1} = \sup\{x' \mid x \leqslant \delta_{1,1} \ \wedge \ -x \leqslant \delta_{1,2} \ \wedge \ x' = x+1 \ \wedge \ x \leqslant 19\}, \quad \delta_{1,2} = 0 \\ \boldsymbol{\delta}_2 = -\infty \end{array} \right\}$$

The max-strategy iteration terminates with: $\ell_1 \to \begin{pmatrix} b_1 \wedge b_2 \\ [0,20] \end{pmatrix}$.

We continue propagating (phase (1)): By (ℓ_1, R, ℓ_2) we get $\ell_2 \to \begin{pmatrix} b_1 \wedge b_2 \\ [20,20] \end{pmatrix}$; then (ℓ_2, R, ℓ_2) results in $\begin{pmatrix} b_1 \wedge \neg b_2 \\ [21,21] \end{pmatrix}$; by joining these values we get $\ell_2 \to \begin{pmatrix} b_1 \\ [20,21] \end{pmatrix}$. Taking (ℓ_2, R, ℓ_2) a second time does not change the Boolean state: $\ell_2 \to \begin{pmatrix} b_1 \\ [20,22] \end{pmatrix}$. Taking any other transition does not discover new Boolean states either, thus, we move on to phase (2) and compute the numerical equation system w.r.t. the current Boolean state: For example for (ℓ_2, R, ℓ_2), we compute

$$(\exists \boldsymbol{b}, \boldsymbol{b'} : R \wedge b_1 \wedge b'_1) = \big((x' = x+1 \vee x' = x) \wedge x \leqslant 99\big)$$

which results in the partial equations

$$\begin{array}{l} \delta_{2,1} = \max \left\{ \begin{array}{llllllllll} \sup\{ & x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x' = x+1 & \wedge & x \leqslant 99 \ \}, \\ \sup\{ & x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x' = x & \wedge & x \leqslant 99 \ \} \end{array} \right\} \\ \delta_{2,2} = \max \left\{ \begin{array}{llllllllll} \sup\{ & -x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x' = x+1 & \wedge & x \leqslant 99 \ \}, \\ \sup\{ & -x' & \mid & x \leqslant \delta_{2,1} & \wedge & -x \leqslant \delta_{2,2} & \wedge & x' = x & \wedge & x \leqslant 99 \ \} \end{array} \right\} \end{array}$$

which have to be completed by the other incoming arcs of ℓ_2. The only possible improvement w.r.t to the current state is w.r.t. $\delta_{2,1}$; phase (2) results in $\ell_2 \to \begin{pmatrix} b_1 \\ [20,100] \end{pmatrix}$.

We continue with phase (1), which filters the above value through (ℓ_2, R, ℓ_1) augmenting the abstract value in ℓ_1 to $\begin{pmatrix} b_1 \\ [0,20] \end{pmatrix}$. Then, none *of the transitions makes the reachable state sets increase (neither Boolean nor numerical), hence we have reached the global fixed point:*

$$\ell_0 \to \top, \quad \ell_1 \to \begin{pmatrix} b_1 \\ [0,20] \end{pmatrix}, \quad \ell_2 \to \begin{pmatrix} b_1 \\ [20,100] \end{pmatrix}$$

Note that a logico-numerical analysis in the same domain with widening and descending iterations yields no information about this example: $S = \lambda\ell.\top$.

Application to data-flow programs. For our experiments in §9.3, we used LUSTRE programs. We generated a CFG using the state space partitioning technique of Def. 8.2. Then, the transition relations are constructed from the transition functions as follows:

$$R_{\ell,\ell'} = \exists \boldsymbol{\beta} : \left\{ \begin{array}{rcl} \boldsymbol{x}' & = & \boldsymbol{f}^x(\boldsymbol{b},\boldsymbol{x},\boldsymbol{\beta},\boldsymbol{\xi}) \\ \boldsymbol{b}' & = & \boldsymbol{f}^b(\boldsymbol{b},\boldsymbol{x},\boldsymbol{\beta},\boldsymbol{\xi}) \end{array} \right\} \wedge \varphi_\ell(\boldsymbol{x},\boldsymbol{b}) \wedge \varphi_{\ell'}(\boldsymbol{x}',\boldsymbol{b}') \wedge \mathcal{A}(\boldsymbol{b},\boldsymbol{x},\boldsymbol{\beta},\boldsymbol{\xi})$$

where φ_ℓ are the location definitions. Boolean input variables $\boldsymbol{\beta}$ are simply quantified existentially. Numerical inputs $\boldsymbol{\xi}$ appear as auxiliary variables (*i.e.*, variables without associated bounds) in the max-strategy iteration, hence, they are treated without modification of the algorithms.

9.2.3 Properties

Theorem 9.1 (Termination) *The logico-numerical max-strategy algorithm terminates after a finite number of iterations.*

Proof Termination follows from these observations:

(a) Phase (1) only propagates as long as new Boolean states are discovered; the number of Boolean states is finite.
(b) Max-strategy iteration is called at most once for each subset of Boolean states. The number of subsets of Boolean states is finite.
(c) There is a unique system of numerical equations (built by generate) for each subset of Boolean states.
(d) Max-strategy iteration terminates after a finite number of improvement steps, because there is a finite number of strategies and each strategy is visited at most once [GS07b].
(e) Max-strategy iteration returns the unique least fixed point w.r.t. the given system of equations [GS07b].

Thus, as soon as all reachable Boolean states have been discovered and the associated numerical fixed point has been computed, the overall fixed point has been reached and the algorithm terminates.

Theorem 9.2 (Soundness) *The logico-numerical max-strategy algorithm computes a fixed point of* $\lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.

Proof Let us denote

- $F = \lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.
- $\lambda S.(\mathit{lfp}^B\ F)(S)$ the truncated Kleene iteration phase (1)
  (lines 4 to 7 in Fig. 9.4),
  i.e., $\lambda S.\big(\text{while } B \neq B' \text{ do } S := S'; S' = F(S) \text{ end; return } S'\big)$.
- $\lambda(B,\boldsymbol{d}).(B,(\mathit{lfp}^X\ F^X)(\boldsymbol{d}))$ the max-strategy iteration in phase (2)
  (lines 9 to 16 in Fig. 9.4),
  i.e., $\lambda S.\big($ $\mathcal{M} = \mathsf{generate}(S); \mu := (\boldsymbol{\delta} = \boldsymbol{d}); \mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu,\boldsymbol{d})$;
  while $\mu' \neq \mu$ do $\mu := \mu'; \boldsymbol{d} := \mathit{lfp}[\![\mu]\!]; \mu' = \mathsf{max\_improve}_{\mathcal{M}}(\mu,\boldsymbol{d})$ done;
  return $S\big)$.

Then, we can write the whole algorithm as $\mathit{lfp}((id^B, \mathit{lfp}^X\ F^X) \circ (\mathit{lfp}^B\ F) \circ F)$.

Since $\mathit{lfp}^B\ F$ and $(id^B, \mathit{lfp}^X\ F^X)$ are both extensive, we can under-approximate them by $id = \lambda S.S$. Hence, we conclude from

$$\mathit{lfp}\ F \sqsubseteq \mathit{lfp}(\underbrace{id}_{id \sqsubseteq (id^B, \mathit{lfp}^X F^X)} \circ \underbrace{id}_{id \sqsubseteq (\mathit{lfp}^B F)} \circ F)$$

that our algorithm computes an over-approximation of the least fixed point, *i.e.*, it is sound.

Theorem 9.3 (Optimality) *The logico-numerical max-strategy algorithm computes the* least *fixed point of* $\lambda S.S_0 \sqcup [\![R]\!]^\sharp(S)$.

Proof Additionally to Thm. 9.2, we have to show that

$$\mathit{lfp}((id^B, \mathit{lfp}^X\ F^X) \circ (\mathit{lfp}^B\ F) \circ F) \sqsubseteq \mathit{lfp}\ F.$$

(a) Phase (1) computes a certain number of iterations $F^k(\bot) = (\mathit{lfp}^B\ F) \circ F(\bot)$ taking into account the whole transition system. We trivially have $F^k(\bot) \sqsubseteq \mathit{lfp}\ F(\bot)$.

(b) Phase (2) $(id^B, \mathit{lfp}^X\ F^X)$ iterates over the transitions of a subsystem. It is known [GS07b] that it computes the least fixed point w.r.t. this subsystem. Hence, the result of phase (2) cannot go beyond the fixed point of the whole system: $(id^B, \mathit{lfp}^X\ F^X) \circ F^k(\bot) \sqsubseteq \mathit{lfp}\ F$.

(c) We can repeat arguments (a) and (b) for the outer loop:
$\ldots \circ (id^B, \mathit{lfp}^X\ F^X) \circ F^{k_2} \circ (id^B, \mathit{lfp}^X\ F^X) \circ F^{k_1}(\bot) \sqsubseteq \mathit{lfp}\ F$
where k_n is the number of iterations in the n^{th} phase (1).
Thus, we have $((id^B, \mathit{lfp}^X\ F^X) \circ F^{k_n})^n(\bot) \sqsubseteq \mathit{lfp}\ F$ for $n \geqslant 0$.

Hence, we conclude from $\mathit{lfp}((id^B, \mathit{lfp}^X\ F^X) \circ (\mathit{lfp}^B\ F) \circ F) \sqsubseteq \mathit{lfp}\ F$ and Thm. 9.2 that our algorithm computes the least fixed point, *i.e.*, it is optimal.

9.2.4 Discussion

An important observation is that, since the overall abstract domain is of the form $L \to \wp(\mathbb{B}^p) \times \overline{\mathbb{R}}^m$, the choice of the CFG has two effects on the performance: first, it determines the set of representable abstract properties, and second, it influences the approximations made in the generation of the numerical equation system for the max-strategy iteration phase (2), because there is only one template polyhedron per location.

Generalization. The structure of the algorithm we presented is quite general. In particular, it does not depend on the method used to compute the numerical least fixed point in phase (2). We conjecture that the algorithm makes every method, that is able to compute the least fixed point of a numerical system by ascending iterations, compute the least fixed point of a logico-numerical system.

For example, we suppose that our algorithm can be used without any modification with the variant of max-strategy iteration for quadratic programs and quadratic templates proposed in [GS10].

If a method computes the fixed point by descending iterations, as for example min-strategy iteration [CGG+05, GGTZ07], our algorithm can still be used, but requires a small adaptation because the abstract value computed in phase (1), which is an under-approximation of the least fixed point, cannot be used to initialize phase (2), which requires an over-approximation: hence, line 10 in Fig. 9.4 must be replaced by

guessing appropriate initial bounds and an initial strategy for phase (2). This makes the algorithm less elegant and the analysis, probably, less efficient.

Logico-numerical max-strategy iteration using a power domain. The algorithm is also rather generic w.r.t. the kind of logico-numerical abstract domain we use. For example, we could consider the logico-numerical power domain $\mathbb{B}^p \to \wp(\mathbb{R}^n)$ (cf. 7.2.3) where $\wp(\mathbb{R}^n)$ is abstracted by any domain that is supported by strategy iteration. Then, the overall domain for our method is $L \to \mathbb{B}^p \to \overline{\mathbb{R}}^m$. This domain implicitly dynamically partitions each location into sub-locations corresponding to Boolean valuations sharing a common numerical abstract value. The construction of the equation system (generate in our algorithm, Fig. 9.4) must take into account these partitions.

This domain is more precise than the product domain described in §9.2, however, the drawback is that the number of partitions might explode if only few Boolean valuations share a common numerical abstract value.

Comparison with logico-numerical min-strategy iteration. The power domain $\mathbb{B}^p \to \overline{\mathbb{R}}^m$ is also used by Sotin et al [SJVG11] who propose an approach to analyzing logico-numerical programs using min-strategies. In accordance with the form of the abstract domain, they consider logico-numerical strategies $\mathbb{B}^p \to \Pi$ (where Π is the set of min-strategies), which dynamically associates the numerical min-strategies to the reachable Boolean states during analysis. They start with an initial logico-numerical strategy $P^{(0)} = \lambda \boldsymbol{b}.\pi^{(0)}$ with a chosen numerical min-strategy $\pi^{(0)}$ and compute a fixed point using logico-numerical Kleene iteration with widening and descending iterations. Then they iteratively improve the min-strategies in $P^{(i)}$ and recompute the fixed point.

This approach does not integrate well with mathematical programming because the only known method for computing the fixed point of a logico-numerical strategy is logico-numerical Kleene iteration (with widening). Hence, in contrast to our approach, there is no guarantee to compute the least fixed point.

Comparison with abstract acceleration. Numerous methods have been developed to alleviate the problem of bad extrapolations due to widening, *e.g.*, abstract acceleration (see §§4–6 and 8), a method for computing the transitive closure of numerical loops. These methods are able to accelerate some cases of self-loops and cycles with certain types of affine transformations, and they rely on widening in the general case. However, due to the use of general convex polyhedra, they are able to "discover" complex invariant constraints.

In contrast, max-strategy iteration is able to "accelerate" globally the whole transition system regardless of the graph structure or type of affine transformation, and it effectively computes the least fixed point. However, this is only possible on the simpler domain of template polyhedra.

Although the use of template polyhedra is a restriction, this kind of (static) approximation is much more predictable than the (dynamic) approximations made by widening.

Remark 9.2 Guided static analysis *[GR07] is a framework for analyzing monotonically increasing subsystems, which makes it possible to reduce the impact of widening by applying descending iterations "in the middle" of an analysis. Our algorithm proceeds in a similar fashion – although for different technical reasons – by applying max-strategy iteration on monotonically increasing subsystems.*

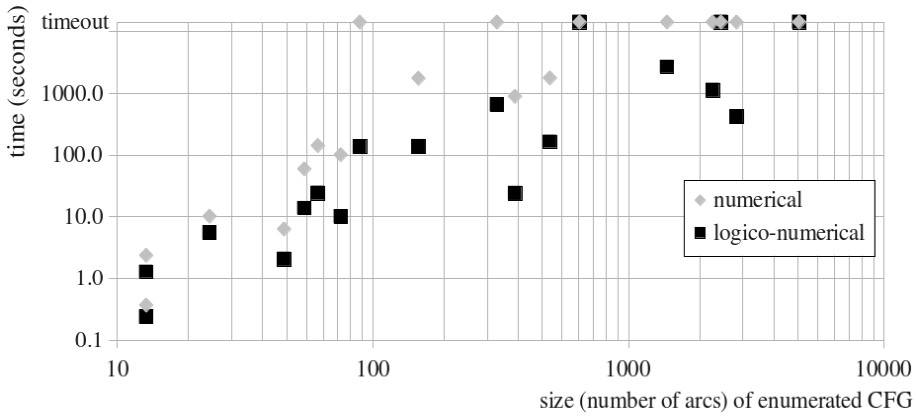


Figure 9.6: Scalability of *logico-numerical* max-strategy iteration in comparison with *numerical* max-strategy iteration on the enumerated CFG, using octagonal constraints. The timeout was set to 3600 seconds. Note the logarithmic scales.

Alternative approach handling Booleans as integers. A naive approach to treating Booleans is to encode them as integers $\in \{0,1\}^p$. The advantage is that max-strategy iteration can be used "as is" by adding template constraints for those Booleans. Yet, such an analysis will yield very rough approximations because templates can only describe convex sets, whereas Boolean state sets are usually highly non-convex.

9.3 Experimental Evaluation

We implemented our method in our tool ReaVer (§14) based on the logico-numerical abstract domain library BddApron [Jea] and the max-strategy iteration solver of Gawlitza et al [DG11a]. Since template polyhedra are not yet implemented in the Apron library [JM09], we emulated template polyhedra operations in phase (1) with the help of general polyhedra, which certainly impaired the efficiency – nonetheless we obtained encouraging results.

Benchmarks. We took 18 of the benchmarks used in §8.4, which are Lustre programs of up to a few hundred lines of code, 27 Boolean and 7 numerical variables, which produce enumerated CFGs of up to 650 locations and 5000 transitions after simplification by Boolean reachability. The focus of the experiments was on comparing the precision of the inferred invariants rather than proving properties.

Results. We performed experiments with octagonal constraints ($\pm x_i$, $x_i \pm x_j$) in order to evaluate efficiency and precision. We compared max-strategy iteration on the enumerated CFGs (MSI) with logico-numerical max-strategy iteration (LNMSI) on CFGs obtained by the static partitioning method by "numerical modes" described in §8.3. The resulting CFGs are on average five times smaller than the enumerated CFGs for the medium-sized benchmarks.

– LNMSI scales clearly better than MSI (see Fig. 9.6): our method was on average 9 times faster – for those benchmarks where both methods terminated before the timeout: MSI hit the timeout in 8 out of 18 cases (versus 3 for our method). The gain in efficiency grows with increasing benchmark sizes.

– The precision is almost preserved: only 0.38% (!) of the bounds were worse but still finite, and 0.16% were lost. This precision loss did not impact the number of proved properties. Due to the better scalability we were even able to prove 3 more benchmarks (10 as opposed to 7).
– We compared the precision of LNMSI with octagonal constraints with a logico-numerical analysis with octagons using the standard approach with widening ($N=2$) and 2 descending iterations on the same CFG. 18% of the bounds of our invariants were strictly better than those computed using the standard analysis. In two cases, these improvements made the difference to prove the property. However, the standard analysis was 19 times faster on average.

Furthermore, we experimented with different templates and CFG sizes:

– Thegain in speed increases with the template size: 3.3 for interval analysis ($\pm x_i$), 5 for zones ($\pm x_i$, $x_i - x_j$) and 9 for octagons (for those benchmarks which did not run into timeouts).
– The precision of LNMSI depends on the CFG size: the general trend is "the bigger the more precise", but the results are less clear: CFGs of the same size seem to have very different *quality* w.r.t. precision. Partitioning methods that find *good* partitions matter!
– A smaller CFG does not automatically mean faster analysis: the fact that a smaller graph means more complicated logico-numerical transition functions and more numerical strategies per location outweighs the advantage of dealing with less locations.
– It is interesting that LNMSI scales also better on the enumerated CFG: it seems to be advantageous to start with a small system with few strategies, iteratively increase the system, and finally, when computing the numerical fixed point of the full system, most of the strategies are already known not to improve the bounds, and thus max-strategy iteration converges faster.

We also experimented with LNMSI using the logico-numerical power domain discussed in §9.2.4, which performed on our CFGs still 6 to 7 times faster and with a 100% preservation of bounds compared to MSI.

9.4 Conclusions

We presented *logico-numerical max-strategy iteration*, a solution to the intricate problem of combining numerical max-strategy iteration with techniques that are able to deal with Boolean variables implicitly and therefore allow to trade off precision for efficiency.

In contrast to the previous attempt of Sotin et al [SJVG11] of extending strategy iteration to logico-numerical programs, which relies on widening operators to converge, our method enables the use of mathematical programming and hence, it indeed computes the *best logico-numerical invariant* w.r.t. the chosen abstract domain.

The effectiveness of our method depends on two factors:

(1) The choice of the templates: in our experiments, we used mainly octagonal constraints, but we could have used methods for inferring template constraints, *e.g.*, [SSM05].
(2) The considered CFG (either of the imperative program, or the one obtained by partitioning in the case of data-flow programs) which determines the abstract domain: the partitioning method by "numerical modes" turned out to be surprisingly effective: compared to the solution based on an enumeration of the reachable Boolean

states, the obtained CFGs were 5 times smaller on average, still the precision loss was negligible, *i.e.*, almost zero, and we gained at least one order of magnitude w.r.t. efficiency.

Furthermore, we deliver the first experimental results of applying *numerical* max-strategy iteration to larger programs: on the one hand max-strategy iteration is guaranteed to compute more precise invariants than standard techniques in the same domain; on the other hand our implementation is not (yet) able to compete with standard techniques w.r.t. efficiency.

Perspectives. Our algorithm is quite generic w.r.t. the numerical analysis method and logico-numerical abstract domain. Yet, in order to tackle efficiency issues evoked above, it would be interesting to design a more integrated logico-numerical max-strategy solver. This would enable us to share more information between subsequent calls to the max-strategy iteration, *e.g.*, to avoid the retesting of strategies that will definitely not lead to an improvement. Beyond that, we could more extensively use SMT-solvers. For instance, checking whether a strategy is an improvement is currently done after having constructed the numerical equation system; it would be beneficial to find the improving strategies already on the logico-numerical level.

We will apply the method presented in this chapter to the analysis of logico-numerical hybrid automata (see §11.2) by extending the hybrid max-strategy iteration method of Dang and Gawlitza [DG11b, DG11a] (see also §12.2.2).

Part III

Modeling and Verification of Hybrid Systems

Chapter 10

Hybrid System Modeling

This chapter gives on overview of concepts and languages for hybrid system modeling.

As explained in the introduction, an embedded system (Fig. 1.1) consists of a computer system interacting with its physical environment – in terms of control theory: a (discrete-time) *controller* and a (continuous-time) *plant* – which form together a hybrid system. The plant is modeled using differential equations (the physical laws) (§10.1), whereas the controller is a computer program represented by a discrete transition system. The prevalent model for hybrid system verification is the *hybrid automaton* (§10.2), which combines these models.

Numerical simulation languages and tools (§10.3), *e.g.*, SIMULINK, are most widely used for hybrid systems modeling. They provide an integrated environment supporting simulation, code generation and test automation. However, their semantics has some peculiarities.

The hybrid synchronous programming language ZELUS (§10.4) tackles these semantical issues. Its semantics is based on non-standard analysis, which allows to specify a deterministic hybrid system semantics independently of numerical integration issues. Moreover, it integrates elegantly with synchronous languages.

10.1 Dynamical Systems

We recall a few concepts from continuous dynamical systems. For further details we refer to textbooks, *e.g.*, [SB02].

Ordinary Differential Equations. An ordinary differential equation (ODE) of order n is of the form

$$\frac{d^n x}{dt^n} = F\left(t, x, \frac{dx}{dt}, \frac{d^2x}{dt^2}, \ldots, \frac{d^{n-1}x}{dt^{n-1}}\right)$$

with $x \in \mathbb{R}, t \in \mathbb{R}^{\geqslant 0}$. An ODE of order n can be written as a system of n ODEs of first order:

$$\begin{cases} x_1 = x \\ \frac{dx_k}{dt} = x_{k+1} \text{ for } k \in [1, n-1] \\ \frac{dx_n}{dt} = F(t, x_1, x_2, \ldots, x_n) \end{cases} \qquad \text{in vector notation: } \dot{\boldsymbol{x}}(t) = F(t, \boldsymbol{x}(t))$$

The existence and uniqueness of a solution to an ODE depends on the continuity of F:

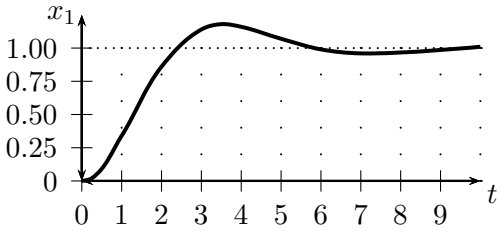


Figure 10.1: Trajectory of the solution for $x_1(t)$ of the LTI system in Ex. 10.1

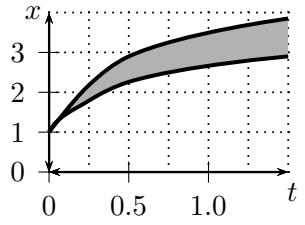


Figure 10.2: Set of solutions of the differential inclusion in Ex. 10.2

Definition 10.1 (Lipschitz continuous) *A function f is* Lipschitz continuous *iff*

$$\exists K \in \mathbb{R}^{\geqslant 0} : \forall \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^n : ||f(\boldsymbol{x}_1) - F(\boldsymbol{x}_2)|| \leqslant K \cdot ||\boldsymbol{x}_1 - \boldsymbol{x}_2||$$

Theorem 10.1 (Picard-Lindelöf) *If F is Lipschitz continuous, then $\dot{\boldsymbol{x}}(t) = F(t, \boldsymbol{x}(t))$ has a unique solution $\boldsymbol{x}(t)$ for a given initial condiiton $\boldsymbol{x}(0) = \boldsymbol{x}_0$.*

Linear time-invariant systems. A system of ODEs is *time-invariant* if F does not depend on t: $\dot{\boldsymbol{x}}(t) = F(\boldsymbol{x}(t))$. The function F can be viewed as a constant vector field $\mathbb{R}^n \to \mathbb{R}^n$, which maps each point $\boldsymbol{x}$ to its derivative vector $\dot{\boldsymbol{x}}$.

Definition 10.2 (Linear time-invariant system) *A linear time-invariant (LTI) system is characterized by the* linear ODE system with constant coefficients

$$\dot{\boldsymbol{x}}(t) = \mathbf{A}\boldsymbol{x}(t) + \mathbf{B}\boldsymbol{\xi}(t)$$

with state variables $\boldsymbol{x} \in \mathbb{R}^n$, input variables $\boldsymbol{\xi} \in \mathbb{R}^m$ and matrices $\mathbf{A} \in \mathbb{R}^{n \times n}, \mathbf{B} \in \mathbb{R}^{n \times m}$.

The solution of such a system for the initial conditions $\boldsymbol{x}(0) = \boldsymbol{x}_0$ is given by:

$$\boldsymbol{x}(t) = e^{\mathbf{A}t}\boldsymbol{x}_0 + \int_0^t e^{\mathbf{A}(t-t')}\mathbf{B}\boldsymbol{\xi}(t')dt' \tag{10.1}$$

Example 10.1 (LTI system) *(see Fig. 10.1)*

$$\begin{cases} x_1(0) = 0 & x_2(0) = 0 \\ \xi(t) = 1 \\ \dot{x_1}(t) = x_2(t) \dot{x_2}(t) = \xi(t) - x_1(t) - \frac{1}{2}x_2(t) \end{cases}$$

Differential inclusions. A differential inclusion is given by a relation $\dot{\boldsymbol{x}}(t) \in F(t, \boldsymbol{x})$. For time-invariant systems we denote this relation $V(\boldsymbol{x}, \dot{\boldsymbol{x}})$. A linear differential inclusion with constant coefficients is defined by $\mathbf{A}\boldsymbol{x}(t) + \mathbf{C}\dot{\boldsymbol{x}}(t) \leqslant \boldsymbol{d}$, *i.e.*, $V(\boldsymbol{x}, \dot{\boldsymbol{x}})$ is a convex polyhedron.

The set of solutions of the ODE system $\dot{\boldsymbol{x}} = F(\boldsymbol{x}, \boldsymbol{\xi})$ generated by all inputs $\boldsymbol{\xi}$ satisfying a predicate $\mathcal{A}(\boldsymbol{x}, \boldsymbol{\xi})$ and the initial conditions $\boldsymbol{x}(0) \in \mathcal{I}$ equals the set of solutions of the linear differential inclusion

$$V(\boldsymbol{x}, \dot{\boldsymbol{x}}) = \exists \boldsymbol{\xi} : (\dot{\boldsymbol{x}} = F(\boldsymbol{x}, \boldsymbol{\xi})) \wedge \mathcal{A}(\boldsymbol{x}, \boldsymbol{\xi})$$

with initial conditions $\boldsymbol{x}(0) \in \mathcal{I}$.

Example 10.2 (Differential inclusions) *(see Fig. 10.2)*

$$\left.\begin{array}{l} x(0)=1 \\ \forall t \geqslant 0 : 3 \leqslant \xi(t) \leqslant 4 \\ \dot{x}(t) = x(t)+\xi(t) \end{array}\right\} \Longleftrightarrow \left\{\begin{array}{l} x(0)=1 \\ 3 \leqslant \dot{x}(t)-x(t) \leqslant 4 \end{array}\right.$$

10.2 Hybrid Automata

Hybrid automata [ACHH92, ACH+95, Hen96, HHWT97] are a well-established formalism for specifying hybrid systems in the context of verification.

Definition 10.3 *A hybrid automaton (HA) is a directed graph defined by* $\langle L, F, J, \Sigma^0 \rangle$ *where*
- L *is the finite set of locations;*
- $F : L \to \mathcal{V}$ *maps a flow relation (a differential inclusion)* $V(\boldsymbol{x}, \dot{\boldsymbol{x}}) \in \mathcal{V}$ *to each stat;*
- $J \subseteq L \times \mathcal{R} \times L$ *defines a finite set of arcs between locations with the discrete transition relation* $R(\boldsymbol{x}, \boldsymbol{x}') \in \mathcal{R}$ *over the state variables* $\boldsymbol{x}$*; and*
- $\Sigma^0 : L \to \mathcal{X}$ *maps to each state the set of initial states* $X^0 \in \mathcal{X}$*, which satisfy the condition* $\forall \ell : \forall \boldsymbol{x} : \Sigma^0(\ell)(\boldsymbol{x}) \Rightarrow \exists \dot{\boldsymbol{x}} : F(\ell)(\boldsymbol{x}, \dot{\boldsymbol{x}})$.

Further notations:
- $C_\ell(\boldsymbol{x}) = \exists \dot{\boldsymbol{x}} : V_\ell(\boldsymbol{x}, \dot{\boldsymbol{x}})$ is the *staying condition* of the flow $V_\ell = F(\ell)$.
- $G_{\ell,\ell'}(\boldsymbol{x}) = \exists \boldsymbol{x}' : R(\boldsymbol{x}, \boldsymbol{x}')$ is the *guard* of the arc $(\ell, R, \ell') \in J$.

Semantics. We use the following definitions: Let $T_{[0,\delta]}$ be the set of differentiable trajectories $\tau : [0, \delta] \to \mathbb{R}^n$. The function flow_{V_ℓ} returns the set of end states of trajectories τ starting in the given state $\boldsymbol{x}$ and that obey the flow relation V_ℓ:

$$\mathit{flow}_{V_\ell}(\boldsymbol{x}) = \left\{ \boldsymbol{x}' \;\middle|\; \begin{array}{l} \exists \delta > 0, \exists \tau \in T_{[0,\delta]} : \\ \quad \tau(0) = \boldsymbol{x} \;\wedge\; \tau(\delta) = \boldsymbol{x}' \;\wedge \\ \quad \forall \delta' \in [0, \delta] : C_\ell(\tau(\delta')) \;\wedge \\ \quad \forall \delta' \in (0, \delta) : V_\ell(\tau(\delta'), \dot{\tau}(\delta')) \end{array} \right\}$$

Definition 10.4 (Semantics) *An execution of a hybrid automaton is a (possibly) infinite trace* $(\ell_0, \boldsymbol{x}_0) \to (\ell_1, \boldsymbol{x}_1) \to (\ell_2, \boldsymbol{x}_2) \to \ldots$ *with* $\to = \to_c \cup \to_d$ *and*

$$\begin{array}{lcl} (\ell, \boldsymbol{x}) \to_c (\ell', \boldsymbol{x}') & \Leftrightarrow & \ell = \ell' \wedge V_\ell = F(\ell) \wedge \boldsymbol{x}' \in \mathit{flow}_{V_\ell}(\boldsymbol{x}) \\ (\ell, \boldsymbol{x}) \to_d (\ell', \boldsymbol{x}') & \Leftrightarrow & \exists (\ell, R, \ell') \in J : R(\boldsymbol{x}, \boldsymbol{x}') \wedge C_{\ell'}(\boldsymbol{x}') \end{array}$$

Fig. 10.3 depicts a hybrid automaton and Fig. 10.4 shows some of its possible executions.

Note that the concrete semantics of hybrid automata exhibit three kinds of non-determinism:
- Non-determinism w.r.t. *flow* transitions, *i.e.*, the choice between different continuous evolutions compatible with the differential inclusions V.
- Non-determinism w.r.t. *flow and jump* transitions: The choice between flow and jump transitions due to an overlapping of staying condition and guards.
- Non-determinism w.r.t. *jump* transitions, which is the choice between several jump transitions.

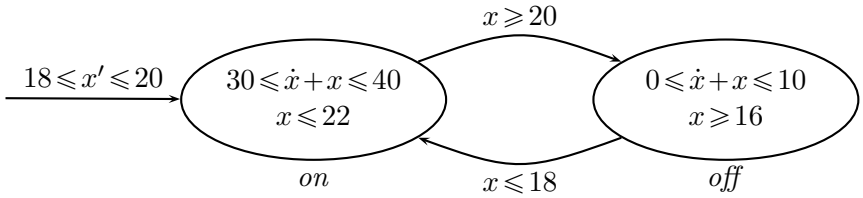


Figure 10.3: Hybrid automaton: Thermostat example (cf. [ACH+95])

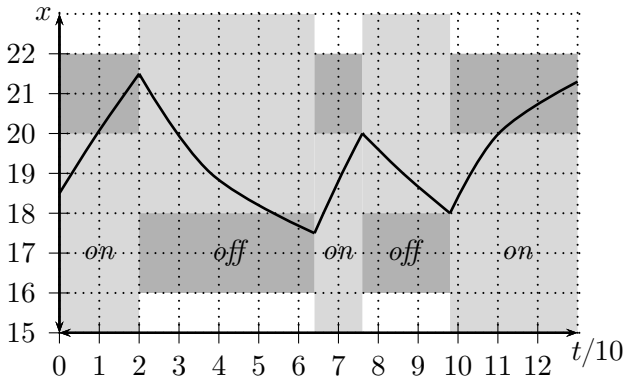


Figure 10.4: Example execution of the hybrid automaton in Fig. 10.3: staying conditions (whole shaded area) and intersection of staying and guards (dark gray).

Verification. We will discuss the verification of hybrid automata in §12.

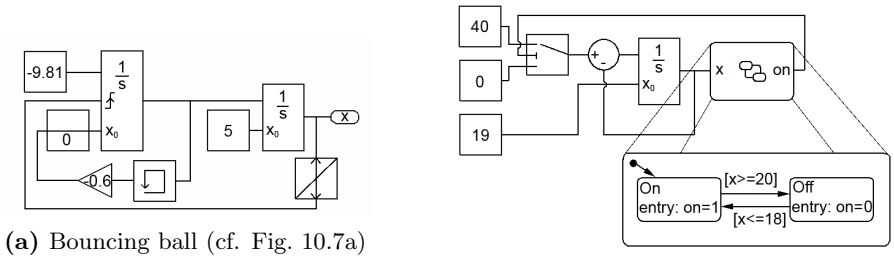
Modeling languages. Hybrid automata are a very low-level formalism and it is cumbersome to use them as a specification or programming language for large systems. However, they can be decomposed into the synchronous product of several hybrid automata in order to enable modular specification in form of hierarchical networks of communicating hybrid automata. This idea is used for example in the languages/tools SHIFT[1], HYVISUAL[2] [LZ05] (based on the PTOLEMY II framework) and CHARON[3] [ADE+03]. They offer tools for simulation; CHARON has also verification support.

Conceptually close to these languages are hybrid process calculi, like HYBRID CC[4] ("concurrent constraints") [GJS95], φ-calculus [RS03], and HYBRID CHI[5] [vBMR+06]. These languages generally support simulation. HYBRID CHI also provides a translation of subsets of the language to input formats for hybrid and timed automata verification tools, *e.g.*, UPPAAL [BLL+95].

10.3 Simulation Languages

Although the hybrid automaton model is well-suited for verification, the most widely used modeling languages for hybrid systems are those provided by numerical simulation tools like SIMULINK [Sim]. They offer features like modularity, hierarchy and a dataflow or equational syntax. We shortly describe some languages and tools and sketch the

[1] `http://path.berkeley.edu/SHIFT/`
[2] `http://ptolemy.eecs.berkeley.edu/hyvisual/`
[3] `http://rtg.cis.upenn.edu/mobies/charon/`
[4] `http://xenon.stanford.edu/~vgupta/hcc/hcc.html`
[5] `http://se.wtb.tue.nl/sewiki/chi/start`



(a) Bouncing ball (cf. Fig. 10.7a)

(b) Thermostat system

Figure 10.5: Hybrid system modeling with SIMULINK/STATEFLOW

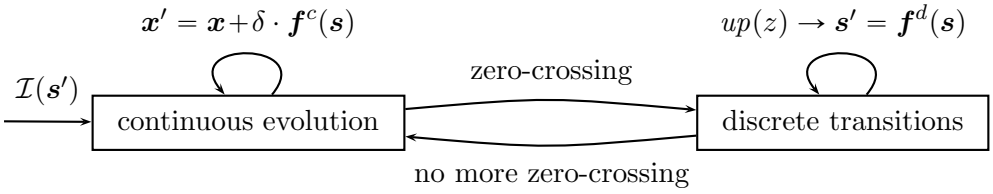


Figure 10.6: Execution scheme of a numerical simulator

mode of operation of such simulators.

At last we will discuss some odd phenomena occurring in hybrid systems w.r.t. discrete events and time: there might be an infinite number of discrete transitions in a finite time interval.

Languages

SIMULINK [Sim] is an integrated tool platform for modeling, simulation, code-generation and test automation of hybrid systems. It has a graphical data-flow-based input language for specifying the continuous and discrete behavior (see Fig. 10.5a). Additionally the STATEFLOW extension enables automata-based specification of the discrete behavior (rightmost block in Fig. 10.5b).

SCICOS[6] has features similar to SIMULINK.

MODELICA[7] [FE98, OEM99] is a standardized language. Its particularity is that it supports equation-based specifications of continuous behavior (differential algebraic equations $f(\boldsymbol{x}, \dot{\boldsymbol{x}}) = 0$), which are common in physics, *e.g.*, for conservation laws like Kirchhoff's laws. There are several commercial and open source simulator implementations.

ZELUS [BCP10, BBCP12, BBCP11a, BBCP11b] (see §10.4) is a recent academic hybrid synchronous data-flow language for modeling, simulation, code-generation and verification (see §11).

Simulators

The simulation of such hybrid models uses numerical ODE solvers for handling continuous evolution. Discrete execution steps that interrupt the continuous-time evolution are

[6] `http://www.scicos.org`
[7] `http://modelica.org`

triggered by the activation of *zero-crossings*. A zero-crossing $up(z)$ is an event occurring during the integration of an ODE system $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}^c(\boldsymbol{x}(t))$, when some expression $z(\boldsymbol{x}(t))$ changes sign from negative to positive. A zero-crossing may also be triggered by a discrete execution step.

Such a simulator works as follows (Fig. 10.6): At start, the main simulation loop initializes the ODE solver with an initial state $\boldsymbol{x}_0$, a system of ODEs $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}^c(\boldsymbol{x}(t))$, and a finite set of zero-crossing expressions z_j. Then the ODE solver integrates using the specified integration method – in Fig. 10.6 an Euler scheme is indicated – until at least one of the zero-crossings is activated. When this happens, the control is given back to the main simulation loop, which executes one or several discrete execution steps $\boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s})$ before reinitializing the ODE solver and continuing the integration.

For the detection of a zero-crossing, ODE solvers usually have to backtrack and decrease the integration step in order to accurately approximate the point of zero-crossing. Since this is quite expensive and slows down simulation, *e.g.*, SIMULINK/STATEFLOW offers the option to execute discrete actions without zero-crossing detection, *i.e.*, the discrete equations $\boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s})$ are evaluated at every integration step. However, this can easily result in unpredictable behavior.

Peculiar behaviors of hybrid systems

The plant model is often a hybrid system itself because of simplifications and idealizations: for example, fast behaviors like opening of a valve or highly non-linear behavior like the one of diodes are often modeled as discontinuities (jumps), *i.e.*, discrete transitions. These (over-)simplifications of reality, but also the interaction with the discrete controller, can result in peculiar behaviors.

Zeno behavior. Fig. 10.7 shows the model of a ball that bounces on a surface losing energy: the time interval between the jump transitions decreases to zero. The sum of the time intervals has a limit (the so-called "Zeno point"): time approaches this point, but it cannot advance beyond it. Hence, there is an infinite number of discrete transitions in a finite time interval.

Numerical simulators face difficulties in such "chattering" situations, because the number of events (zero-crossings) explodes and the simulation advances very slowly. Usually, the simulator issues a warning when a certain number of events per time interval is exceeded. Moreover, simulation results are very unpredictible due to numerical errors and the simulation might erroneously continue beyond the Zeno point.

There are methods for "regularizing" and extending certain types of systems beyond the Zeno point [ZLA06], but there is no general theory. In practice, Zeno behavior is avoided in design by introducing a threshold below which the system jumps to the asympotic value.

Sliding modes. Chattering along a surface can also arise without time being blocked, like in the following example:

Example 10.3 (Chattering) *Assume a system where initially, $x(0) = -1$ and $\dot{x}(t) = 1$; then, every time when x becomes strictly greater than zero ($up(x(t))$, x continues evolving according to the dynamics $\dot{x}(t) = -1$, and every time when x becomes strictly less than zero $up(-x(t))$, x continues with the dynamics $\dot{x} = 1$. The resulting trajectory is illustrated in Fig. 10.8a.*

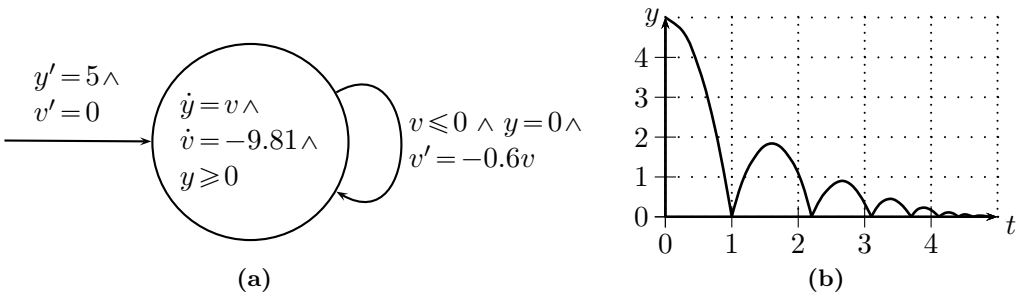


Figure 10.7: Bouncing ball: hybrid automaton (a) and zeno behavior (b)

(a) Chattering trajectory **(b)** Equivalent sliding mode

Figure 10.8: Sliding modes

Such infinitely fast switching between modes is the principle of *sliding mode control*, used for non-linear and uncertain systems, *e.g.*, anti-lock braking systems. The goal is to control a system by chattering along the desired trajectory. This trajectory lying "in" the intersection of the switching surfaces is called the *sliding mode* (Fig. 10.8b). For certain cases of physical systems, the sliding mode behavior can be computed in closed form [Fil60, Utk92, AB08].

In some applications, for example, if Ex. 10.3 represented a thermostat system, it would not be desirable to incessantly switch on and off the gas boiler. In such cases one introduces two switching thresholds, *i.e.*, *hysteresis*, in the model (cf. Fig. 10.3).

Remark 10.1 *Chattering in the context of sliding modes cannot be exactly modeled in hybrid automata. We will discuss this issue in §11.1.*

10.4 Zelus – A Hybrid Synchronous Language

Embedded controllers interact tightly with the plant. Hence, for validation purposes, it is propitious to integrate a plant model into the development process and consider the whole system. For this reason, simulation platforms, like Simulink, have become very popular in industry, because they support such an approach.

However, the semantics of these hybrid simulators is not so well-understood: Usually the semantics is induced by the simulation engine and the multitude of its parameters like fixed or variable integration methods, zero-crossing detection, thresholds and step sizes. Additionally numerical issues due to floating point arithmetics play an important rule. Besides that, there are also strange semantical choices: *e.g.*, in Simulink the evaluation order of the block elements in a diagram depends on their graphical position. All these issues sometimes result in hardly reproducible simulation results with limited trustworthiness.

$\langle decl \rangle$ ::= $\langle typedecl \rangle$ | $\langle fundecl \rangle$ | $\langle decl \rangle$ $\langle decl \rangle$

$\langle typedecl \rangle$ ::= `type` t `=` L `|` ... `|` L

$\langle fundecl \rangle$ ::= `let` [`node` | `hybrid`] f [$\langle pat \rangle$] `=` $\langle expr \rangle$

$\langle pat \rangle$ ::= v | (`(`$\langle pat \rangle$`,` ... `,`$\langle pat \rangle$`)`)

$\langle expr \rangle$ ::= v | cst | op $\langle expr \rangle$ | f $\langle expr \rangle$ | (`(`$\langle expr \rangle$`,` ... `,`$\langle expr \rangle$`)`)
| $\langle expr \rangle$ `fby` $\langle expr \rangle$ | $\langle expr \rangle$ `->` $\langle expr \rangle$ | `pre` $\langle expr \rangle$ | `last` v | `up` $\langle expr \rangle$ | `init`
| $\langle expr \rangle$ `on` $\langle expr \rangle$ | `let` [`rec`] $\langle equ \rangle$ `in` $\langle expr \rangle$

$\langle equ \rangle$ ::= v `=` $\langle expr \rangle$ | $\langle equ \rangle$ `and` $\langle equ \rangle$
| `der` v `=` $\langle expr \rangle$ `init` $\langle expr \rangle$ `reset` $\langle res \rangle$
| v `=` $\langle res \rangle$ `init` $\langle expr \rangle$
| `automaton` (S [$\langle pat \rangle$] `->` $\langle loc \rangle$ `unless` $\langle trans \rangle$ `|` ... `|` $\langle trans \rangle$ `done`)$^+$

$\langle res \rangle$ ::= $\langle expr \rangle$ `every` $\langle expr \rangle$ `|` ... `|` $\langle expr \rangle$ `every` $\langle expr \rangle$

$\langle loc \rangle$ ::= `local` v `in` $\langle loc \rangle$ | `do` $\langle equ \rangle$ `until` $\langle trans \rangle$ `|` ... `|` $\langle trans \rangle$

$\langle trans \rangle$ ::= $\langle expr \rangle$ (`then` | `continue`) S

Table 10.1: Zelus syntax (subset).

There are several attempts of designing languages that reconcile the requirements for an easy-to-use all-in-one platform for programming, simulation, and verification of an embedded (hybrid) system, *e.g.*, Ptolemy II [LZ05] or Zelus [BCP10, BBCP12, BBCP11a, BBCP11b]. The latter combines the synchronous programming language Lucid Synchrone (§2.2) with ODEs.

We present the Zelus language in §10.4.1. Interesting is the aspect that the semantics (§10.4.3) is based on non-standard analysis (§10.4.2), which allows an "ideal" discretization of the evolution of continuous variables. Thus, it provides an elegant formal definition of the semantics of hybrid systems, which is independent of numerical integration methods.

Actually, non-standard analysis has already been employed by Iwasaki et al [IFS+95] in the context of hybrid systems. Later, Bliudze et al [BK09, Bli06] have proposed non-standard analysis as a semantic domain for specifiying and analyzing hybrid systems. These ideas have then influenced the development of Zelus as well as the imperative language While^{dt} [SH11].

10.4.1 Syntax of Zelus

Zelus is a language in development and hence undergoes an evolution. We present here roughly the (stable) subset of the language presented in [BBCP11b]. The syntax rules are listed in Table 10.1).

Most of the constructs are similar to Lucid Synchrone (§2.2). We explain the additional concepts with the help of examples:

Example 10.4 (Zelus program) *The following program describes a ball that starts to move from the initial position* `x0` *with the initial speed* `v0` *as soon as* `start` *is enabled. The change of speed is governed by gravity. There is a floor at* $x = 0$ *from which it*

rebounces with inverted and reduced speed until the speed on hitting the floor falls below a threshold `eps`*. In this case the speed is set to* 0 *in order to avoid Zeno behavior.*

```
let hybrid bouncingball (x0,v0,start,eps) = x where
  rec der x = v init x0
  and der v = -9.81 init 0.0
              reset v0 every start
              | -0.6*(last v) every (up(-x))
              | 0.0 every ((up(-x)) on (-v<eps))
```

In addition to stateless functions and stateful, discrete `node`s, Zelus has also stateful, `hybrid` nodes. Only hybrid nodes may contain ODEs. `der x = v init x0` defines an ODE $\dot{x}(t) =$`v` with initial condition $x(0) =$`x0`.

Discontinuities (jumps) are defined by reset handlers: `v0 every start` states that `v` will be set to `v0` every time the zero-crossing `start` (here an input of the node) occurs. In the second reset handler, the zero-crossing is defined by `(up(-x))`, meaning that `v` will be set to `-0.6*v` every time `x` crosses zero from above. The third reset handler sets `v` to zero if `(up(-x))` is activated *and* the condition `(-v<eps)` is satisfied. If several zero-crossings are enabled simultaneously, then the first enabled one appearing in the list has priority.

Zelus has also hierarchical automata [BBCP11b] with the same features as in Lucid Synchrone (cf. §2.2). Ex. 10.4 can be reformulated using this construct:

Example 10.5 (Zelus program) *(cf. [BBCP11b])*

```
let hybrid bouncingball (x0,v0,start,eps) = x where
    rec init x = x0
    and automaton
        | Wait -> do der v = 0.0 until start then Bounce(v0) done
        | Bounce(v00) ->
            local z,v in
            do der v = -9.81 init v00
              and der x = v
              and z = up(-x)
            until (z on (-v<eps)) then Wait
                | z then Bounce(-0.6*v)
            done
        end
```

In this example, jumps are defined by location parameters (`v00` in `Bounce(v00)`) and entry by reset (`then`): for instance, when the zero-crossing `start` is activated in location `Wait`, then we switch to location `Bounce` where `v` is initialized with the value of `v0` given as parameter.

We will describe the semantics of Zelus in §10.4.3 after having introduced the basics of non-standard analysis which it is based on.

10.4.2 Introduction to Non-Standard Analysis

Non-standard analysis was proposed by Robinson [Rob96] in the 1960s to allow the explicit manipulation of *infinitesimals* in analysis. Robinson's formulation is an axiomatic

approach. We follow here the constructive formulation of Lindstrøm [Lin88], which defines non-standard numbers as equivalence classes of infinite sequences: this is similar to the definition of $\mathbb{R}$ as the set of equivalence classes of Cauchy sequences $\langle x_n \rangle$ over rational numbers $\mathbb{Q}^{\mathbb{N}}/\equiv$ with the equivalence relation $\langle x_n \rangle \equiv \langle y_n \rangle \Leftrightarrow \lim_{n\to\infty}(x_n - y_n) = 0$.

Non-standard real numbers. Non-standard real numbers ${}^\star\mathbb{R}$ shall extend $\mathbb{R}$ with infinitely large numbers and numbers infinitely close to zero (infinitesimals). Hence, an equivalence relation ${}^\star\equiv$ must be defined such that ${}^\star\mathbb{R} = \mathbb{R}^{\mathbb{N}}/{}^\star\equiv$.

For this purpose a finitely additive[8] measure μ over $\mathbb{N}$ is *fixed* with the following properties:
(1) $\mu : \wp(\mathbb{N}) \to \{0, 1\}$,
(2) $\mu(X) = 0$ for all finite sets X,
(3) $\mu(\mathbb{N}) = 1$.
This measure partitions $\wp(\mathbb{N})$ into two classes: "small" sets (which include all finite sets) with $\mu = 0$ and "big" sets with $\mu = 1$. The existence of such a measure is proved using Zorn's lemma (see [Lin88] for details).

Definition 10.5 (${}^\star\equiv$) *[Lin88]* $\langle x_n \rangle {}^\star\equiv \langle y_n \rangle$ *iff* $\mu\{n \mid x_n = y_n\} = 1$
i.e., $\langle x_n \rangle$ *equals* $\langle y_n \rangle$ *almost everywhere.*

Definition 10.6 (Infinitesimals and infinite numbers) *[Lin88]*

$$\partial \in {}^\star\mathbb{R} \text{ is infinitesimal if } \forall a \in \mathbb{R}^{>0} : -a < \partial < a$$
$$x \in {}^\star\mathbb{R} \text{ is infinite if } \neg\exists a \in \mathbb{R}^{>0} : -a < x < a$$

For example, $\langle \frac{1}{n} \rangle$ is an infinitesimal, because for any $a \in \mathbb{R}$ the set $\{n \mid -a < \frac{1}{n} < a\}$ is infinite and thus $\mu = 1$. Observe that 0 is the only infinitesimal number in $\mathbb{R}$. Conversely, $\langle n^2 \rangle$ and $\langle -n \rangle$, for instance, are positive and negative infinite numbers respectively.

The following proposition describes the relationship between finite real and non-standard real numbers:

Proposition 10.1 (Standard part) *[Lin88] Any* finite $x \in {}^\star\mathbb{R}$ *can be written uniquely as a sum* $x = a + \partial$ *where* $a \in \mathbb{R}$ *and* ∂ *is an infinitesimal.*

We denote $a = st(x)$ *the* standard part *of* x.

The real numbers are embedded within the non-standard reals: the *non-standard version* ${}^\star a$ of $a \in \mathbb{R}$ is the constant sequence $\langle a, a, a, \ldots \rangle$.

See Fig. 10.9 for an illustration of the real and non-standard number lines.

Sets, functions and the transfer principle. A non-standard set $X = \langle A_n \rangle$ can be constructed by a sequence of $A_n \in \mathbb{R}$ such that

$$\langle a_n \rangle \in \langle A_n \rangle \text{ iff } \mu\{n \mid a_n \in A_n\} = 1$$

Similarly, a non-standard function $\langle f_n \rangle : {}^\star\mathbb{R} \to {}^\star\mathbb{R}$ can be constructed by a sequence of functions $f_n : \mathbb{R} \to \mathbb{R}$ such that

$$\langle f_n \rangle(\langle x_n \rangle) = \langle f_n(x_n) \rangle$$

Sets and functions constructed in this way are called *internal*: the construction is applied componentwise to the elements of the sequence. This principle can be generalized to first-order formulas:

[8] $\mu(X \cup Y) = \mu(X) + \mu(Y)$ for $X \cap Y = \varnothing$

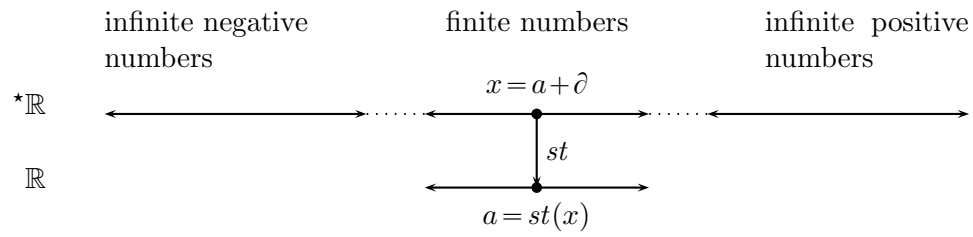


Figure 10.9: Number line of non-standard reals and their relation to standard reals (cf. [Lin88])

Theorem 10.2 (Transfer principle) *[Lin88] Let denote φ a first-order formula over $\mathbb{R}$, and $^\star\varphi$ its non-standard version,* i.e.*, where all symbols have been replaced by their non-standard versions. Then*

$$\langle A_n \rangle \models {}^\star\varphi \iff \mu\{n \mid A_n \models \varphi\} = 1$$

In other words, a first-order formula φ is true iff its non-standard version $^\star\varphi$ is true.

As a consequence all operations over subsets of $\mathbb{R}$ carry over to internal subsets of $^\star\mathbb{R}$, for instance:

$$\begin{array}{llcl} a+b & \langle a_n \rangle + \langle b_n \rangle & = & \langle a_n + b_n \rangle \\ a<b & \langle a_n \rangle < \langle b_n \rangle & \iff & \mu\{n \mid a_n < b_n\} = 1 \\ A \cap B & \langle A_n \rangle \cap \langle B_n \rangle & = & \langle A_n \cap B_n \rangle \\ \int_A f dx & \int_{\langle A_n \rangle} \langle f_n \rangle dx & = & \langle \int_{A_n} f_n dx \rangle \end{array}$$

Hyperfinite sets and non-standard calculus. In the same way as $^\star\mathbb{R}$, we can define *non-standard natural numbers* $^\star\mathbb{N} = \mathbb{N}/^\star\equiv$ that extend $\mathbb{N}$ by infinite natural numbers represented by sequences $\langle a_n \rangle$ of natural numbers a_n of which the limit is infinity.

Definition 10.7 (Hyperfinite set) *[Lin88] An internal set $X = \langle A_n \rangle$ is called* hyperfinite *if $\mu\{n \mid A_n$ is finite$\} = 1$.*

A hyperfinite set is an infinite set with all the combinatorial structure of finite sets: in particular, it has a smallest and greatest element and all other elements have a unique predecessor and a unique successor.

Let us consider the hyperfinite set $T = \langle T_n \rangle$ and an infinite number $N = \langle N_n \rangle \in {}^\star\mathbb{N}$:

$$T_n = \left\{ 0, \frac{1}{N_n}, \frac{2}{N_n}, \frac{3}{N_n}, \ldots \frac{N_n - 1}{N_n}, 1 \right\}$$

and a continuous function $f : \mathbb{R} \to \mathbb{R}$: we compute the sum

$$\sum_{t \in T} \frac{1}{N} {}^\star f(t) = \left\langle \sum_{t \in T_n} \frac{1}{N_n} f(t_n) \right\rangle$$

Since $\frac{1}{N_n}$ converges towards 0, the right-hand side above converges to the Riemann integral $\int_0^1 f(t)dt$:

$$\int_0^1 f(t)dt = st\left(\sum_{t \in T} \frac{1}{N} {}^\star f(t) \right)$$

Hence, the Riemann integral in $\mathbb{R}$ is the standard part of a hyperfinite sum.

Now, consider the following standard initial value problem:

$$\dot{a} = f(a,t), \quad a(0) = a_0$$

Assume that the ODE has a solution $a : \mathbb{R} \to \mathbb{R}$. We rewrite the ODE in its equivalent integral form $a(t) = a_0 + \int_0^t f(a(\tau),\tau)d\tau$ which equals the hyperfinite sum

$$a(t) = st\left({}^\star a_0 + \sum_{t \in T} \frac{1}{N} {}^\star f({}^\star a(t), t) \right)$$

By substituting $\partial = 1/N$, we get $T = \{t_n = n\partial \mid n = 0, \ldots, N\}$ and we can write the solution $a(t)$ of the initial value problem for $n = 0, \ldots, N$ as the standard part $st(x)(t)$ of the solution of an equivalent non-standard "hyperdiscrete" dynamical system

$$\begin{array}{lcl} x(t_0) & = & {}^\star a_0 \\ x(t_{n+1}) & = & x(t_n) + \partial \cdot {}^\star f(x(t_n), t_n) \end{array} \tag{10.2}$$

Hence, like the semantics of discrete systems, the semantics of continuous systems can be described by a (non-standard) discrete sequence of states. This is exploited in the semantics of Zelus.

10.4.3 Sketch of the Semantics of Zelus

As in Lucid Synchrone, discrete statements and nodes in Zelus execute on a logical base clock. ODEs, however, execute on continuous (physical) time. The relation between the two is established with the help of zero-crossings: discrete statements are activated by zero-crossings, *i.e.*, the "base clock" of a discrete statement in a hybrid program consists of the instants when its associated zero crossings are enabled.

Example 10.6 (Relating physical and logical time)

```
let node cnt () = n where rec n = 0 fby n+1
let hybrid main () = x where
  rec der t = 1 init 0 reset 0 every (up (t-10))
  and x = (cnt ()) every init or (up (t-10))
```

One step of the discrete node `cnt` *is executed at the initial instant ("zero-crossing"* `init`*) and every time the physical clock* `t` *reaches* 10*. Observe that the output of* `cnt` *is discrete, whereas* `x` *is a (piecewise-constant) continuous variable.*

All discrete behaviors – discrete variable changes, discontinuities of continuous variables, and changes of the dynamics of continuous variables – happen on the activation of a zero-crossing. Hence, an execution of a program starts with an *initialization* followed by an alternation of a *continuous evolution* phase and a discrete transition phase. The *discrete transition* phase may consist of several discrete transitons, because a discrete transition may activate a zero-crossing and thus another discrete transition.

This operational semantics actually emulates the behavior of a simulator described in §10.3 and depicted in Fig. 10.6. Non-standard analysis comes into the semantics w.r.t. the following aspects:

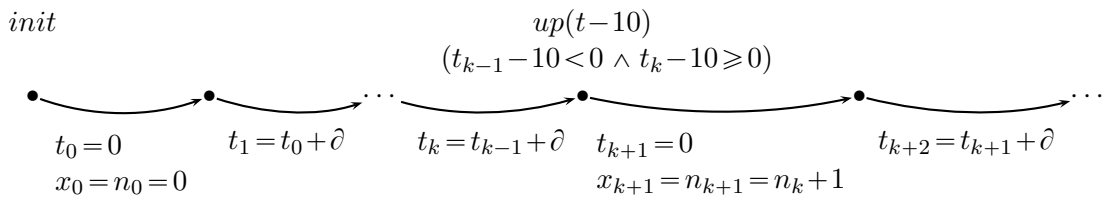


Figure 10.10: Execution of the program in Ex. 10.6

– The continuous evolution of an ODE is defined using §10.4.2 according to Eq. (10.2), *i.e.*, the continuous evolution phase is actually a non-standard infinite sequence of infinitesimal integration steps $\boldsymbol{x}' = \boldsymbol{x} + \partial \cdot \boldsymbol{f}^c(\boldsymbol{s})$, *i.e.*, in Fig. 10.6 the integration step $\delta \in \mathbb{R}^{>0}$ is replaced by a non-standard infinitesimal ∂. This allows us to describe the whole execution of a program as a (non-standard) sequence of infinitesimal (continuous) and discrete steps.
– The semantics of a zero-crossing is then defined as a predicate over two neighboring configurations in the execution sequence $\ldots \to x_{k-1} \to x_k \to \ldots$, *e.g.*, $up(10 - x)$ is enabled if $(10 - x_{k-1} \leqslant 0) \wedge (10 - x_k > 0)$.

Fig. 10.10 illustrates the execution of the program in Ex. 10.6.

In §11.1 we will present a basic hybrid data-flow formalism that can be obtained from Zelus by replacing the syntactic sugar by the corresponding data-flow primitives (see [Pou02, CPP05]). The semantics of this hybrid data-flow formalism can be defined in ten lines (Def. 11.2). For a formal definition of the semantics over Zelus itself, we refer to [BBCP12, BBCP11a].

Compilation

The current Zelus compiler [BBCP11a] targets simulation. It features type inference, initialization and causality analysis (inherited from Lucid Synchrone (§2.2)) and it ensures the proper combination of discrete and continuous statements.

After these passes, the compiler performs a source-to-source compilation, which translates the hybrid program into a discrete one. This transformation extends the function signatures in order to enable communication with the ODE solver. Currently, the variable-step numerical solver Sundials CVODE [HBG+05] is supported.

The obtained program is called by the ODE solver in order to evaluate the derivatives and the zero-crossing expressions, and by the main simulation loop for computing the discrete transitions.

While this transformation enables simulation, – besides the passes for ensuring semantical consistency – it is not useful for program verification. Therefore, we will present a translation of the language to hybrid automata, the prevalent representation for hybrid system verification, in the next section §11.

Chapter 11

From Hybrid Data Flow to Logico-Numerical Hybrid Automata

Our goal is to verify hybrid systems written in simulation languages like Simulink [Sim], Modelica [FE98], or Zelus (§10.4).

However, there is a conceptual mismatch between these high-level hybrid system languages and *hybrid automata* (§10.2), the representation well-suited for verification. The main differences between these *simulation* and *verification* formalisms can be summarized as follows:

– equations with implicitly (*i.e.*, in Boolean variables) encoded continuous modes vs their explicit encoding in locations of an automaton,
– discrete transitions triggered by zero-crossings vs combinations of staying conditions and guards,
– deterministic, open systems with inputs vs non-deterministic, closed systems.

Our primary goal is to formalize the translation from a hybrid data-flow formalism to hybrid automata, and in particular to focus on the translation of zero-crossings. However, a secondary aspect we have in mind is that we want to address hybrid systems specified as the composition of a discrete controller and its physical environment. Hence, the discrete part of the system's state space might be complex, and defined by Boolean and numerical variables (*e.g.*, counters and thresholds manipulated by the controller).

Consequently, we show how to translate the data-flow input language to *logico-numerical hybrid automata* that can manipulate symbolically discrete variables, in addition to the continuous variables governed by differential equations. Such automata allow a compact representation by not requiring the enumeration of the discrete state space.

Outline. Our contributions presented in this chapter can be summarized as follows, see Fig. 11.1:

1. We present a simple, yet complete *hybrid data-flow formalism* (§11.1) inspired by Zelus. The purpose of this formalism is to serve as a common low-level basis for the translation of languages like Simulink or Zelus.
2. We introduce *logico-numerical hybrid automata* (§11.2), *i.e.*, an extension of classical hybrid automata by Boolean variables. This extension prepares us w.r.t. the verification

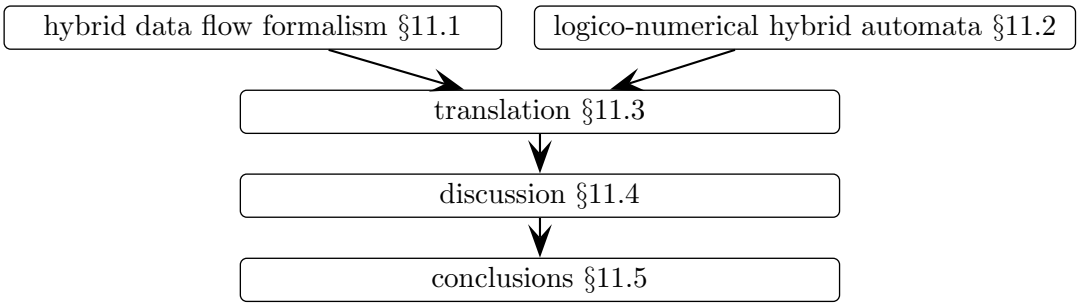


Figure 11.1: Chapter organization

of programs with a large Boolean state space.
3. We propose *sound translations* (§11.3) from the hybrid data-flow formalism to logico-numerical hybrid automata and we discuss various zero-crossing semantics. Since the target language of the translation is less expressive than the source language, the translation entails an over-approximation. We investigate the extent to which the original semantics is preserved in the translation.

Related work. There are existing translations of Simulink/Stateflow to hybrid automata, but they only treat a subset of the ways in which discrete transitions may be activated. They suppose that the diagram obeys a specific scheme (cf. Fig. 10.5b) where the discrete behavior is defined only by Stateflow diagrams and the output of the Stateflow diagram is used to select the continuous behavior using switches in the Simulink diagram. Such diagrams are straightforward to translate because the structure is already the one of a hybrid automaton and they contain no zero-crossings.

Such translations of subsets of the Simulink/Stateflow language are for example proposed by Agrawal et al [ASK04] for the purpose of verification, by Alur et al [AKRS08] with the goal of improving simulation coverage, and in the tool HyLink [MMBC11] targeting the applications of verification and controller synthesis. HyLink also introduces blocks for specifying non-deterministic inputs as required by verification methods. A formal definition of their translation is ongoing work.

Briand and Jeannet [BJ10] pursue a similar goal to ours: the verification of hybrid systems with a large discrete state space. However, they do not consider an integrated hybrid system language, but a kind of hybrid automata embedding Lustre programs with their own semantics, whereas our goal is to be compatible with standard hybrid automata model.

The translation of discrete-time Simulink models with periodic triggers to Lustre is presented in [TSCC05]. The inverse of what we are doing, namely the embedding of hybrid automata in a hybrid system language (here Scicos), is the goal of [NN07].

11.1 Hybrid Data-flow Formalism

Simulink and Zelus are full programming languages with constructs for modularity. In order to abstract from such constructs, we present here a lower-level data-flow formalism that will serve as the generic input language for the translation.

As this formalism is dedicated not only to simulation, but also serves as a specification language, we use the notion of *inputs* constrained by an *assertion* as in Lus-

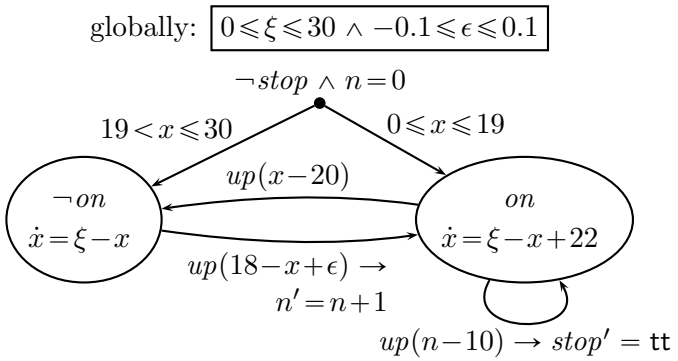


Figure 11.2: Thermostat (Example 11.1): partitioned data-flow model.

TRE (§2.2). This allows us to give a semantics to the components of a more general system. Simulation can still be performed by connecting a component with inputs to an input generator, see for instance [RRJ08] for the simulation of discrete synchronous systems.

Notations. Additional to our notational conventions of logico-numerical programs (§7.1), we will use the following notations:

$e(\boldsymbol{s}, \boldsymbol{i})$	:	an arithmetic expression without test, *e.g.*, $n+2x+\xi$;
$up(z(\boldsymbol{s}, \boldsymbol{i}))$	:	a zero-crossing, *e.g.*, $up(x+\xi-n)$;
$\varphi^Z(\boldsymbol{s}, \boldsymbol{i})$	:	a logical combination of zero-crossings, *e.g.*, $up(z_1) \wedge \neg up(z_2) \vee up(z_3)$;
$\phi(\boldsymbol{b})$	:	a Boolean expression over discrete state variables;
$\Phi(\boldsymbol{s}, \boldsymbol{i})$	:	an arbitrary expression without zero-crossings.

Program model. A hybrid data-flow program is defined similarly to a logico-numerical program (§7.1), but extended with ODEs:

Definition 11.1 (Hybrid data-flow program) *A hybrid data-flow program is defined by:*

$$\left\{ \begin{array}{l} \mathcal{I}(\boldsymbol{s}) \\ \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \left\{ \begin{array}{l} \dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}, \boldsymbol{i}) \\ \boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s}, \boldsymbol{i}) \end{array} \right. \end{array} \right.$$

where the predicate $\mathcal{I}(\boldsymbol{s})$ defines the initial states, the predicate $\mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ is the global assertion *constraining the inputs, the continuous flow equations $\dot{\boldsymbol{x}} = \boldsymbol{f}^c(\boldsymbol{s}, \boldsymbol{i})$ and the discrete transition functions $\boldsymbol{s}' = \boldsymbol{f}^d(\boldsymbol{s}, \boldsymbol{i})$ are of the form:*

$$\dot{\boldsymbol{x}} = \left\{ \begin{array}{c} \cdots \\ \boldsymbol{e}_l(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \phi_l(\boldsymbol{b}) \\ \cdots \end{array} \right. \qquad \boldsymbol{s}' = \left\{ \begin{array}{c} \cdots \\ \boldsymbol{\Phi}_j(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i}) \\ \cdots \end{array} \right.$$

We assume that the conditions ϕ_l define a partition of the discrete state space, and that $\forall \boldsymbol{s} \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i})$ (*i.e.*, the assertion does not constrain the state-space, see §2.3).

Although hybrid system languages often include explicit automata representations, for uniformity of presentation we assume that they have first been transformed into data-flow equations (see [CPP05]).

Example 11.1 (Thermostat system) *As an illustrative example we use a variant of the classical thermostat example:*

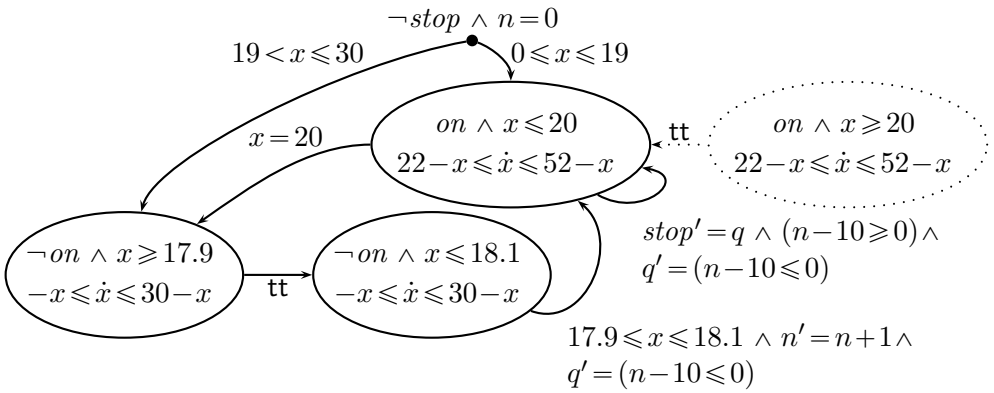


Figure 11.3: Thermostat (Example 11.1): resulting hybrid automaton (q is a Boolean state variable introduced by the translation).

```
let node main (xi,eps) = (assert,ok) where
    rec assert = 0<=xi && xi<=30 && -0.1<=eps && eps<=0.1 and ok = true
    and der x = if on then xi-x+22 else xi-x init xi
    and on = true every (up(18-x+eps))
          | false every (up(x-20)) init xi<=19
    and n = (last n)+1 every (up(18-x+eps)) init 0
    and stop = true every (up(n-10)) init false
```

The input `xi` *represents the external temperature, the input* `eps` *models the inaccuracy of the temperature sensor*[1]*, room temperature, the discrete Boolean state variable* `on` *indicates the state of the heating system, and the discrete integer state variable* `n` *counts the number of times the temperature goes from below to above 18 degrees (modulo the uncertainty). At last, the state variable* `stop` *becomes true when* `n` *reaches* 10 *from below.*

The output pair `(assert,ok)` *corresponds to the outputs* $(\mathcal{A}, \mathcal{G})$ *of a synchronous observer used for specifying a property to verify – in the sequel we need only the assumption* $\mathcal{A}$ *on the inputs.*

Translating this program to our hybrid data-flow formalism we obtain:

$$
\begin{aligned}
\mathcal{I}(on, stop, n, x) &= \neg stop \ \wedge \ n=0 \ \wedge \ 0 \leqslant x \leqslant 30 \ \wedge \\
&\quad ((x \leqslant 19 \ \wedge \ on) \ \vee \ (x > 19 \ \wedge \ \neg on)) \\
\mathcal{A}((on, stop, n, x), (\xi, \epsilon)) &= 0 \leqslant \xi \leqslant 30 \wedge -0.1 \leqslant \epsilon \leqslant 0.1 \\
\dot{x} &= \begin{cases} \xi - x + 22 & \text{if } on \\ \xi - x & \text{if } \neg on \end{cases} \\
(on', stop', n', x') &= \begin{cases} (\mathsf{ff}, stop, n, x) & \text{if } up(x-20) \\ (\mathsf{tt}, stop, n+1, x) & \text{if } up(18-x+\epsilon) \\ (on, \mathsf{tt}, n, x) & \text{if } up(n-10) \end{cases}
\end{aligned}
$$

Observe that this translation factorizes the evolution of discrete variables according to the zero-crossing conditions.

[1] We do not use `eps` in the expression $up(x-20)$, in order to show an example of a deterministic zero-crossing.

Semantics of zero-crossings. A *zero-crossing* is an expression of the form $up(z)$ that becomes true when the sign of $z(\boldsymbol{s},\boldsymbol{i})$, an arithmetic expression without tests, switches from negative to positive during an execution. Instead of just a valuation of variables of the form $(\boldsymbol{s}_k,\boldsymbol{i}_k)$ zero-crossings are interpreted on an execution fragment of the form $(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1})\to(\boldsymbol{s}_k,\boldsymbol{i}_k)$, *i.e.*, two consecutive configurations of an execution trace. We will use the notation $(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1},\boldsymbol{s}_k,\boldsymbol{i}_k)$ for short. Several interpretations are possible, which are discussed in §11.1. For now, we arbitrarily select the so-called "contact" semantics, formally defined as:

$$(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1},\boldsymbol{s}_k,\boldsymbol{i}_k)\models up(z(\boldsymbol{s},\boldsymbol{i}))\ \text{ iff }\ \begin{cases} z(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1})<0\\ z(\boldsymbol{s}_k,\boldsymbol{i}_k)\geqslant 0\end{cases} \tag{11.1}$$

In other words, a zero-crossing $up(z)$ is activated (and taken into account for computing the next step $k+1$) if the expression z was strictly negative in the previous step $k-1$ and evaluates to some positive value or zero in the current step k.

A conjunction $\varphi^Z(\boldsymbol{s},\boldsymbol{i}) = \bigwedge_p up(z_p(\boldsymbol{s},\boldsymbol{i}))$ is activated if for all p, $up(z_p(\boldsymbol{s},\boldsymbol{i}))$ is activated in the same step.

Semantics. We define a trace semantics based on an ideal discretization of the continuous equations that follows the one of Zelus (§10.4.3). This semantics uses the theory of non-standard analysis (§10.4.2) to model the way typical numerical simulators proceed (cf. Fig. 10.6): Such solvers are given an initial state $\boldsymbol{x}_0$, an ODE $\dot{\boldsymbol{x}}(t)=\boldsymbol{f}(\boldsymbol{x}(t))$, and a finite set of zero-crossing expressions z_j. They integrate the ODE until at least one of the zero-crossings is activated. When this happens, the control is given back to the main simulation loop, which executes one or several discrete execution steps before continuing integration.

Definition 11.2 (Semantics) *An* execution *of a hybrid data-flow program is a trace* $(\boldsymbol{s}_0,\boldsymbol{i}_0)\to(\boldsymbol{s}_1,\boldsymbol{i}_1)\to(\boldsymbol{s}_2,\boldsymbol{i}_2)\to\ldots$ *such that* $\mathcal{I}(\boldsymbol{s}_0)$, $\to=\to_c\cup\to_d$,

$$(\boldsymbol{s}_k,\boldsymbol{i}_k)\to_c(\boldsymbol{s}_{k+1},\boldsymbol{i}_{k+1})\iff\mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i}_k)\ \wedge$$
$$\exists l,\exists\partial>0:\begin{cases}\phi_l(\boldsymbol{b}_k)\wedge\forall j:\neg\big((\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1},\boldsymbol{s}_k,\boldsymbol{i}_k)\models\varphi^Z_j(\boldsymbol{s},\boldsymbol{i})\big)\\ (\boldsymbol{b}_{k+1},\boldsymbol{x}_{k+1})=(\boldsymbol{b}_k,\boldsymbol{x}_k+\boldsymbol{e}_l(\boldsymbol{s}_k,\boldsymbol{i}_k)\cdot\partial)\end{cases}$$

where ∂ *is a non-standard infinitesimal, and*

$$(\boldsymbol{s}_k,\boldsymbol{i}_k)\to_d(\boldsymbol{s}_{k+1},\boldsymbol{i}_{k+1})\iff\mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i}_k)\wedge$$
$$\exists j:\begin{cases}(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1},\boldsymbol{s}_k,\boldsymbol{i}_k)\models\varphi^Z_j(\boldsymbol{s},\boldsymbol{i})\ \wedge\\ \qquad\forall j'<j:\neg\big((\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1},\boldsymbol{s}_k,\boldsymbol{i}_k)\models\varphi^Z_{j'}(\boldsymbol{s},\boldsymbol{i})\big)\\ \boldsymbol{s}_{k+1}=\Phi_j(\boldsymbol{s}_k,\boldsymbol{i}_k)\end{cases}$$

A transition $\to_c$ corresponds to an infinitesimal continuous-time evolution, which is possible only if no zero-crossing condition φ^Z_j has been activated in the previous execution step. A transition $\to_d$ corresponds to a discrete transition triggered by the first enabled φ^Z_j.

We pinpoint some properties of this formalism:

(1) Discrete transitions are always guarded by zero-crossings, and continuous modes are always defined by a Boolean expression over discrete variables, which are piecewise constant in continuous time. This is to make sure that a mode change (change of dynamics) can only happen on discrete transitions.

(2) Furthermore, discrete transitions are *urgent*, *i.e.*, they must be taken at the first point in time possible.

(3) In case of simultaneously occurring zero-crossings – as in ZELUS – the only the first activated zero-crossings in the program source is taken into account.

(4) Zero-crossings may not only be triggered by continuous evolution, but also by discrete transitions. This is the case in Example 11.1: if the zero-crossing $up(18-x+\epsilon)$ occurs when $n=9$, n is first incremented to 10, activating the zero-crossing $up(n-10)$ that makes *stop* become true. This feature can cause infinite sequences of discrete zero-crossings. Such a behavior can be avoided by forbidding circular dependencies between states variables through zero-crossings in the source program.

Partitioned representation. The hybrid data-flow model we have defined does not have any concept of control structure. However, for pedagogical purpose, one can partition the state space to generate an explicit automaton that may be easier to understand, see Fig. 11.2. When doing so, partial evaluation may be used to simplify expressions and removing infeasible transitions (cf. §7.3.3). This has been done in Fig. 11.2.

Standardization. As already mentioned the semantics of the hybrid data flow model is based on non-standard analysis (§10.4.2), which gives an unambiguous meaning to hybrid systems even if they contain Zeno behavior for instance.

However, the semantics of the output formalism of our translation, *i.e.*, hybrid automata, relies on standard analysis. Hence non-standard behaviors need to be mapped to standard behaviors. This *standardization* is based on the transfer principle (Thm. 10.2): since each standard system has a non-standard representation, a non-standard system is standardizable if it is a non-standard representation of a standard system.

For instance, w.r.t. continuous evolution, we have the following property: a non-standard sequence consisting of infinitesimal continuous steps

$$(\boldsymbol{s}_0, \boldsymbol{i}_0) \rightarrow \ldots \rightarrow (\boldsymbol{s}_n, \boldsymbol{i}_n)$$

with $n \in {}^{\star}\mathbb{N}$, $\boldsymbol{s}_k = (\boldsymbol{b}_k, \boldsymbol{x}_k)^T$, $\boldsymbol{x}_0 \in \mathbb{R}^p$, $\boldsymbol{x}_n \in \mathbb{R}^p$, has the following standard meaning: assuming that the input sequence $\boldsymbol{i}_0 \rightarrow \ldots \rightarrow \boldsymbol{i}_n$ forms a continuous function $\boldsymbol{i} : [0, \delta] \rightarrow I$, the sequence $\boldsymbol{s}_0 \rightarrow \ldots \rightarrow \boldsymbol{s}_n$ corresponds to a continuous function $\boldsymbol{s} : [0, \delta] \rightarrow S$ with (cf. Eq. 10.2)

$$\boldsymbol{x}(\delta') = \boldsymbol{x}_0 + \int_0^{\delta'} \boldsymbol{e}_\ell((\boldsymbol{b}_0, \boldsymbol{x}(t))^T, \boldsymbol{i}(t))dt$$

for $\delta' \in [0, \delta]$, $\delta = st(n\partial) \in \mathbb{R}^{\geqslant 0}$, and I and S denote the input and state space respectively.

However, we can write programs that are not standardizable, *i.e.*, for which non-standard and standard meaning differ: For example the program fragment $b' = (x > 0)$ if $up(x)$ with "crossing" semantics (see below) gives us $b' = \mathtt{tt}$ in the non-standard interpretation, but $b' = \mathtt{ff}$ in the standard interpretation.

Naturally, we can only correctly translate standardizable programs.

Semantics of zero-crossings

As mentioned above, a zero-crossing can be activated in two ways (Fig. 11.4):

– It can be triggered by a continuous time evolution, as $up(x-20)$ in Fig. 11.2; in this case it is active during the second step of an execution fragment $\boldsymbol{s} \xrightarrow{\boldsymbol{i}}_c \boldsymbol{s}' \xrightarrow{\boldsymbol{i}'}_d \boldsymbol{s}''$;

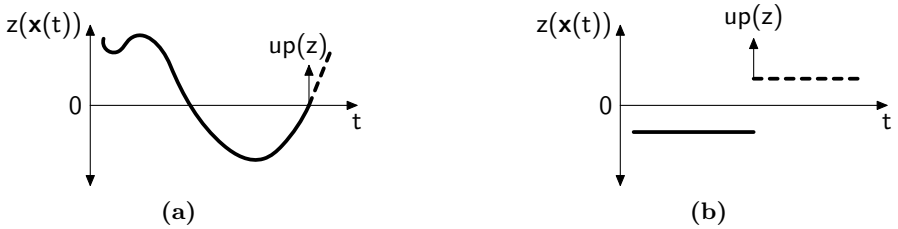


Figure 11.4: Continuous (a) and discrete (b) zero-crossings

– It can be triggered by a discrete transition, as $up(n-10)$ in Fig. 11.2; in this case it is active during the second step of an execution fragment $\boldsymbol{s} \xrightarrow{\boldsymbol{i}}_d \boldsymbol{s}' \xrightarrow{\boldsymbol{i}'}_d \boldsymbol{s}''$.

Because a zero-crossing may depend both on discrete and continuous variables, the same zero-crossing $up(z)$ can be triggered in both ways in an execution. We will use the terms *continuous* (resp. *discrete*) *zero-crossing* to indicate its source of activation.

Three semantics for zero-crossings. We consider an execution fragment $\xrightarrow{\boldsymbol{i}_{k-1}} \boldsymbol{s}_{k-1} \xrightarrow{\boldsymbol{i}_k} \boldsymbol{s}_k$ and we define $z_k = z(\boldsymbol{s}_k, \boldsymbol{i}_k)$. There are three natural choices for the semantics of zero-crossings:

– "At-zero" semantics : $z_{k-1} \leqslant 0 \wedge z_k \geqslant 0$
– "Contact" semantics : $z_{k-1} < 0 \wedge z_k \geqslant 0$
– "Crossing" semantics : $z_{k-1} \leqslant 0 \wedge z_k > 0$

Figs. 11.5b, 11.6a and 11.6d illustrate the activation of continuous zero-crossings for some typical trajectories according to each semantics.

The last two semantics are used in simulators. The zero-crossing semantics of Simulink is the disjunction of "contact" and "crossing" semantics. In Modelica it is up to the programmer to choose between these two semantics. We state the first option, because it fits better to the semantics of hybrid automata (as it does not involve strict inequalities).

Chattering behavior. An issue specific to the "crossing" semantics is that it is possible to write programs that produce executions that contain periodic sequences of infinitesimal continuous evolutions with distinct dynamics. This happens for example when a trajectory *chatters* along a surface with opposed zero-crossings, like in Ex. 10.3.

As explained in §10.3 one way to treat such behavior would be to transform the program such as to replace the chattering by its corresponding *sliding mode* (Fig. 10.8b); yet, since we do not know how to achieve this in general, we have chosen to translate such programs into hybrid automata that allow chattering in their concrete semantics.

11.2 Logico-Numerical Hybrid Automata

We extend hybrid automata (cf. §10.2) in the sense that we allow also Boolean variables in the expressions occurring in the automaton.

Definition 11.3 (Logico-numerical hybrid automaton) *A logico-numerical hybrid automaton (HA) is a directed graph defined by* $\langle L, F, J, \Sigma^0 \rangle$ *where*
– *L is the finite set of locations,*

- $F : L \to \mathcal{V}$ *is a function that returns for each location the flow relation* $V(\boldsymbol{s}, \dot{\boldsymbol{x}}) \in \mathcal{V}$ *relating the state variables* $\boldsymbol{s}$ *and the time-derivatives* $\dot{\boldsymbol{x}}$ *of the numerical state variables, and*
- $J \subseteq L \times \mathcal{R} \times L$ *defines a finite set of arcs between locations with the discrete transition relation* $R(\boldsymbol{s}, \boldsymbol{s}') \in \mathcal{R}$ *over the state variables* $\boldsymbol{s}$*, and*
- $\Sigma^0 : L \to \mathcal{S}$ *is a function that returns for each location the set of initial states* $S^0 \in \mathcal{S}$*, which have to satisfy* $\forall \ell : \forall \boldsymbol{s} : \Sigma^0(\ell)(\boldsymbol{s}) \Rightarrow \exists \dot{\boldsymbol{x}} : F(\ell)(\boldsymbol{s}, \dot{\boldsymbol{x}})$.

Further notations:
- $C_\ell(\boldsymbol{s}) = \exists \dot{\boldsymbol{x}} : V_\ell(\boldsymbol{s}, \dot{\boldsymbol{x}})$ is the *staying condition* of the flow $V_\ell = F(\ell)$.
- $G_{\ell,\ell'}(\boldsymbol{s}) = \exists \boldsymbol{s}' : R(\boldsymbol{s}, \boldsymbol{s}')$ is the *guard* of the arc $(\ell, R, \ell') \in J$.

Fig. 11.3 depicts an example of such a logico-numerical hybrid automaton.

Semantics. We use the following auxiliary definitions: Let $T_{[0,\delta]}$ be the set of differentiable trajectories $[0, \delta] \to \mathbb{R}^n$. The function *flow*$_V$ returns the set of end states of trajectories $\tau \in T$ starting in the given state and that obey the flow relation V_ℓ:

$$\mathit{flow}_{V_\ell}(\boldsymbol{b}, \boldsymbol{x}) = \left\{ (\boldsymbol{b}, \boldsymbol{x}') \;\middle|\; \begin{array}{l} \exists \delta > 0, \exists \tau \in T_{[0,\delta]} : \\ \quad \tau(0) = \boldsymbol{x} \;\wedge\; \tau(\delta) = \boldsymbol{x}' \;\wedge \\ \quad \forall \delta' \in [0, \delta] : C_\ell(\boldsymbol{b}, \tau(\delta')) \;\wedge \\ \quad \forall \delta' \in (0, \delta) : V_\ell((\boldsymbol{b}, \tau(\delta')), \dot{\tau}(\delta')) \end{array} \right\}$$

We define the concrete semantics in terms of an *execution* of a hybrid automaton:

Definition 11.4 (Semantics) *An* execution *of a logico-numerical hybrid automaton is a (possibly) infinite trace* $(\ell_0, \boldsymbol{s}_0) \to (\ell_1, \boldsymbol{s}_1) \to (\ell_2, \boldsymbol{s}_2) \to \ldots$ *with* $\to = \to_c \cup \to_d$ *and*

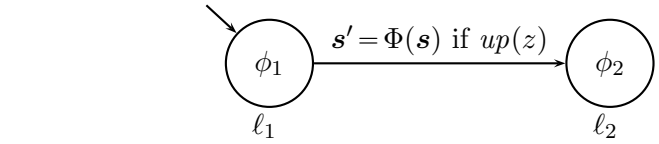
$$\begin{array}{rcl} (\ell, \boldsymbol{s}) \to_c (\ell', \boldsymbol{s}') & \Leftrightarrow & \ell = \ell' \wedge V_\ell = F(\ell) \wedge \boldsymbol{s}' \in \mathit{flow}_{V_\ell}(\boldsymbol{s}) \\ (\ell, \boldsymbol{s}) \to_d (\ell', \boldsymbol{s}') & \Leftrightarrow & \exists (\ell, R, \ell') \in J : R(\boldsymbol{s}, \boldsymbol{s}') \wedge C_{\ell'}(\boldsymbol{s}') \end{array}$$

If we eliminate all Boolean variables by enumerating their valuations and encoding them with locations, the semantics above will be equivalent to the semantics of standard hybrid automata that deal only with numerical variables (see §10.2).

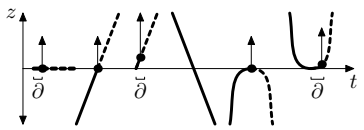
11.3 Translation

The main issue in the translation are the zero-crossings: the fundamental difference between the zero-crossing concept used in our input language and the combination of staying and jump conditions in our output language is that the activation of a zero-crossing depends on the history (*i.e.*, a part of the past trajectory) whereas the truth value of staying and jump conditions depends only on the current state.

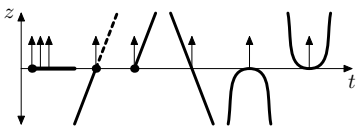
We start with translations for continuous zero-crossings without (§11.3.1) and with (§11.3.2) inputs; then we will show how to translate logical combinations of zero-crossings in §11.3.3. §11.3.4 will discuss the case of discrete zero-crossings, the translation of which is much less dependent on the choice between the three zero-crossing semantics. However, because of the limitations of the hybrid automata model, in all cases the translation will add behaviors that are not present in the original program.



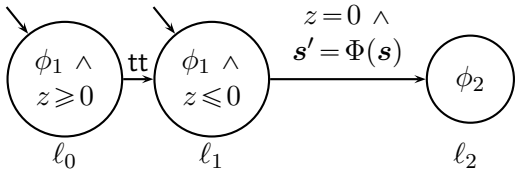
(a) Single zero-crossing



(b) Exact "at-zero" semantics



(c) Induced semantics



(d) Translation according to "at-zero" semantics

Notes: (1) The arrows pointing upwards in (b) indicate the points where zero-crossings are activated, and in (c), the points where the jump transition may be taken non-deterministically. The dotted trajectories indicate that the preceding transition is urgent.
(2) When $\boldsymbol{s}$ does not appear in the jump condition of a HA, the equality $\boldsymbol{s}' = \boldsymbol{s}$ is implicit.
(3) The flow equation $\dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s})$ *if* $\phi_1(\boldsymbol{b})$ in location ℓ_1 in (a), is translated to the flow relation $V(\dot{\boldsymbol{x}}, \boldsymbol{s}) = (\phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s}))$ in locations ℓ_0 and ℓ_1 in (d) (only the staying conditions are given in the figures).

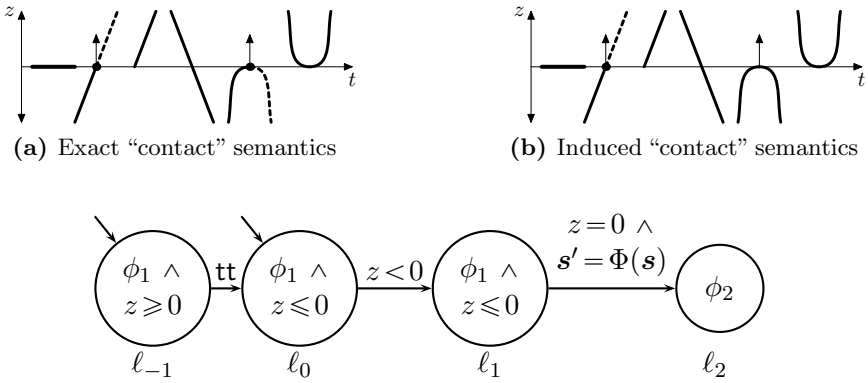
Figure 11.5: Zero-crossing semantics of the hybrid data-flow language and their translations. The diagram in (b) shows typical trajectories in the original semantics of the partitioned data flow model in (a), the one in (c) shows typical trajectories in the semantics of the proposed translation to the hybrid automaton in (d).

11.3.1 Continuous Zero-Crossing without Inputs

We investigate here the translation of a continuous zero-crossing of the form $up(z(\boldsymbol{x}))$: for the sake of simplicity, we assume that there are neither inputs $\boldsymbol{i}$ nor discrete variables $\boldsymbol{b}$ in z. We consider the simple case of an origin location ℓ_1 with a single discrete transition $\boldsymbol{s}' = \Phi(\boldsymbol{s})$ if $up(z(\boldsymbol{x}))$ going from ℓ_1 to a location ℓ_2, such that $\phi_1(\boldsymbol{b}) \wedge (\boldsymbol{s}' = \Phi(\boldsymbol{s})) \Rightarrow \phi_2(\boldsymbol{b}')$, see Fig. 11.5a. The continuous dynamics in location ℓ_1 corresponds to the flow equation $\dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s})$ *if* $\phi_1(\boldsymbol{b})$.

As the satisfaction of a zero-crossing depends on the history, the principle of the translation is to add locations to record the history of the continuous evolution. In the sequel, we develop translations corresponding to the three zero-crossing semantics stated in §11.1.

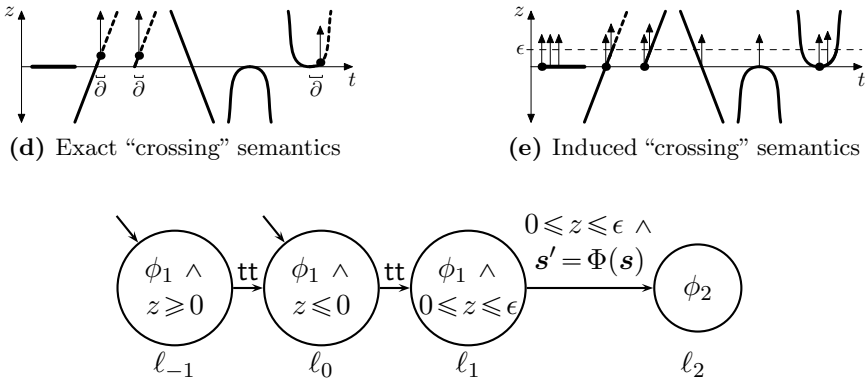
"At-zero" semantics. The translation of "at-zero" semantics ($z_{k-1} \leqslant 0 \wedge z_k \geqslant 0$) is depicted in Fig. 11.5. The origin location ℓ_1 in Fig. 11.5a is partitioned in two locations, ℓ_0 and ℓ_1, in Fig. 11.5d: there is a discrete transition from ℓ_0 to ℓ_1, but not from ℓ_1 to



(a) Exact "contact" semantics

(b) Induced "contact" semantics

(c) Translation according to "contact" semantics



(d) Exact "crossing" semantics

(e) Induced "crossing" semantics

(f) Translation according to "crossing" semantics

Figure 11.6: Zero-crossing semantics of the hybrid data-flow language and their translations (continuation, see Fig. 11.5 for explanations).

ℓ_0 to force the urgency of the discrete transition when $z = 0$ is reached from below 0. The zero-crossing condition translates to $z = 0$ (see transition from ℓ_1 to ℓ_2). The rationale for the condition $z = 0$ is based on the assumption of continuity of the function $z(t) = z(\boldsymbol{x}(t))$ and the urgency of the zero-crossing: $z(t_{k-1}) < 0 \wedge z(t_k) \geqslant 0$ with $t_{k-1} < t_k$ implies that there exists $t \in (t_{k-1}, t_k]$ such that $z(t) = 0$.

This translation induces two kinds of approximations in terms of executions:

– We lose urgency for all trajectories but the second one in Fig. 11.5c. In case of the first trajectory the zero-crossing may be triggered in a dense interval of time.
– We add a jump transition in the fourth trajectory because the resulting hybrid automaton is not able to distinguish whether the state $z = 0$ is reached from below or from above 0.

We will not consider any more the "at-zero" semantics in the sequel, as – to our knowledge – it is not used by any simulation tool.

"Contact" semantics. In order to translate the "contact" semantics defined as $z_{k-1} < 0 \wedge z_k \geqslant 0$, we split the original location into three locations, ℓ_{-1}, ℓ_0, and ℓ_1, as depicted in Fig. 11.6c. The two locations, ℓ_0 and ℓ_1, both with the staying condition $z \leqslant 0$, are connected by a transition guarded by $z < 0$: this is in order to check that the trajectory was actually strictly below zero before touching zero. This prohibits the triggering of

the jump transition in the first, third and last trajectory in Fig. 11.6b. This induces the following approximation:

– The loss of urgency for the fifth trajectory that touches (possibly several times) the line $z=0$ from below.

Observe, that the "at-zero" translation in Fig. 11.5d is actually a sound translation of the "contact" semantics, though with coarser approximations.

"Crossing" semantics. The "crossing" semantics ($z_{k-1} \leqslant 0 \wedge z_k > 0$) is more subtle to translate. By continuity of the function $z(t) = z(\boldsymbol{x}(t))$ we can deduce that $z(t)=0$ is valid at the zero-crossing point in standard semantics: by standardizing $z(t) \leqslant 0 \wedge z(t+\partial) > 0$ we get $st(z(t)) = st(z(t+\partial)) = 0$.

However, we cannot simply reuse the "at-zero" translation in Fig. 11.5d, because it is not sound w.r.t. chattering behaviors: in Ex. 10.3, time cannot advance, because only discrete transitions can be taken. Since we do not rely on standardizing chattering behaviors, we have to allow chattering in the standard semantics. For this reason, we allow the trajectories to actually go above zero, but only up to a constant $\epsilon > 0$ (see Fig. 11.6f). As a consequence, we have the following approximation:

– Urgency is completely lost. In case of the second, third, and last trajectories, the zero-crossing may be triggered in a bounded time interval with a dense interval of values for z (see Fig. 11.6e).

Observe, that this translation simulates the translations of the two other semantics.

Remark 11.1 (Blocked executions) *These translations may add so-called* blocked *executions. Consider the fifth trajectory of Fig. 11.6d ("crossing" semantics): this trajectory is possible when staying in the second location in Fig. 11.6f.*

Yet, it is also possible to move to location ℓ_1 when this trajectory reaches zero, but then it gets stuck at zero: neither continuous nor discrete transitions are possible in ℓ_1.

However, this phenomenon has no effect w.r.t. reachability properties, on which we focus: the goal of our translation is to obtain a precise model suitable for safety verification techniques.

Remark 11.2 (Choice of ϵ) *Any translation involving an ϵ close to zero is not really well-suited for verification: computations with arbitrary-precision rationals become indeed very expensive (*e.g.*, least common denominators become huge).*

Remark 11.3 (Exploiting derivatives) *The difficulty of translating "crossing" semantics comes from the fact that being at $z=0$ we have to peek an infinitesimal step into the future of the continuous evolution. In standard semantics, this is possible by looking at the time derivatives $\frac{d^n z}{dt^n}$ of the zero-crossing expression – if the zero-crossing contains inputs we also need bounds on the derivatives of the inputs.*

Theoretically, this would allow avoiding the activation of the discrete transition in the translation w.r.t. the first, fourth, and fifth trajectory in Fig. 11.6e. However, firstly, without any restrictions on the dynamics and the form of z, we would have to look at all *derivatives before being able to decide that a trajectory is actually crossing zero. Secondly, the translation would still be unsound w.r.t. chattering behavior of sliding modes because only discrete transitions would be taken, and hence, time cannot advance.*

11.3.2 Continuous Zero-Crossing with Inputs

Now, we investigate the translation of zero-crossings of the form $up(z(\boldsymbol{x},\boldsymbol{i}))$, where the inputs $\boldsymbol{i}$ have to satisfy an assertion $\mathcal{A}(\boldsymbol{s},\boldsymbol{i})$, see §11.1. We assume that in the discrete infinitesimal semantics of §11.1 inputs tend to continuous trajectories (between two discrete transitions). Inputs allow us to introduce non-determinism in a model, as illustrated by Fig. 11.3. The principle of translation as described in §11.3.1 and depicted in Fig. 11.5 remains the same, except that the computation of jump and flow transition relations involves an existential quantification of the inputs $\boldsymbol{i}$.

We use the notation $\boxslash\psi = \overline{\neg\psi}$, where $\bar{\cdot}$ denotes the topological closure operator. We have for instance $\boxslash(z\leqslant 0) = z\geqslant 0$.

Considering the "contact" semantics and using the continuity of the function $z(\boldsymbol{x}(t),\boldsymbol{i}(t))$ during continuous evolution (see §11.1) the condition

$$\begin{array}{l}\exists \boldsymbol{i}_{k-1},\boldsymbol{i}_k : z(\boldsymbol{x}_{k-1},\boldsymbol{i}_{k-1})<0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1}) \wedge \\ \qquad\qquad z(\boldsymbol{x}_k,\boldsymbol{i}_k)\geqslant 0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i}_k) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k,\boldsymbol{i}_k)\end{array}$$

is equivalent to

$$\begin{array}{l}\exists \boldsymbol{i}_{k-1},\boldsymbol{i}_k : z(\boldsymbol{x}_{k-1},\boldsymbol{i}_{k-1})<0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1},\boldsymbol{i}_{k-1}) \wedge \\ \qquad\qquad z(\boldsymbol{x}_k,\boldsymbol{i}_k)=0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i}_k) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k,\boldsymbol{i}_k)\end{array}$$

which in turn is equivalent to

$$\begin{array}{l}\exists \boldsymbol{i} : z(\boldsymbol{x}_{k-1},\boldsymbol{i})<0 \wedge \mathcal{A}(\boldsymbol{s}_{k-1},\boldsymbol{i}) \wedge \\ \exists \boldsymbol{i} : z(\boldsymbol{x}_k,\boldsymbol{i})=0 \quad \wedge \mathcal{A}(\boldsymbol{s}_k,\boldsymbol{i}) \wedge \boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s}_k,\boldsymbol{i})\end{array} \tag{11.2}$$

– The first line of Eqn. (11.2) defines the new guard of the transition between locations ℓ_0 and ℓ_1 of Fig 11.6c:

$$\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge z(\boldsymbol{x},\boldsymbol{i})<0$$

– The second line gives us the new transition relation between locations ℓ_1 and ℓ_2 of Fig. 11.6c:

$$R(\boldsymbol{s},\boldsymbol{s}') = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge z(\boldsymbol{x},\boldsymbol{i})=0 \wedge \boldsymbol{s}'=\Phi(\boldsymbol{s},\boldsymbol{i})$$

– The new flow relation of ℓ_0 and ℓ_1 is

$$\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge z(\boldsymbol{x},\boldsymbol{i})\leqslant 0 \wedge \phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s},\boldsymbol{i})$$

which induces the staying condition

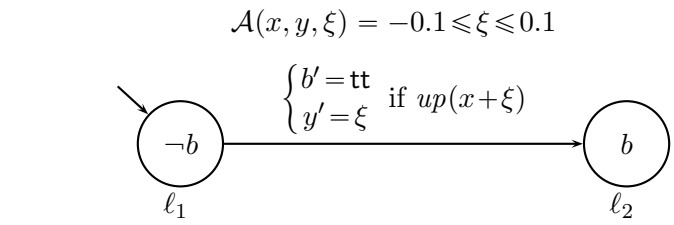
$$\psi_{23}(\boldsymbol{s}) = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge z(\boldsymbol{x},\boldsymbol{i})\leqslant 0) \wedge \phi_1(\boldsymbol{b})$$

– The new flow relation of ℓ_{-1} of Fig. 11.6c is

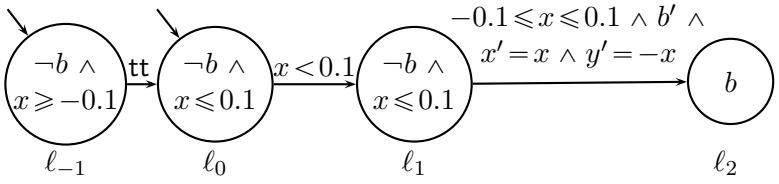
$$V_1(s,\dot{\boldsymbol{x}}) = \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s},\boldsymbol{i}) \wedge z(\boldsymbol{x},\boldsymbol{i})\geqslant 0 \wedge \phi_1(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_1(\boldsymbol{s},\boldsymbol{i})$$

The result is illustrated by Fig. 11.7b. One can strengthen the flow relation V_1 by conjoining it with $\boxslash\psi_{23}$, so as to minimize the non-determinism between staying in ℓ_{-1} or jumping to ℓ_0, as done in Fig. 11.7c.[2]

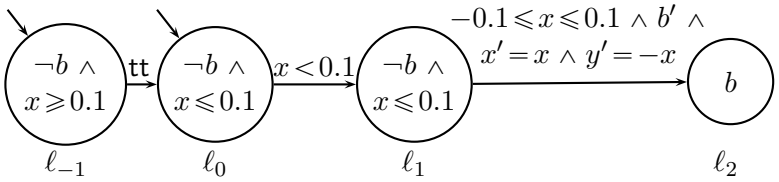
[2]We use the operator $\boxslash$ instead of $\neg$ in order to obtain a topologically closed flow relation.



(a) Zero-crossing with input ξ



(b) Translation according to "contact" semantics



(c) Strengthening the staying condition of the first location

Figure 11.7: Translation of a continuous zero-crossing with inputs, described in Example 11.2.

Example 11.2 (Continuous zero-crossing with input) *Fig. 11.7b illustrates this translation on the original system of Fig. 11.7a, where b, x, y are state variables and ξ is a numerical input variable constrained by the assertion. The jump condition of the rightmost transition is obtained from*

$$\begin{aligned} & \exists \xi : (-0.1 \leqslant \xi \leqslant 0.1 \ \wedge \ x+\xi = 0 \ \wedge \ b' = \mathsf{tt} \ \wedge \ x' = x \ \wedge \ y' = \xi) \\ = \ & \exists \xi : (-0.1 \leqslant x \leqslant 0.1 \ \wedge \ x = -\xi \ \wedge \ b' \ \wedge \ x' = x \ \wedge \ y' = -x) \\ = \ & -0.1 \leqslant x \leqslant 0.1 \ \wedge \ b' \ \wedge \ x' = x \ \wedge \ y' = -x \end{aligned}$$

Observe that we obtain the non-trivial relation $y = -x$ after the jump transition.

11.3.3 Logical Combinations of Zero-Crossings

We consider here a discrete transition function of the form $\boldsymbol{s}' = \boldsymbol{\Phi}(\boldsymbol{s})$ if $\varphi^Z(\boldsymbol{s})$ where φ^Z is a logical combination of zero-crossings $up(z_1), \ldots, up(z_M)$ satisfying the assumption of §11.1. In order to simplify the presentation, we consider the case without inputs first (see §11.3.5 for the general case).

Why do we need such logical combinations? Conjunctions and negations typically occur when combining two parallel equations $s_i' = \Phi_i$ if $up(z_i)$ for $i = 1, 2$, which results in an equation

$$(s_1', s_2') = \begin{cases} (\Phi_1, \Phi_2) & \text{if } up(z_1) \wedge up(z_2) \\ (\Phi_1, s_2) & \text{if } up(z_1) \wedge \neg up(z_2) \\ (s_1, \Phi_2) & \text{if } up(z_2) \wedge \neg up(z_1) \end{cases}$$

A discrete transition function of the form $s' = \begin{cases} \Phi_1 & \text{if } up(z_1) \\ \Phi_2 & \text{else if } up(z_2) \end{cases}$
is rewritten as $s' = \begin{cases} \Phi_1 & \text{if } up(z_1) \\ \Phi_2 & \text{if } \neg up(z_1) \wedge up(z_2) \end{cases}$
Disjunctions allow us to express that the same transition may be triggered by different zero-crossings:

$$s' = \Phi \text{ if } up(z_1) \vee up(z_2)$$

Because successive graph refinements are cumbersome to describe, we reformulate the translation scheme of the previous sections by using additional discrete state variables to the system, rather than by introducing locations. This will make it easier to define this generalization. We sketch this principle using the "contact" semantics (the translation for the general case will be presented in §11.3.5).

To encode locations, we add M discrete state variables $q_1 \dots q_M$ of the enumerated type $\{\mathsf{above}, \mathsf{below}, \mathsf{ready}\}$ for each distinct zero-crossing $up(z_m)$ occurring in the zero-crossing formulas φ^Z.

– Their transition relations are defined as

$$\begin{array}{rl} R_m = & \textbf{match } q_m \textbf{ with} \\ & \mathsf{above} \rightarrow q'_m \in \{\mathsf{above}, \mathsf{below}\} \\ & \mathsf{below} \rightarrow z_m < 0 \wedge q'_m = \mathsf{ready} \qquad \vee\, q'_m = q_m \\ & \mathsf{ready} \rightarrow z_m = 0 \wedge q'_m \in \{\mathsf{above}, \mathsf{below}\} \vee q'_m = q_m \end{array}$$

– The staying condition defined by the zero-crossing $up(z_m)$ is:

$$\begin{array}{rl} C_m = & \textbf{match } q_m \textbf{ with} \\ & \mathsf{above} \rightarrow z_m \geqslant 0 \\ & \mathsf{below} \rightarrow z_m \leqslant 0 \\ & \mathsf{ready} \rightarrow z_m \leqslant 0 \end{array}$$

– The guard G_m associated to the zero-crossing $up(z_m)$ is

$$G_m = (q_m = \mathsf{ready}) \wedge (z_m = 0)$$

We can now build the global flow and discrete transition relations:

$$\begin{array}{rl} R((\boldsymbol{q}, \boldsymbol{s}), (\boldsymbol{q}', \boldsymbol{s}')) & = (\bigwedge_m R_m) \wedge (\neg H \wedge \boldsymbol{s}' = \boldsymbol{s} \vee H \wedge \boldsymbol{s}' = \boldsymbol{\Phi}) \\ V((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) & = (\bigwedge_m C_m) \wedge (\bigvee_l (\phi_l(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \dot{\boldsymbol{e}}_l(\boldsymbol{s}))) \end{array} \tag{11.3}$$

with $H = \varphi^Z[\forall m : up(z_m) \leftarrow G_m]$ where $e[x \leftarrow y]$ means that y is substituted for x in expression e.

In order to obtain an explicit automaton one has to enumerate the valuations of the discrete state variables $\boldsymbol{q}$ and to encode them into explicit locations (see §11.4).

It is interesting to mention that this translation keeps enough information in order to preserve urgency in case of conjunctions like $s' = \Phi$ if $up(z_1) \wedge up(z_2)$ where the trajectory can move all around the intersection $z_1 = 0 \wedge z_2 = 0$ while not satisfying both zero-crossings at the same time. Fig. 11.8 gives an illustration of such a trajectory and Fig. 11.9 shows the corresponding automaton.

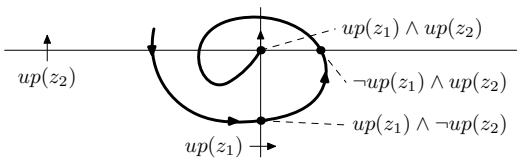


Figure 11.8: Conjunction of two zero-crossings

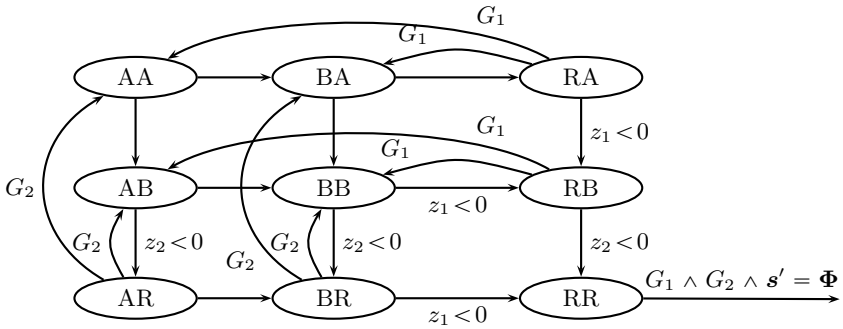


Figure 11.9: Translation of conjunctions of zero-crossings: example of two zero-crossings with "contact" semantics. $G_1 = (z_1 = 0)$, $G_2 = (z_2 = 0)$, $AA = (q_1 = \mathsf{above} \wedge z_1 \geqslant 0 \wedge q_2 = \mathsf{above} \wedge z_2 \geqslant 0)$, analogously for B (below) and R (ready). The urgency of the discrete transition $G_1 \wedge G_2 \wedge \boldsymbol{s}' = \boldsymbol{\Phi}$ is preserved to the same extent as for single zero-crossings.

Remark 11.4 (Conjunctions of zero-crossings in simulation) *Conjunctions of zero-crossings are quite delicate in hybrid simulation models. The problem is that models relying on conditions stating that two physical quantities become zero simultaneously are not numerically robust and produce unpredictable simulation results. However, from a programming language semantics point of view we have to deal with them.*

11.3.4 Discrete Zero-Crossings

Discrete zero-crossings are activated by discrete transitions. Discrete zero-crossings occur in so-called *zero-crossing cascades*, which are sequences of zero-crossings, the first of which is triggered by continuous evolution, whereas the others are discrete zero-crossings. Example 11.1 contains such a zero-crossing cascade, which is commented in §11.1 point 4.

Principle of translation. The translation that we propose applies the same principle as above to encode the history of the execution into locations (using discrete state variables).

We explain it using the "contact" semantics (again without inputs and logical combinations of zero-crossings). We consider $\boldsymbol{s}' = \boldsymbol{\Phi}$ if $up(z)$ and we introduce a Boolean variable q^d, which holds at each step k the value of $z < 0$ at step $k-1$.

- The evolution of q^d is defined by the initial state $q^d = \mathsf{ff}$ and the relation $R^d = ((q^d)' = (z < 0))$;
- the condition $up(z_m)$ is translated to the guard $G^d = (q^d \wedge z \geqslant 0)$;
- the global transition relation R is generated as in Eqn. (11.3).

Interrupting continuous evolution. The transitions as translated above are not

urgent, *i.e.*, the continuous states can evolve on intermediate states of a cascade. We need to prohibit this evolution explicitly if one of the discrete zero-crossings is activated. This is done by strengthening the global flow relation $V(\boldsymbol{q}, \boldsymbol{s}, \dot{\boldsymbol{x}})$ with $V' = V \wedge \boxslash G^d$.

In case of inputs, we have $G^d = (q^d \wedge z(\boldsymbol{s}, \boldsymbol{i}) \geqslant 0)$ and we take $V' = V \wedge \boxslash(\forall \boldsymbol{i} : G^d)$: the idea is that in a state $(\boldsymbol{q}, \boldsymbol{s})$, if the discrete zero-crossing is activated for any input (*i.e.*, $\forall \boldsymbol{i} : G^d$), then the continuous evolution is blocked. Otherwise, for some input, the discrete zero-crossing is not activated and the continuous evolution should be possible.

Remark 11.5 (Discrete/continuous zero-crossings) *A zero-crossings can be both discrete and continuous,* e.g., *$up(x+n)$. In this case it must be translated twice: once as a continuous zero-crossing and a second time as a discrete one. The resulting guard is the disjunction of both translations.*

Mind that a conjunction of a purely discrete and a purely continuous zero-crossing is not satisfiable.

Remark 11.6 (Compressing cascades) *Zero-crossing cascades can be "compressed" into a single discrete transition triggered by a continuous zero-crossing by* composing *the discrete transitions forming the cascade. This is possible if there are no instantaneous cyclic dependencies between the variables. The advantage of this kind of pre-processing is that the translation does not have to deal with discrete zero-crossings. However, care must be taken w.r.t. safety verification, because this transformation does not preserve the set of reachable states (it removes intermediate states).*

Remark 11.7 (Discrete zero-crossings in system modeling) *The main use of discrete zero-crossings is to trigger a cascade of several discrete transitions by a single continuous zero-crossing. For this purpose, discrete zero-crossings enable very concise modeling, but the resulting behavior can be very hard to understand, and thus, they can easily result in undesirable behavior. On the other hand, if a language does not support discrete zero-crossings, the programmer has to program such a behavior himself by manually composing transitions explicitly, as mentioned in Remark 11.6.*

11.3.5 The Complete Translation

We give here the formulas for the complete translation of a hybrid data-flow program

$$\mathcal{I}(\boldsymbol{s}) \quad , \quad \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \left\{ \begin{array}{l} \cdots \\ \dot{\boldsymbol{x}} = \left\{ \begin{array}{l} \boldsymbol{e}_l(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \phi_l(\boldsymbol{b}) \\ \cdots \end{array} \right. \\ \cdots \\ \boldsymbol{s}' = \left\{ \begin{array}{l} \boldsymbol{\Phi}_j(\boldsymbol{s}, \boldsymbol{i}) \text{ if } \varphi_j^Z(\boldsymbol{s}, \boldsymbol{i}) \\ \cdots \end{array} \right. \end{array} \right.$$

as defined in §11.1 into a hybrid automaton by combining all the concepts presented in §§11.3.1 to 11.3.4.

We use the notation ζ_m^σ to denote the constraints induced by a zero-crossing $up(z_m)$, *e.g.*, $\zeta_m^{\cdot=0} = \big(z_m = 0\big)$ or $\zeta_m^{0<\cdot\leqslant\epsilon} = \big(0 < z_m \leqslant \epsilon\big)$.

Discrete zero-crossings. For each discrete zero-crossing $up(z_m)$ we introduce a Boolean state variable q_m^d and define its transition relation R_m^d and guard G_m^d:

$R_m^d = (q'^d_m = \zeta_m^\sigma)$
$G_m^d = (q_j^d \wedge \zeta_m^{\overline{\sigma}})$

	σ	$\overline{\sigma}$
"contact"	$\cdot < 0$	$\cdot \geqslant 0$
"crossing"	$\cdot \leqslant 0$	$\cdot > 0$

Continuous zero-crossings. For each continuous zero-crossing $up(z_m)$ we introduce a state variable q^c_m.

– Their transition relations are defined as follows:

	θ	σ
"contact"	$\cdot < 0$	$\cdot = 0$
"crossing"	$\cdot = 0$	$0 \leqslant \cdot \leqslant \epsilon$

$$\begin{aligned} R^c_m = \ & \textbf{match } q^c_m \textbf{ with} \\ & \mathsf{above} \to {q'}^c_m \in \{\mathsf{above}, \mathsf{below}\} \\ & \mathsf{below} \to ((q^c_m)' = q^c_m) \ \vee \ ((q^c_m)' = \mathsf{ready}) \wedge \zeta^\theta_m \\ & \mathsf{ready} \to (q^c_m)' \in \{\mathsf{above}, \mathsf{below}\} \wedge \zeta^\sigma_m \ \vee \ ((q^c_m)' = q^c_m) \end{aligned}$$

– The guards G^c_m are defined as

	σ
"contact"	$\cdot = 0$
"crossing"	$0 \leqslant \cdot \leqslant \epsilon$

$$G^c_m = (q^c_m = \mathsf{ready}) \wedge \zeta^\sigma_m$$

– Using $\psi = \bigvee_l \big(\phi_l(\boldsymbol{b}) \wedge \dot{\boldsymbol{x}} = \boldsymbol{e}_l(\boldsymbol{s}, \boldsymbol{i})\big)$ we define the partial flow relations (containing the staying conditions):

	σ
"contact"	$\cdot \leqslant 0$
"crossing"	$0 \leqslant \cdot \leqslant \epsilon$

$$\begin{aligned} V_m = \ & \textbf{match } q^c_m \textbf{ with} \\ & \mathsf{above} \to \begin{cases} \big(\exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta^{\cdot \geqslant 0}_m \wedge \psi\big) \wedge \\ \big(\boxslash \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta^{\cdot \leqslant 0}_m \big) \end{cases} \\ & \mathsf{below} \to \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta^{\cdot \leqslant 0}_m \wedge \psi \\ & \mathsf{ready} \to \exists \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \zeta^\sigma_m \wedge \psi \end{aligned}$$

Transition relations. We define $G_m = G^c_m \vee G^d_m$ and $R_m = R^c_m \wedge R^d_m$. Now, we can finally put things together and define the jump and flow transition relations:

$$R((\boldsymbol{q}, \boldsymbol{s}), (\boldsymbol{q}', \boldsymbol{s}')) = \exists \boldsymbol{i} : \begin{cases} \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \big(\bigwedge_m R_m\big) \wedge \\ \Big(\neg\big(\bigvee_j H_j\big) \wedge \boldsymbol{s}' = \boldsymbol{s} \ \vee \\ \quad \bigvee_j \big(H_j \wedge \boldsymbol{s}' = \boldsymbol{\Phi}_j\big)\Big) \end{cases}$$

$$V((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) = \begin{cases} \bigvee_m V_m((\boldsymbol{q}, \boldsymbol{s}), \dot{\boldsymbol{x}}) \wedge \\ \quad \boxslash\big(\forall \boldsymbol{i} : \mathcal{A}(\boldsymbol{s}, \boldsymbol{i}) \wedge \bigvee_j H^d_j\big) \end{cases}$$

with $H_j = \varphi^Z_j[\forall m : up(z_m) \leftarrow G_m]$.

We obtain a hybrid automaton $\langle \{\ell_0\}, F, J, \Sigma^0 \rangle$ with

– $F(\ell_0) = V$,
– $J = \{(\ell_0, R, \ell_0)\}$, and
– $\Sigma^0(\ell_0) = \{(\boldsymbol{q}, \boldsymbol{s}) \mid \boldsymbol{q}^d = \mathbf{ff} \wedge \bigwedge_m q^c_m \in \{\mathsf{above}, \mathsf{below}\} \wedge \mathcal{I}(\boldsymbol{s})\}$

Theorem 11.1 (Sound translation) *All executions of a hybrid data-flow program are simulated by an execution of its translation to a logico-numerical hybrid automaton,* i.e.*, the translation is a sound over-approximation.*

Proofs. The proofs are based on demonstrating that the translation induces a left-to-right simulation relation $\precsim$ between the hybrid data-flow formalism (HDF) and logico-numerical hybrid automata (HA). We sketch here only the scheme for the inductive construction of this simulation relation following the semantics of the hybrid data flow formalism as illustrated below (the thick dots symbolize the configurations of an execution).

Base case: The initial state of HDF is included in the initial state of HA.	$\underbrace{\bullet}_{\models \mathcal{I}}$

Then, we split the induction step in several sub-steps:

Induction step: In a given state, every trajectory compliant to a *flow* equation in HDF *up to a zero-crossing* must be included in the set of trajectories compliant to the flow relation in HA up to the staying condition. (A corner case is that there is no zero-crossing in the future.)	$\bullet \rightarrow^c \ldots \rightarrow^c \underbrace{\bullet \rightarrow^c \bullet}_{\models \varphi_j^Z}$
If a *continuous zero-crossing* is activated in HDF then the guard of the corresponding jump transition in HA is satisfied.	$\underbrace{\bullet \rightarrow^c \bullet}_{\models \varphi_j^Z}$
If a *discrete zero-crossing* is activated in HDF then the guard of the corresponding jump transition in HA is satisfied.	$\underbrace{\bullet \rightarrow^d \bullet}_{\models \varphi_j^Z}$
The resulting state of a *discrete transition* in HDF is included in the set of states resulting from taking the corresponding jump transition in HA.	$\bullet \rightarrow^d \underbrace{\bullet}_{\boldsymbol{s}' = \boldsymbol{\Phi}_j(\boldsymbol{s})}$

The details of the proofs of the translation can be found in [SJ12c].

11.4 Discussion

We have presented the complete translation of a hybrid data-flow specification to a hybrid automaton. However, further pre-processing steps are necessary to enable verification using classical hybrid analysis methods.

Explicit representation. As explained in §11.3.3, we have chosen to present our translation by encoding the locations of the HA with N additional finite-state variables $\boldsymbol{q}$. This results in a HA with a single location and a single self-loop jump transition. Of course, it is possible to expand this "compressed" representation into a more explicit one, such as shown in Figs. 11.5 and 11.7. This is done by enumerating the valuations of these finite-state variables and by partitioning the system into these $\mathcal{O}(2^N)$ states. As already mentioned in §11.1, partial evaluation may be used to simplify expressions and to remove infeasible jump transitions.

Non-convex staying conditions and guards. The induced staying conditions $C(\boldsymbol{s})$ and guards $G(\boldsymbol{s})$ of jump transitions might be non-convex w.r.t. numerical constraints. However, most hybrid verification methods require convex staying conditions and guards. We will deal with this issue when developing logico-numerical hybrid verification methods in §13.

Approximations during analysis. In §11.3.1 we have explained that the translation to hybrid automata loses several properties, like determinism and urgency, which may result in an over-approximation in terms of reachable states. Moreover, hybrid reachability analysis methods further approximate the reachable states with (finite disjunctions of) convex sets, such as convex polyhedra.

The translation with "contact" semantics involves strict inequalities. Thus, the analysis may benefit from the ability to represent open sets. In this case, a suitable abstract domain might be convex polyhedra with strict inequalities [BHZ05]. Otherwise, if the analysis can only handle closed sets, the translation with "contact" semantics (Fig. 11.6b) will behave like the one for "at-zero" semantics (Fig. 11.5c).

Experiments. We have implemented the translation in our tool ReaVer (§14). Preliminary experiments have confirmed our intuition that the major parameter affecting verification efficiency is the number of zero-crossings. This becomes apparent when making locations explicit. In the applications that we are targeting, *i.e.*, synchronous controllers connected to their physical environment, zero-crossings are (1) those used for modeling the sampling of inputs and (2) those in the environment model. Since the number of (1) is usually small, the total number of zero-crossings inherently depends on the complexity of the environment model in practice.

11.5 Conclusions

We have presented a complete translation of a hybrid data-flow formalism to logico-numerical hybrid automata. In comparison with previously proposed translations, our translation handles zero-crossings. Moreover, we have proved that it is sound w.r.t. the semantics of the source language.

To achieve this, we considered a simple yet expressive hybrid data-flow formalism to which large subsets of existing hybrid system languages can actually be reduced.

We discussed different choices of zero-crossing semantics and their possible translations to hybrid automata. Since hybrid automata are not as expressive as the source language, we can only provide sound over-approximations of the original semantics.

However, this enables the use of existing hybrid verification tools such as HyTech [HHWT97], PHaver [Fre05] and SpaceEx [FGD+11], which are all based on the standard hybrid automaton model.

Yet, these tools require to encode Boolean variables explicitly in locations. As this enumeration results in an exponential blow-up of the hybrid automaton size, we assume that this is a major bottleneck in verifying controllers with complex discrete state spaces jointly with their physical environment. Therefore, §13 will propose methods for combining existing hybrid system analysis with implicit handling of Boolean variables in order to counter state space explosion. Our translation to logico-numerical hybrid automata lays the basis for such an approach.

Chapter 12

Hybrid System Verification

Before we propose methods for analyzing logico-numerical hybrid automata in §13, we give an overview of existing verification methods for numerical hybrid systems in this chapter.

In addition to handling discrete transitions, the *reachability analysis* of hybrid automata (§12.1) requires computing the states reachable by continuous evolution.

A vast variety of continuous reachability methods has been proposed. We will concentrate on *unbounded-time methods* (§12.2) which are based on abstract interpretation with widening or strategy iteration. In contrast, *bounded-time methods* (§12.3) have their origin in model-checking and set-based simulation.

At last, we list some *alternative approaches* (§12.4) which are not necessarily based on the computation of the reachable state space.

12.1 Reachability Analysis of Hybrid Automata

The reachability analysis of hybrid automata by abstract interpretation is formulated as the following fixed point equation over the power domain $L \to A$:

$$S^\sharp = S^{\sharp 0} \sqcup \lambda\ell \,.\, \left(\mathsf{post}^\sharp_{V_\ell}(S^\sharp_\ell) \sqcap C_\ell\right) \sqcup \lambda\ell \,.\, \left(\bigsqcup_{\ell' \in L} \mathsf{post}^\sharp_{R_{\ell',\ell}}(S^\sharp_{\ell'}) \sqcap C_\ell\right)$$

where $S^\sharp, S^{\sharp 0} \in (L \to A)$.

The discrete post-conditions $\mathsf{post}^\sharp_R$ of the jump transitions is computed in the same way as for discrete systems. The difference to discrete systems is the continuous post-condition $\mathsf{post}^\sharp_V$ of a flow transition, which requires continuous reachability methods.

Continuous reachability. Most continuous reachability methods have their origin in set-based simulation: they seek a precise over-approximation of trajectories emanating from a given set X_0 following a given dynamics. Since these trajectories (for non-constant dynamics) form non-convex sets, disjunctions of convex sets, *e.g.*, polyhedra, are often used. Different methods for computing such approximations have been developed according to the dynamics, which we categorize roughly in piece-wise constant, piece-wise linear, and non-linear.

Time-horizon and convergence. Set-based simulation methods consider the analysis up to a bounded time horizon. Convergence ($X_{k+1} \sqsubseteq X_k$) is detectable if the

system is stable (*e.g.*, constant or bounded asymptotic behavior). If the system is meta-stable (*e.g.*, limit cycle) then inclusion is only guaranteed if no over-approximation is added over one period. For convergence of unstable systems, extrapolation (widening) is needed. Since standard widening operators extrapolate linearly, they yield very imprecise results for systems with non-constant dynamics.

We are interested in unbounded time analysis (§12.2) provided by methods like *relational abstractions* that, similarly to abstract acceleration, try to find better extrapolations than standard widening, and very recently, *strategy iteration*-based methods that are able to compute the best inductive invariants for systems with linear dynamics using template polyhedra.

Decidability Results. Hybrid automata are infinite state systems, hence the reachability problem is undecidable in general. Nevertheless, there are classes of hybrid automata with restricted forms of transitions such that the reachability problem is decidable [HKPV98]:

– *Timed automata*: flow transitions of the form $\dot{\boldsymbol{x}} = \mathbf{1} \wedge C$ and jump transitions of the form $G \wedge \boldsymbol{x}' = \boldsymbol{f}(\boldsymbol{x})$ where C and G are conjunctions of constraints of the form $x_i \leqslant c$ or $x_i - x_j \leqslant c$, and $f_i = x_i$ or $f_i = 0$.
– *Initialized rectangular hybrid automata*: flow transitions of the form $\boldsymbol{a} \leqslant \dot{\boldsymbol{x}} \leqslant \boldsymbol{b} \wedge \boldsymbol{c} \leqslant \boldsymbol{x} \leqslant \boldsymbol{d}$ and jump transitions $\boldsymbol{g} \leqslant \boldsymbol{x} \leqslant \boldsymbol{h} \wedge \boldsymbol{j} \leqslant \boldsymbol{x}' \leqslant \boldsymbol{k}$. "Initialized" means that a jump transition $\boldsymbol{g} \leqslant \boldsymbol{x} \leqslant \boldsymbol{h} \wedge \boldsymbol{x}' = \boldsymbol{x}$ is only allowed if the flow transitions in origin and destination locations are equal.

[HKPV98] prove that dropping any of the restrictions (variables decoupled in dynamics, staying condition and guards, and "initialization") makes reachability undecidable.

[AMP95] show that the reachability problem for hybrid automata with flow transitions $\dot{\boldsymbol{x}} = \boldsymbol{c} \wedge \mathbf{A}\boldsymbol{x} \leqslant \boldsymbol{b}$ and jump transitions $\mathbf{G}\boldsymbol{x} \leqslant \boldsymbol{h} \wedge \boldsymbol{x}' = \boldsymbol{x}$ is decidable in two dimensions but undecidable in three dimensions.

12.2 Unbounded-Time Analysis Methods

Generalizing timed automata, early methods for hybrid systems deal with piece-wise constant dynamics, which have linear analytic solutions. Thus, methods for computing the continuous post-condition ("*time-elapse*", §12.2.1) are very close to discrete program analysis with polyhedra. Unbounded time is achieved with the help of widening.

Hybrid max-strategy iteration (§12.2.2) extends the max-strategy iteration approach (§3.4.3) for computing invariants over template polyhedra to hybrid automata with linear dynamics.

Relational abstractions (§12.2.3) transform a hybrid system into a discrete one by approximating the effect of the continuous evolution by a discrete transition.

We will explain these methods in detail because we are going to use them in §12.

12.2.1 Polyhedral Time-Elapse

Halbwachs et al [HRP94, HPR97] use abstract interpretation with convex polyhedra for analyzing hybrid systems with flow relations of the form of piecewise constant differential inclusions, *i.e.*,

$$V(\boldsymbol{x}, \dot{\boldsymbol{x}}) = \{(\boldsymbol{x}, \dot{\boldsymbol{x}}) \mid \dot{\boldsymbol{x}} \in D \wedge \boldsymbol{x} \in C\}$$

where D and C are convex polyhedra.

Then the set of reachable states by continuous evolution up to the staying condition C starting from a convex polyhedron X_0 can be easily computed as:

$$X = (X_0 \nearrow D) \sqcap C$$

where $X_0 \sqsubseteq C$.

This formula is very similar to the case of polyhedral translations $C \to \boldsymbol{x}' = \boldsymbol{x} + D$ in abstract acceleration, except that, in the discrete case, we had to account for the fact that the number of iterations is integral. Since the behavior of the considered systems is linear, standard widening with polyhedra performs reasonably well and enables an unbounded-time analysis.

12.2.2 Hybrid Max-Strategy Iteration

Dang and Gawlitza [DG11b, DG11a] formulate the reachability analysis of hybrid systems with linear dynamics in the framework of max-strategy iteration (see §3.4.3) with linear templates.

Definition 12.1 (Positive invariant) *Let* $\mathbf{T}$ *be a linear template.* $S = \{\boldsymbol{x} \mid \mathbf{T}(\boldsymbol{x}) \leqslant \boldsymbol{d}\}$ *is a positive invariant w.r.t the affine vector field* $V : \mathbb{R}^n \to \mathbb{R}^n$ *iff*

$$\forall j : \forall \boldsymbol{x} : \mathbf{T}_j.\boldsymbol{x} = d_j \wedge \bigwedge_{i \neq j} \mathbf{T}_i.\boldsymbol{x} \leqslant d_i \Longrightarrow \mathbf{T}_j.V(\boldsymbol{x}) < 0$$

i.e., *on the boundaries of* S *all vectors of* V *point inside* S.

A positive invariant is an inductive invariant.

The continuous post-condition $\mathsf{post}_V(X_0)$ associated with the flow relation V is the least positive invariant w.r.t. V starting from X_0. post_V must be encoded as the fixed point of a monotonic, concave operator f in order to be integrated into the max-strategy iteration framework. f is defined as follows:

$$f(\boldsymbol{d}) = \boldsymbol{d} + \epsilon \cdot \Delta(\boldsymbol{d})$$

with the j^{th} component of the vector Δ defined as:

$$\Delta_j(\boldsymbol{d}) = \sup\{\mathbf{T}_j\dot{\boldsymbol{x}} \mid \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d} \ \wedge \ \mathbf{T}_j\boldsymbol{x} \geqslant d_j \ \wedge \ (\boldsymbol{x}, \dot{\boldsymbol{x}}) \in V\}$$

and a "small enough" ϵ, such that f is monotonic on $\overline{\mathbb{R}}^m$ (see [DG11a] for the conditions on ϵ).

Moreover, the closure $cl(\boldsymbol{d})$ of an abstract value $\boldsymbol{d}$ w.r.t. the staying condition C (of which the constraints must be contained in the template) is defined as:

$$cl_j(\boldsymbol{d}) = \sup\{\mathbf{T}_j\boldsymbol{x} \mid \mathbf{T}\boldsymbol{x} \leqslant \boldsymbol{d} \wedge \boldsymbol{x} \in C\}$$

Then the continuous evolution up to the staying condition is given by

$$\mathsf{post}_V(X_0) \sqcap C = \Big(\big(lfp\lambda X.X \sqcup (cl \circ f)(X)\big) \circ cl\Big)(X_0)$$

Fig. 12.1 illustrates this encoding, which results in a discrete CFG.

The corresponding semantic inequations are then solved by max-strategy iteration.

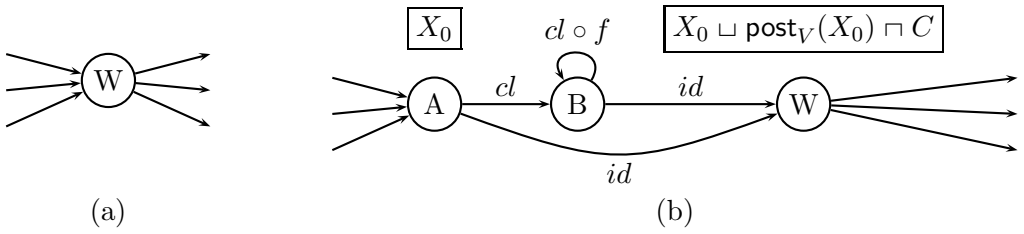


Figure 12.1: Discrete encoding (b) of the flow relation V in (a).

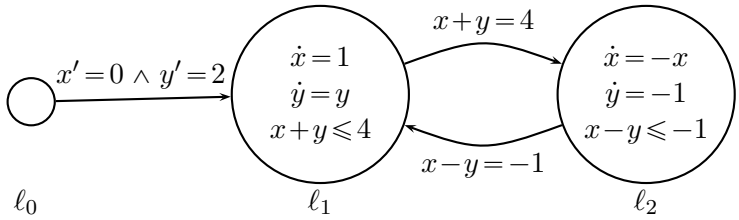


Figure 12.2: Hybrid max-strategy iteration: Ex. 12.1

Example 12.1 (Hybrid max-strategy iteration) *We analyze the hybrid automaton in Fig. 12.2: We choose the template constraints* $(-x, x+y, x-y)$ *and* $\epsilon = 10^{-3}$. *Notice that the locations* ℓ_1 *and* ℓ_2 *are actually replaced by the structure according to Fig. 12.1b. We denote the bound variables associated to these locations with the corresponding subscripts A, B, W. For example,* $\boldsymbol{\delta}_{1A}$ *denotes the bound variable vector corresponding to the location A in the structure replacing* ℓ_1.

Let us assume that we have already performed the first iteration having improved the strategies w.r.t. $\boldsymbol{\delta}_{1A}$ *and that we obtain the valuation:*

$$\boldsymbol{d}_1 = \left\{ \begin{array}{l} \boldsymbol{\delta}_0 \to \infty \\ \boldsymbol{\delta}_{1A} \to (0,2,-2) \qquad \boldsymbol{\delta}_{1B} \to -\infty \qquad \boldsymbol{\delta}_{1W} \to -\infty \\ \boldsymbol{\delta}_2 \to -\infty \end{array} \right\}$$

In the second step $(0,2,-2)$ *will be propagated to* ℓ_{1B} *and* ℓ_{1W} *– it is inside the staying condition, thus* $cl(\cdot)$ *has no effect. In the third step we can improve the strategies w.r.t.* $\boldsymbol{\delta}_{1B}$*:*

$$\mu_3 = \left\{ \begin{array}{ll} \boldsymbol{\delta}_0 \geqslant \infty & \boldsymbol{\delta}_2 \geqslant -\infty \\ \boldsymbol{\delta}_{1A} \geqslant \sup\left\{ \mathbf{T}\begin{pmatrix} x' \\ y' \end{pmatrix} \mid \mathbf{T}\begin{pmatrix} x \\ y \end{pmatrix} \leqslant \boldsymbol{\delta}_0 \wedge (x,y)' = (0,2) \right\} & \\ \delta_{1B} \geqslant cl(\boldsymbol{\delta}_{1B} + \epsilon \cdot \Delta^{V_1}(\boldsymbol{\delta}_{1B})) & \boldsymbol{\delta}_{1W} \geqslant \boldsymbol{\delta}_{1A} \end{array} \right\}$$

where $\Delta_j^{V_1}(\boldsymbol{\delta}_{1B}) = \sup\left\{ \mathbf{T}_j \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \mid \mathbf{T}\begin{pmatrix} x \\ y \end{pmatrix} \leqslant \boldsymbol{\delta}_{1B} \wedge \mathbf{T}_j \begin{pmatrix} x \\ y \end{pmatrix} \geqslant \delta_{1Bj} \wedge \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} 1 \\ y \end{pmatrix} \right\}$. *We thus get the valuation:*

$$\boldsymbol{d}_3 = \left\{ \begin{array}{l} \boldsymbol{\delta}_0 \to \infty \\ \boldsymbol{\delta}_{1A} \to (0,2,-2) \qquad \boldsymbol{\delta}_{1B} \to (0,4,-2) \qquad \boldsymbol{\delta}_{1W} \to -\infty \\ \boldsymbol{\delta}_2 \to -\infty \end{array} \right\}$$

In the next steps $(0,4,-2)$ *will be propagated (by strategy improvement) to* ℓ_{1W}, ℓ_{2A},

ℓ_{2B} and ℓ_{2W}. Then we can improve the strategy w.r.t. $\boldsymbol{\delta}_{2B}$:

$$\mu_7 = \left\{ \begin{array}{lll} \boldsymbol{\delta}_0 \geqslant \infty & & \\ \boldsymbol{\delta}_{1A} \geqslant (0,2,-2) & \delta_{1B} \geqslant cl(\boldsymbol{\delta}_{1B} + \epsilon \cdot \Delta^{V_1}(\boldsymbol{\delta}_{1B})) & \boldsymbol{\delta}_{1W} \geqslant \delta_{1B} \\ \boldsymbol{\delta}_{2A} \geqslant \sup\{\mathbf{T}\begin{pmatrix} x' \\ y' \end{pmatrix} \mid \mathbf{T}\begin{pmatrix} x \\ y \end{pmatrix} \leqslant \boldsymbol{\delta}_{1W} \wedge x+y=4\} & & \\ & \boldsymbol{\delta}_{2B} \geqslant cl(\boldsymbol{\delta}_{2B} + \epsilon \cdot \Delta^{V_2}(\boldsymbol{\delta}_{2B})) & \boldsymbol{\delta}_{2W} \geqslant \boldsymbol{\delta}_{2A} \end{array} \right\}$$

which results in the valuation:

$$\boldsymbol{d}_7 = \left\{ \begin{array}{lll} \boldsymbol{\delta}_0 \to \infty & & \\ \boldsymbol{\delta}_{1A} \to (0,2,-2) & \boldsymbol{\delta}_{1B} \to (0,4,-2) & \boldsymbol{\delta}_{1W} \to (0,4,-2) \\ \boldsymbol{\delta}_{1A} \to (0,4,-2) & \boldsymbol{\delta}_{1B} \to (0,4,-1) & \boldsymbol{\delta}_{1W} \to (0,4,-2) \end{array} \right\}$$

$(0,4,-1)$ will be propagated to ℓ_{2W}, ℓ_{1A}, ℓ_{1B}, ℓ_{1W} and ℓ_{2A}. At this point no more strategy improvement is possible, i.e.*, we have reached the least fixed point, with the valuation:*

$$\boldsymbol{d}_{10} = \left\{ \begin{array}{l} \boldsymbol{\delta}_0 \to \infty \\ \boldsymbol{\delta}_{1\cdot} \to (0,4,-1) \\ \boldsymbol{\delta}_{2\cdot} \to (0,4,-1) \end{array} \right\}$$

As with all template-based methods, the precision crucially depends on the choice of the template. Also the computed invariants are inductive: some dynamics like rotations have no inductive invariants w.r.t. linear templates. The advantage of the technique in comparison with discretization-based methods is that the analysis is time-unbounded and thus the computation time is independent from the time horizon.

12.2.3 Relational Abstractions

Several methods for turning hybrid automata into discrete transition systems have been considered since the early times of hybrid automata based on finite bisimulation abstractions [Hen95, CK01, ADI02, GT08].

A recent method are *relational abstractions* [Tiw03, Tiw08, ST11, ZST12]. The idea is very similar to loop acceleration (§4): the flow transition is replaced by a discrete self-loop transition labeled with the jump relation $R(\boldsymbol{x}, \boldsymbol{x}')$, *i.e.*, a relation between values entering a location $\boldsymbol{x}$ and all values reachable by the flow relation $\boldsymbol{x}'$.

Definition 12.2 (Relational abstraction) *[ST11] $R \subseteq \mathbb{R}^{2n}$ is a relational abstraction if for all trajectories $\tau : [0,\delta) \to \mathbb{R}^n$ it is the case that $\forall t \in [0,\delta) : (\tau(0), \tau(t)) \in R$.*

A relational abstraction is *complete*, if, whenever $R(\boldsymbol{x}, \boldsymbol{x}')$ holds, then there is a trajectory τ from $\boldsymbol{x}$ to $\boldsymbol{x}'$.

Relational abstractions for certain dynamics of hybrid automata can be given right away [ST11]:

– Piecewise constant dynamics ($\dot{\boldsymbol{x}} = \boldsymbol{d}$ with $d_i \neq 0$):

$$R(\boldsymbol{x}, \boldsymbol{x}') = \bigwedge_{i=2}^{n} \left(\frac{x_1' - x_1}{d_1} = \frac{x_i' - x_i}{d_i} \right) \wedge \left(\frac{x_1' - x_1}{d_1} \geqslant 0 \right)$$

This abstraction is complete. A special case are timed automata for which $\boldsymbol{d} = \mathbf{1}$.

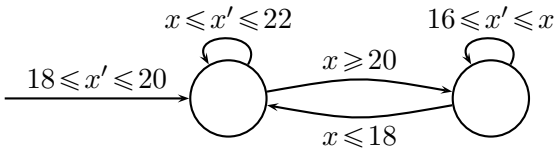


Figure 12.3: Relational abstraction (Ex. 12.2) of the hybrid automaton in Fig. 10.3

– Piecewise constant inclusions ($\boldsymbol{C}\dot{\boldsymbol{x}} \leqslant \boldsymbol{d}$): by rewriting as $\boldsymbol{A}\dot{\boldsymbol{x}} \leqslant \boldsymbol{b} \wedge \boldsymbol{A}'\dot{\boldsymbol{x}} \geqslant \boldsymbol{b}'$ with $\boldsymbol{b} > 0$ and $\boldsymbol{b}' > 0$, ./ is the component-wise division):

$$R(\boldsymbol{x}, \boldsymbol{x}') = 0 \leqslant \max(\mathbf{A}(\boldsymbol{x}' - \boldsymbol{x})./\boldsymbol{b}) \leqslant \min(\mathbf{A}'(\boldsymbol{x}' - \boldsymbol{x})./\boldsymbol{b}')$$

  This abstraction is complete. A special case are rectangular hybrid automata for which $\mathbf{A} = \mathbf{A}' = \mathbf{I}$.
– Linear dynamics ($\dot{\boldsymbol{x}} = \mathbf{C}\boldsymbol{x}$) have (in general incomplete) abstractions depending on the matrix $\mathbf{C}$:
  – If $\mathbf{C} = \mathit{diag}(\lambda_1, \ldots, \lambda_n)$ then $R(\boldsymbol{x}, \boldsymbol{x}') = \exists t \geqslant 0 : \bigwedge_j x'_j = e^{\lambda_j t} x_j$. In practice, the existential quantification is performed approximately over linear templates, or the relational abstraction is constructed by splitting cases w.r.t. the values of λ_i (*e.g.*, asymptotic behavior).
  – If $\mathbf{C}$ is diagonizable with rational eigenvalues, then one can perform a basis change to reduce it to the previous case.
  – If $\mathbf{C}$ is nilpotent, then one can obtain a complete abstraction [ST11].

Moreover, a single flow transition can be abstracted by a logical combination of relational abstractions.

Example 12.2 (Relational abstraction) *We relationalize the hybrid automaton in Fig. 10.3: In the first mode we have the dynamics* $30 \leqslant \dot{x} - x \leqslant 40$ *with the staying condition* $x \leqslant 22$*: Exploiting the information about the asymptotic behavior of the solution we get the abstraction*

$$\begin{aligned} R_1(x, x') &= \big((x < 40 \Rightarrow x \leqslant x' < 40) \vee (x > 30 \Rightarrow 30 < x' \leqslant x)\big) \wedge x \leqslant 22 \\ &= x \leqslant x' \leqslant 22 \end{aligned}$$

Analogously, we get for the second mode $R_2(x, x') = 16 \leqslant x' \leqslant x$*. Fig. 12.3 shows the resulting discrete CFG.*

This method represents a lightweight way to derive useful continuous invariants in some practical cases. Furthermore, the whole arsenal of existing discrete methods can be employed for analysis. It is implemented in the tool HybridSAL[1].

12.3 Bounded-Time Analysis Methods

Initially, bounded model checking techniques for hybrid automata considered systems piecewise-constant dynamics that can be dealt with using the polyhedral time-elapse. More complicated dynamics require a time-discretization. Hence, such methods actually perform a set-based simulation.

[1] `http://sal.csl.sri.com/hybridsal/`

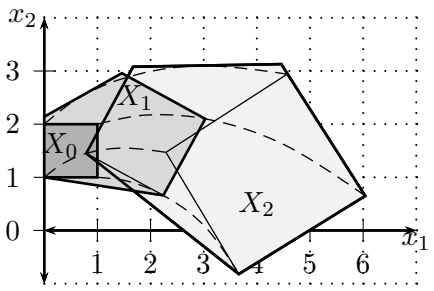


Figure 12.4: Flow pipe approximation of the set of trajectories starting from X_0: the segment X_1 over-approximating time interval $[0, \delta]$, X_2 over-approximating $[\delta, 2\delta]$; dashed... trajectories emanating from the vertices of X_0.

HyTech and PHAVer HyTech[2] was the first reachability analysis tool for hybrid systems [HH94, Ho95, HHWT97, HHWT98] and it had a major impact on subsequent developments. It considers systems with (or approximated by) constant differential inclusions and performs a polyhedral analysis (see §12.2.1) without widening.

PHAVer[3] [Fre05] improves HyTech in several aspects, *inter alia*, it uses state space partitioning based on a user-provided set of hyperplanes (predicates) and over-approximates on-the-fly linear dynamics by piecewise constant dynamics.

Set-based simulation. The idea of set-based simulation is to divide time into intervals $[0, t_1]$, $[t_1, t_2]$,... and enclose the trajectories in each interval by, *e.g.*, a polyhedron. The resulting disjunction of sets is called a *flow-pipe* (see Fig. 12.4). Starting from an initial set X_0 each integration step with a time step δ adds a new segment X_k to the flow-pipe. These techniques generalize guaranteed integration [Löh88, NJC99, Bou08] from enclosing intervals to relational domains.

Academic verification tools for linear differential inclusions based on this technique are for example CheckMate[4] [CK98, CK99, CK03] and d/dt[5] [DM98, ABDM00]. Besides, *zonotopes* [Gir05] and *ellipsoids* [KV00, BT00] have been considered for such computations, too.

A certain effort has been made in order to scale up the flow pipe computation while keeping an acceptable level of precision (*e.g.*, [GM99], [SK03]). A breakthrough has been achieved by Le Guernic and Girard [GG09] with their double representation of convex sets using template polyhedra and *support functions* (see *e.g.*, [BV04]). This method is implemented in the tool SpaceEx[6] [FGD+11], which can handle systems with more than 100 variables.

For linear systems, the flow-pipe segments are computed based on the analytic solution. However, this is no more possible in the case of non-linear systems. *Hybridization* [ADG07, DGM09] techniques enable to exploit the efficiency of linear system methods with the help of state space partitioning and linearization of the non-linear dynamics.

[2] `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`
[3] `http://www-verimag.imag.fr/~frehse/phaver_web/`
[4] `http://www.mathworks.com/matlabcentral/fx_files/15441/3/content/doc/main.htm`
[5] `http://www-verimag.imag.fr/~tdang/Tool-ddt/ddt.html`
[6] `http://spaceex.imag.fr`

12.4 Alternative Approaches

Other approaches include methods involving level sets, satisfiability solving, quantifier elimination and theorem proving.

Level set methods. Using level sets for proving safety properties is similar to Lyapunov functions for proving stability: a function f must be found of which the zero level set $\{\boldsymbol{x} \mid f(\boldsymbol{x}) = 0\}$ separates safe and unsafe states. f is constructed by formulating a sum of squares problem and solving by semidefinite programming [PJ04], or from the Hamilton-Jacobi differential equation associated with the level set function (see [Tom98, MT00] for details). These methods provide unbounded time verification for arbitrary dynamics, although their scalability is limited to at most 5 dimensions.

Constraint propagation and SAT-based approaches. HySAT [FH07, FHT+07] is a bounded model checker (BMC, see §3.1) for logico-numerical hybrid systems (see §11.2) with piece-wise constant dynamics. The BMC problem for a hybrid automaton is encoded as the SAT-problem of a formula involving constraints. Inside the DPLL procedure, the satisfiability of constraints is checked using an LP solver, arithmetic inference methods are employed to derive new facts about linear relations, and the repetitive structure of the BMC formula is exploited to prune the search space.

iSAT [EFH08, ERNF11] provides BMC for hybrid systems with arbitrary dynamics. It is based on a Boolean SAT-solver and interval constraint propagation. It uses a guaranteed interval ODE solver for the enclosure of trajectories. Due to these over-approximations, iSAT can only prove unsatisfiability. The advantage of the tool is that it can deal with logico-numerical hybrid systems with non-linear dynamics. The drawbacks are that an interval analysis yields quite coarse results and that it is limited to bounded time analysis.

HSolver [RS07] combines interval constraint propagation with a grid-based abstraction refinement technique.

Other recent techniques like [GT08] rely on generic SMT solvers to compute template-based invariants: *i.e.*, the hybrid automaton and the property are translated into a big $\exists\forall$ formula. Then the least positive invariant w.r.t. the vector field is computed by mapping rational numbers to bounded-range integers and solving the system by a bit-vector decision procedure.

Differential dynamic logic. A completely different approach is differential dynamic logic [Pla08], which is first-order real arithmetic equipped with embedded dynamic behavior, *i.e.*, ODEs, and modal operators "for all states" and "exists a state" for defining properties. A system is proved with the help of a sequent calculus based on symbolic decomposition according to proof rules and real quantifier elimination. The method is implemented in the semi-automatic tool KeYmaera [PQ08] which combines a deductive theorem prover, a computer algebra system, and differential induction [Pla10].

The advantages are unbounded time verification for systems with arbitrary dynamics and the fact that it is not restricted to safety properties. The drawbacks are that the size of the proof may explode for systems with a complex logical structure and that user intervention is required when the theorem prover fails to find a proof automatically.

Chapter 13

Analysis of Logico-Numerical Hybrid Automata

The problem of analyzing logico-numerical hybrid automata bears much resemblance to the analysis of discrete logico-numerical programs: we have analysis methods for numerical systems available, but we cannot directly apply them to a logico-numerical system because the transition relations are formulas mixing Boolean and numerical variables. Also, we want to avoid enumerating the Boolean states by using logico-numerical analysis methods instead.

Hence, in this chapter, we will adapt and apply the basic concepts that we developed in the context of discrete logico-numerical systems to the analysis of hybrid logico-numerical systems.

The contributions of this chapter can be summarized as follows, see Fig. 13.1:

1. In §13.1 we propose techniques for *adapting existing numerical hybrid analysis methods to logico-numerical systems* and we apply them to three unbounded-time analysis methods.

2. Regarding state space partitioning, a good candidate for equivalence classes which we want to identify in hybrid systems are the continuous modes, *i.e.*, distinct continuous behaviors. In §13.2 we propose two *methods for discovering continuous modes* within a logico-numerical hybrid system.

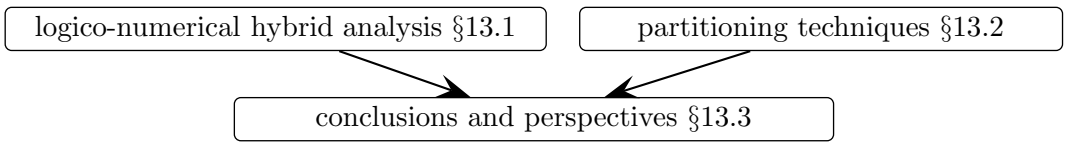


Figure 13.1: Chapter organization

13.1 Logico-Numerical Hybrid Analysis Methods

The flow relations in a logico-numerical hybrid automaton are general logico-numerical formulas, but most existing continuous reachability methods require convex numerical flow relations.

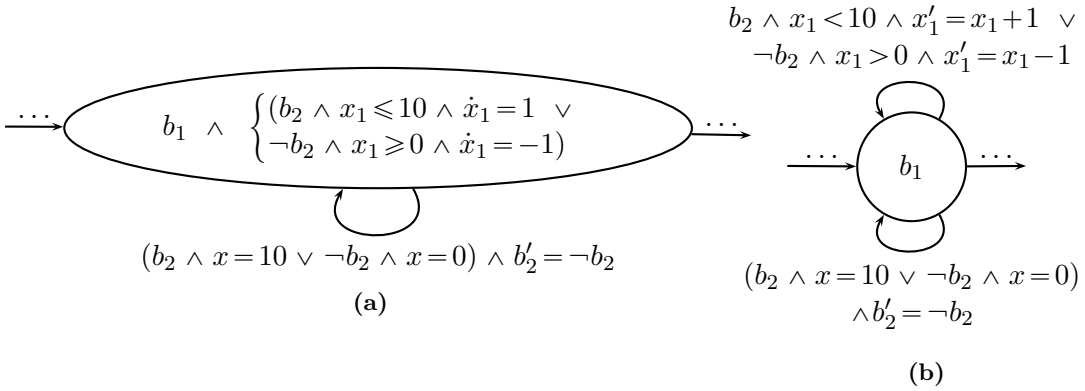


Figure 13.2: Analogies between a logico-numerical hybrid automaton (a) and a CFG of a logico-numerical program (b).

Since a flow transition can be somehow considered as a self-loop (see Fig. 13.2), the problem resembles the one of logico-numerical abstract acceleration (§8). But actually, it is even simpler, because only continuous (numerical) variables evolve in flow transitions, while the discrete (Boolean and numerical) variables are known to stay unmodified – whereas in discrete self-loops, both, Boolean and numerical variables evolve "hand-in-hand". Therefore, decoupling Boolean and numerical variables is not necessary.

We describe first the principles (13.1.1) for adapting numerical hybrid analysis methods to logico-numerical hybrid automata. Then we will show how to apply these principles to three selected hybrid analysis methods (13.1.2): polyhedral time elapse, hybrid max-strategy iteration, and relational abstractions.

13.1.1 Basic Techniques

Structure of abstract domains. Logico-numerical hybrid automata have the state space $\mathbb{B}^m \times \mathcal{N}^p \times \mathbb{R}^n$ with the discrete state space $\mathbb{B}^m \times \mathcal{N}^p$, where $\mathcal{N}^p$ may be *e.g.*, $\mathbb{Z}^q \times \mathbb{R}^{p-q}$, and the continuous state space $\mathbb{R}^n$.

We abstract all numerical variables together in the same abstract domain. Hence, we get the structure of logico-numerical domains for discrete programs (see §7.2.3) with the product domain $\wp(\mathbb{B}^m) \times A(\mathbb{R}^{p+n})$ and the power domain $\mathbb{B}^m \to A(\mathbb{R}^{p+n})$, where A is an abstract domain, *e.g.*, convex polyhedra or template polyhedra.

This choice enables us to establish relations between discrete and continuous numerical variables:

Example 13.1 (Discrete/continuous relations)

```
let hybrid dcrel () = (x,n) where
  rec der t = 1 reset 0 every (up(10-t)) init 0
  and der x = 2 init 0
  and n = (last n+1) every (up(10-t)) init 0
```

Using the techniques described below, we can prove the mixed discrete/continuous property: $20n \leqslant x \leqslant 20(n+1)$.

We propose an on-the-fly computation of a numerical flow-relation that can be processed by a continuous reachability method:

Specializing the flow relation. The first step is the partial evaluation (specialization) of the flow relation by the current reachable states in the respective location. For this purpose, we compute the current state $S^{\textit{zero-dynamics}}$ of all variables $\boldsymbol{s}^{\textit{zero-dynamics}}$ which are constant during the flow transition of the location: these are all the discrete variables and those continuous variables the derivatives of which evaluate to zero: $S^{\textit{zero-dynamics}} = \exists \boldsymbol{s}^{\textit{non-zero-dynamics}} : S$.

Example 13.2 (Specializing the flow relation) *We explain this procedure for the flow relation*

$$V = \big((b_1 \wedge b_2 \ \wedge\ \dot{x}_1 = x_2 - x_1 \ \wedge\ \dot{x}_2 = n-1) \ \vee\ (b_1 \wedge \neg b_2 \ \wedge\ \dot{x}_1 = 1 \ \wedge\ \dot{x}_2 = x_1)\big)$$

and the current state $S = \big(b_2 \ \wedge\ n = 1 \ \wedge\ 0 \leqslant x_1 \ \wedge\ 0 \leqslant x_2 \ \wedge\ x_1 + x_2 \leqslant 4\big)$*. In order to find out which variables have non-zero dynamics, we simplify the flow relation by the discrete states* $S^{\textit{discrete}} = \exists \boldsymbol{x} : S = (b_2 \ \wedge\ n = 1)$*:*

$$\exists b_1, b_2, n : V \uparrow S^{\textit{discrete}} = (\dot{x}_1 = x_2 - x_1 \ \wedge\ \dot{x}_2 = 0)$$

Hence, the only variable with non-zero dynamics is x_1*:*

$$S^{\textit{zero-dynamics}} = (\exists x_1 : S) = (b_2 \ \wedge\ n = 1 \ \wedge\ 0 \leqslant x_2 \leqslant 4)$$

and we can compute the flow relation specialized w.r.t. the current reachable states:

$$V \uparrow S^{\textit{zero-dynamics}} = (b_1 \ \wedge\ 0 \leqslant \dot{x}_1 + x_1 \leqslant 4 \ \wedge\ \dot{x}_2 = 0)$$

Extracting convex flow relations. The second step is to extract convex numerical flow relations. There are two opposite approaches: (1) enumerating and splitting disjunctions or (2) convexifying disjunctions.

Method (1) involves splitting $V(\boldsymbol{s}, \dot{\boldsymbol{x}})$ into a disjunction of numerically convex formulas $\bigvee_j V_j(\boldsymbol{s}, \dot{\boldsymbol{x}}) = \mathsf{decomp\_convex}(V)$ (see §7.2.2), and then computing the continuous post-condition by $\bigsqcup_j \mathsf{post}_{V_j}$.

Example 13.3 (Splitting disjunctions) *The flow relation*

$$\begin{array}{l}(b_1 \vee \neg b_2) \wedge (x \leqslant 0 \vee x \geqslant 5) \wedge (\dot{x} = 1 - x) \ \vee \\ (\neg b_1 \wedge b_2) \wedge (0 \leqslant x \leqslant 5) \wedge (\dot{x} = 1)\end{array}$$

is decomposed into three numerically convex disjuncts

$$\begin{array}{l}(b_1 \vee \neg b_2) \wedge (x \leqslant 0) \wedge (\dot{x} = 1 - x) \ \vee \\ (b_1 \vee \neg b_2) \wedge (x \geqslant 5) \wedge (\dot{x} = 1 - x) \ \vee \\ (\neg b_1 \wedge b_2) \wedge (0 \leqslant x \leqslant 5) \wedge (\dot{x} = 1)\end{array}$$

The less precise but also less expensive technique is (2), *i.e.*, to take the convex hull of the flow relation $V' = \mathsf{cvx}(V)$.

Example 13.4 (Convexifying disjunctions) *For the flow relation in Ex. 13.3 we obtain the convexified flow relation* $1 \leqslant \dot{x} + x \leqslant 6$*.*

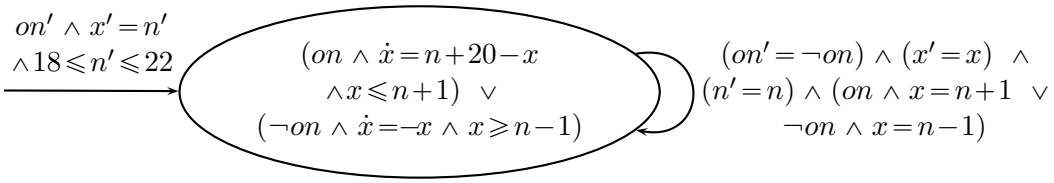


Figure 13.3: Logico-numerical polyhedral time-elapse Ex. 13.5

13.1.2 Application to Numerical Hybrid Analysis Methods

We show the application of these principles to three hybrid system analysis techniques:

Polyhedral Time-elapse

The polyhedral time elapse method (§12.2.1) computes the reachable states by a flow transition with the constant differential inclusion $\dot{\boldsymbol{x}} \in D$ and the staying condition C by $(X \nearrow D) \sqcap C$.

We can apply the convexification method to logico-numerical flow transitions by computing $D = \mathsf{cvx}(\exists \boldsymbol{x} : V')$ and $C = \mathsf{cvx}(\exists \dot{\boldsymbol{x}} : V')$ where

$$V' = \exists \boldsymbol{b} : \big(V(\boldsymbol{b}, \boldsymbol{x}, \dot{\boldsymbol{x}}) \uparrow (\exists \boldsymbol{x}^{\textit{non-zero-dynamics}} : S)\big).$$

Splitting disjunctions means applying the computation above to each disjunct in $\bigcup_i V_i = \mathsf{decomp\_convex}(V)$ and computing the continuous post-condition for each pair (D_i, C_i), *i.e.*, $\bigsqcup_i (X \nearrow D_i) \sqcap C_i$.

Example 13.5 (Logico-numerical polyhedral time-elapse) *We analyze the thermostat system with setpoint n shown in Fig. 13.3 using the logico-numerical product domain and the splitting method:*

The initial state w.r.t. variables with zero-dynamics is $S_0^{\textit{zero-dynamics}} = (on \wedge 18 \leqslant n \leqslant 22)$. Thus, in the first iteration the flow relation simplifies to the first disjunct: $V_1 = \big(on \ \wedge \ \dot{x} = n+20-x \ \wedge \ x \leqslant n+1\big)$. We compute

$$V_1' = \big(\exists n, on : on \ \wedge \ \dot{x} = n+20-x \ \wedge \ x \leqslant n+1 \ \wedge \ 18 \leqslant n \leqslant 22\big) = \big(38 \leqslant \dot{x}+x \leqslant 42 \ \wedge \ x \leqslant 23\big)$$

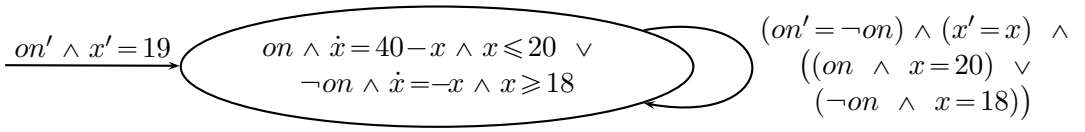
By existential quantification of x and $\dot{x}$ respectively, we get $D_1 = (19 \leqslant \dot{x})$ and $C_1 = (x \leqslant 23)$. The result after the flow transition is $S_1 = (on \ \wedge \ 18 \leqslant n \leqslant 22 \ \wedge \ 18 \leqslant x \leqslant 23)$ and after the discrete transition $S_2 = (18 \leqslant n \leqslant 22 \ \wedge \ 18 \leqslant x \leqslant 23)$.

In the next iteration we have $S_1^{\textit{zero-dynamics}} = (18 \leqslant n \leqslant 22)$, thus, the whole flow relation is active and we decompose it into the two disjuncts $V_1 \vee V_2$. D_1 and C_1 do not change, for $V_2 = (\neg on \ \wedge \ \dot{x} = -x \ \wedge \ x \geqslant n-1)$ we obtain the polyhedra $D_2 = (\dot{x} \leqslant -17)$ and $C_2 = (17 \leqslant x)$.

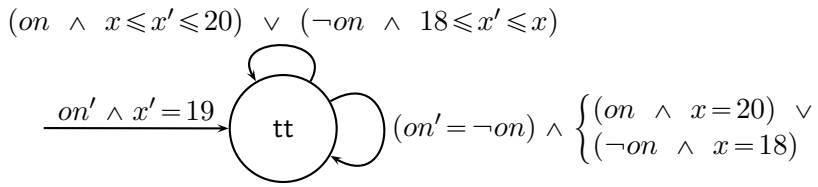
The flow transition results in $S_3 = (18 \leqslant n \leqslant 22 \ \wedge \ 17 \leqslant x \leqslant 23)$, which is a fixed point.

Relational Abstractions

Relational abstractions (§12.2.3) over-approximate the flow relation by a discrete transition, and thus, they transform a hybrid automaton into a discrete CFG.



(a) Hybrid automaton.



(b) Relationalized (discrete) system.

Figure 13.4: Logico-numerical relationalization Ex. 13.6

The extension to logico-numerical hybrid automata is straightforward. However, in order to enable the reuse of existing discrete analysis tools, it is not possible to simplify the flow relation on-the-fly. So, we have to relationalize the whole hybrid automaton before analyzing the obtained discrete system.

Using the splitting technique, the continuous part of each disjunct is relationalized separately. The convexification method, *i.e.*, relationalizing the convex hull, is very likely to lose important information, because it is applied without any knowledge about reachable states.

Example 13.6 (Logico-numerical relationalization) *We analyze the thermostat system in Fig. 13.4a): We split* $V = V_1 \vee V_2$ *with* $V_1 = on \wedge \dot{x} = 40 - x \wedge x \leqslant 20$ *and* $V_2 = on \wedge \dot{x} = -x \wedge x \geqslant 18$*. We compute the relational abstraction of* V_1 *by exploiting the asymptotic behavior of the dynamics:*

$$\begin{aligned} R_1 &= \big((x < 40 \Rightarrow x \leqslant x' < 40) \ \vee \ (x > 40 \Rightarrow 40 < x' \leqslant x)\big) \wedge on \wedge x \leqslant 20 \wedge x' \leqslant 20 \\ &= on \ \wedge \ x \leqslant x' \leqslant 20 \end{aligned}$$

In the same way we obtain for V_2*:* $R_2 = (\neg on \ \wedge \ 18 \leqslant x' \leqslant x)$*. The resulting discrete CFG is shown in Fig. 13.4b. The analysis using the logico-numerical product domain yields the invariant* $18 \leqslant x \leqslant 20$*.*

Using the convexification method, we obtain $\mathsf{cvx}(V) = (0 \leqslant \dot{x} + x \leqslant 40)$ *and after relationalization* $R' = (x < 0 \Rightarrow x \leqslant x' < 40) \ \vee \ (0 \leqslant x \leqslant 40 \Rightarrow 0 \leqslant x' \leqslant 40) \ \vee \ (x > 40 \Rightarrow 0 < x' \leqslant x)$*. Here, the convexification loses the important constraints of the staying condition, thus the analysis yields the weaker invariant* $0 \leqslant x \leqslant 40$*.*

Max-Strategy Iteration

We extend the logico-numerical max-strategy iteration (§9) to hybrid systems using the encoding of flow transitions as explained in §12.2.2.

As in §9, we use the *splitting* method for decomposing numerically non-convex transitions into several strategies. This results in an encoding of a flow relation according to the automaton scheme of Fig. 12.1, but with one self-loop per disjunct in location B.

The number of strategies can be reduced by using the convexification technique, which generates only a single continuous strategy per flow transition.

Example 13.7 (Hybrid logico-numerical max-strategy iteration) *We analyze the thermostat system in Fig. 13.4a using an interval template $\pm x$ in the product domain $\mathbb{B}^n \times Pol^{\mathbf{T}}(\mathbb{R})$.*

We start in the state $(on, x) = (\mathtt{tt}, [19, 19])$. No discrete transition which changes a Boolean state can be taken. We switch to phase (2): there are three available strategies associated to the transitions $(x' = 19)$, $\big((\dot{x} = 40 - x) \ \wedge \ x \leqslant 20\big)$ and $(x' = x)$. We can improve w.r.t. the upper bound of x by using the second strategy and obtain $(\mathtt{tt}, [19, 20])$. No more improvement is possible. We continue with phase (1) and take the discrete transition that results in $(\top, [19, 20])$. Then, in phase (2), four strategies are available: the three above and $\big((\dot{x} = -x) \ \wedge \ x \geqslant 18\big)$. We can improve the lower bound of x by the latter one and obtain $(\top, [18, 20])$, which is a fixed point.

13.2 Partitioning Techniques for Hybrid Systems

In logico-numerical hybrid automata, a flow relation is a logico-numerical formula containing the staying condition and differential inclusions. One location can regroup many distinct continuous behaviors, and thus, the analysis over a CFG consisting of a single location is not very precise.

As for discrete systems, our approach is to increase the precision by partitioning the system into locations such that distinct continuous behaviors are associated with separate locations.

We propose two partitioning methods for discovering these continuous modes: one detects modes defined by equivalence classes of *Boolean* states (§13.2.1), whereas the other one performs an analysis to detect modes defined by equivalence classes of *discrete numerical* states (§13.2.2).

13.2.1 Boolean-Defined Continuous Modes

The first method we describe is the hybrid analog of the method described in §8.3 to find the Boolean states that imply the same numerical actions (Boolean-defined "discrete numerical modes"). Now, we apply it to the flow relation in order to detect the Boolean states with the same continuous dynamics.

We represent flow relations $V(\boldsymbol{b}, \boldsymbol{x}, \dot{\boldsymbol{x}})$ with the help of flow functions of the form $\dot{x}_i = f_i^{cx}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi})$ (where f_i^{cx} is an arithmetic expression with tests, see §7.2.2), such that, together with the assertion $\mathcal{A}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})$, we obtain semantically $V(\boldsymbol{b}, \boldsymbol{x}, \dot{\boldsymbol{x}}) = \exists \boldsymbol{\beta}, \boldsymbol{\xi} : \mathcal{A} \wedge \bigwedge_i (\dot{x}_i = f_i^{cx})$.

f_i^{cx} is represented as an MTBDD (in the same way as discrete transition functions (see §7.2.2)). We denote $\boldsymbol{f}^{cx}$ the vectorized form with factorized guards (MTBDD product, §7.2.2) of the flow functions f_i^{cx}:

$$\boldsymbol{f}^{cx}(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C}, \boldsymbol{x}, \boldsymbol{\xi}) = \bigvee_j \big(\dot{\boldsymbol{x}} = \boldsymbol{e}_j(\boldsymbol{x}, \boldsymbol{\xi})\big) \wedge \phi_j(\boldsymbol{b}, \boldsymbol{\beta}, \mathcal{C})$$

Then a partition is characterized by the following equivalence relation (cf. Def. 8.3):

Definition 13.1 (Boolean states with the same set of ODEs)

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \left\{ \begin{array}{ll} \forall \boldsymbol{\beta}_1, \mathcal{C}_1 : & \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) \Rightarrow \\ & \exists \boldsymbol{\beta}_2, \mathcal{C}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \wedge \boldsymbol{f}^{cx}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}_1) = \boldsymbol{f}^{cx}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}_2) \\ \quad \textit{and} & \textit{vice versa} \end{array} \right.$$

The intuition of this heuristics is to make equivalent the Boolean states which imply to the same set of ODEs.

Example 13.8 (Boolean-defined modes I) *We compute the product of the flow relations*

$$\dot{x} = \left\{ \begin{array}{ll} 2-x & \textit{if } b_1 \wedge x \leqslant 1 \vee \neg b_2 \wedge x \geqslant 3 \\ -y & \textit{if } \neg b_1 \wedge b_2 \wedge x \geqslant 0 \wedge y \geqslant 0 \end{array} \right. \qquad \dot{y} = \left\{ \begin{array}{ll} 5-y & \textit{if } \neg b_2 \wedge y \leqslant 3 \\ -x & \textit{if } b_2 \wedge x \geqslant 0 \wedge y \geqslant 0 \end{array} \right.$$

and we obtain

$$(\dot{x}, \dot{y}) = \left\{ \begin{array}{ll} (2-x, 5-y) & \textit{if } \big((b_1 \wedge x \leqslant 1) \vee (\neg b_1 \wedge x \geqslant 3)\big) \wedge \neg b_2 \wedge y \leqslant 3 \\ (2-x, -x) & \textit{if } b_1 \wedge b_2 \wedge 0 \leqslant x \leqslant 1 \wedge y \geqslant 0 \\ (-y, -x) & \textit{if } \neg b_1 \wedge b_2 \wedge x \geqslant 0 \wedge y \geqslant 0 \end{array} \right.$$

Thus, we have the partition $\{\neg b_2, b_1 \wedge b_2, \neg b_1 \wedge b_2\}$*, in which each equivalence class corresponds to a distinct pair of ODEs.*

Variants. Analogously to Def. 8.2, we can formulate an equivalence relation identifying the Boolean states associated to the same set of ODEs *and* the same staying conditions:

Definition 13.2 (Boolean states with the same set of ODEs and stay. cond.)

$$\boldsymbol{b}_1 \sim \boldsymbol{b}_2 \Leftrightarrow \left\{ \begin{array}{ll} \forall \boldsymbol{\beta}_1, \mathcal{C} : & \mathcal{A}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) \Rightarrow \\ & \exists \boldsymbol{\beta}_2 : \mathcal{A}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \wedge \boldsymbol{f}^{cx}(\boldsymbol{b}_1, \boldsymbol{\beta}_1, \mathcal{C}) = \boldsymbol{f}^{cx}(\boldsymbol{b}_2, \boldsymbol{\beta}_2, \mathcal{C}) \\ \quad \textit{and} & \textit{vice versa} \end{array} \right.$$

Example 13.9 (Boolean-defined modes II) *Let's consider the first case in the product equation in Ex. 13.8:*

$$(\dot{x}, \dot{y}) = (2-x, 5-y) \textit{ if } \big((b_1 \wedge x \leqslant 1) \vee (\neg b_1 \wedge x \geqslant 3)\big) \wedge \neg b_2 \wedge y \leqslant 3$$

Requiring that the ODEs and *the staying conditions must be equal, we obtain two equivalence classes for this case:* $b_1 \wedge \neg b_2$ *and* $\neg b_1 \wedge \neg b_2$*. Hence the complete partition is* $\{b_1 \wedge \neg b_2, \neg b_1 \wedge \neg b_2, b_1 \wedge b_2, \neg b_1 \wedge b_2\}$*.*

13.2.2 Numerically Defined Continuous Modes

Often the continuous modes are characterized by the values of discrete numerical variables, like in the following example:

Example 13.10 (Partitioning by numerically defined modes I)

$$-5 \leqslant \xi \leqslant 5 \rightarrow \left\{ \begin{array}{lcl} \dot{x} & = & n-x \\ n' & = & \left\{ \begin{array}{ll} 40+\xi & \textit{if } up(18-x) \\ \xi & \textit{if } up(x-20) \end{array} \right. \end{array} \right.$$

This system has two modes $\dot{x} = 40 + \xi - x$ and $\dot{x} = \xi - x$ according to the values of n. Hence, we want to assign these two modes to distinct locations by considering the partition $\{35 \leqslant n \leqslant 45, -5 \leqslant n \leqslant 5\}$ of the reachable state space.

Thus, our goal is to find a finite partition of valuations of those discrete numerical variables occurring in the flow relation. We could detect such partitions syntactically by inspecting the transition relations. However, we can do better by performing the following analysis:

Analysis

The idea is to perform an interval analysis in an abstract domain that represents the reachable values of those variables as a disjunction of interval vectors for which the numerical transitions are guaranteed to generate only a finite set of intervals. The other variables are abstracted by a single interval vector as in the standard interval domain (§3.3.1).

We define the following abstract domain:

Finitely disjunctive, partitioned interval domain $FdpInt(\mathbb{R}^n)$**.** This domain partitions the set of variables in two partitions with q and $n-q$ variables respectively and abstracts them as follows:

$$\wp(\mathbb{R}^n) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \wp(Int(\mathbb{R}^q)) \times Int(\mathbb{R}^{n-q})$$

An abstract value $(P^J, I^K) \in FdpInt(\mathbb{R}^n)$ is a pair consisting of a finite set (disjunctions) P of interval vectors for the set of variables J ($|J| = q$), and one interval vector I for the set of variables K ($|K| = n-q$). J and K form a partition of the set of variables.

We list the domain operations in Tab. 13.1. We denote $\mathsf{cvx}(P)$ the convex hull operation which transforms a set of interval vectors P into a single interval vector. $\top$ and $\bot$ are represented as $(\varnothing, \top)$ and $(\varnothing, \bot)$ respectively.

The partition of variables $\{J, K\}$ changes dynamically according to the operations: For example, the resulting partition of a union of two abstract values with partitions $\{J_1, K_1\}$ and $\{J_2, K_2\}$ is $\{J_1 \cap J_2, K_1 \cup K_2\}$.

The post-condition (image) of a transition $\boldsymbol{f}$ gathers only the values of those variables into the disjunctive component $P^{J'}$ of the abstract value that result from so-called "finitely generating transitions" (defined below).

The image operator is defined as $[\![\boldsymbol{f}]\!](P^J, I^K) = (P^{J'}, I^{K'})$ with
- $P^{J'} = \times_i [\![f_i]\!](P^J, I^K)$ for those f_i which are finitely generating w.r.t. (P^J, I^K),
- $I^{K'} = \times_i [\![f_i]\!](\mathsf{cvx}(P^J), I^K)$ for those f_i which are not finitely generating w.r.t. (P^J, I^K).

Definition 13.3 (Finitely generating transitions) *A transition $f_i = \big(g(\boldsymbol{x}, \boldsymbol{\xi}) \rightarrow x_i' = a_i(\boldsymbol{x}, \boldsymbol{\xi})\big)$ is finitely generating w.r.t. (P^J, I^K) if it is*
(1) an assignment to a constant: $g(\boldsymbol{x}, \boldsymbol{\xi}) \rightarrow x_i' = d_i$, or
(2) an assignment to inputs: $g'(\boldsymbol{x}) \wedge g''(\boldsymbol{\xi}) \rightarrow x_i' = \mathbf{C}_i\boldsymbol{\xi} + d_i$, or

$$\text{Inclusion:} \quad (P_1^{J_1}, I_1^{K_1}) \sqsubseteq (P_2^{J_2}, I_2^{K_2}) \quad \Leftrightarrow \quad K_1 \subseteq K_2 \wedge \begin{cases} P_1^{J_2} & \subseteq & P_2^{J_2} \\ I_1^{K_1} & \sqsubseteq^{Int} & I_2^{K_1} \\ \mathsf{cvx}(P_1^{J_1 \cap K_2}) & \sqsubseteq^{Int} & I_2^{J_1 \cap K_2} \end{cases}$$

$$\begin{array}{lrcl} \text{Abstraction:} & \alpha(X) & = & (\alpha^{Int}(X), \bot) \\ \text{Concretization:} & \gamma(P, I) & = & \left(\bigcup_{p \in P} \gamma^{Int}(p)\right) \times \gamma^{Int}(I) \\ \text{Union:} & (P_1^{J_1}, I_1^{K_1}) \sqcup (P_2^{J_2}, I_2^{K_2}) & = & (P^{J_1 \cap J_2}, I^{K_1 \cup K_2}) \end{array}$$

$$\text{where} \begin{cases} P^{J_1 \cap J_2} & = & P_1^{J_1 \cap J_2} \cup P_2^{J_1 \cap J_2} \\ I^{K_1 \cup K_2} & = & \begin{pmatrix} I_1^{K_1 \cap K_2} & \sqcup^{Int} & I_2^{K_1 \cap K_2} \\ \mathsf{cvx}(P_1^{J_1 \cap K_2}) & \sqcup^{Int} & I_2^{J_1 \cap K_2} \\ I_1^{J_2 \cap K_1} & \sqcup^{Int} & \mathsf{cvx}(P_2^{J_2 \cap K_1}) \end{pmatrix} \end{cases}$$

$$\begin{array}{lrcl} \text{Image:} & [\![\boldsymbol{f}]\!](P^J, I^K) & = & (P^{J'}, I^{K'}) \text{ as defined in the text} \\ \text{Widening:} & (P_1, I_1) \nabla (P_2, I_2) & = & \begin{cases} \text{same as union,} \\ \text{but with } \nabla^{Int} \text{ instead of } \sqcup^{Int} \end{cases} \end{array}$$

Table 13.1: Operations of the finitely disjunctive, partitioned interval domain. The superscript *Int* indicates the operations of the standard interval domain.

(3) the identity or a "simple" variable exchange possibly with negation: $g'(\boldsymbol{x}^{\mathsf{nf}}, \boldsymbol{\xi}) \wedge g''(\boldsymbol{x}^{\mathsf{f}}, \boldsymbol{\xi}) \rightarrow x_i' = cx_j$ *with* $c \in \{-1, 1\}$ *and variables* $\boldsymbol{x}^{\mathsf{f}}, x_j$ *in* J *and variables* $\boldsymbol{x}^{\mathsf{nf}}$ *in* K*.*

Mind that case (2) requires that the inputs are not related to the state variables in the guard, and case (3) requires that the variables occurring on the right-hand side of the assignment are not related to variables in K via the guard. The reason for these requirements is the observation made in Prop. 5.1 that, otherwise, we could encode any (non-finitely generating) transition.

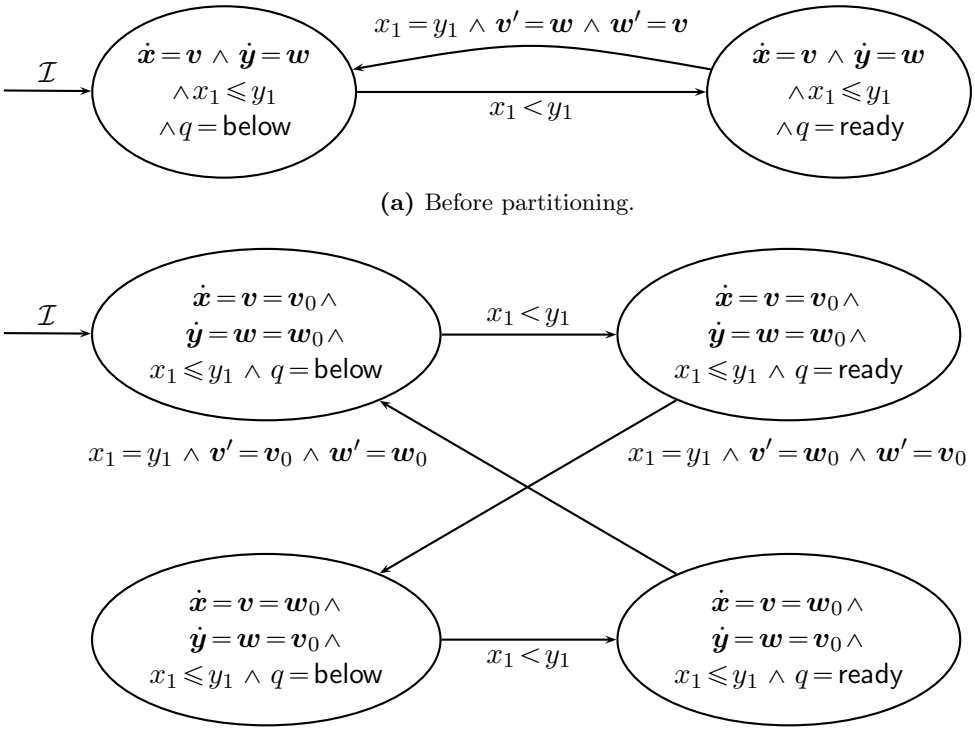
The conditions (1) to (3) assure that a transition generates a only finite number of intervals, however, they probably do not characterize the maximal set of such transitions.

The domain $\mathit{FdpInt}(\mathbb{R}^n)$ can be combined with Booleans in the usual way, resulting either in the power domain $\mathbb{B}^p \rightarrow \mathit{FdpInt}(\mathbb{R}^n)$ or the product domain $\wp(\mathbb{B}^p) \times \mathit{FdpInt}(\mathbb{R}^n)$. The behavior of continuous variables in a location is abstracted by the staying condition.

Partitioning

After having computed the fixed point with Kleene iteration, widening and descending iterations, for example in the domain $L \rightarrow \wp(\mathbb{B}^p) \times \mathit{FdpInt}(\mathbb{R}^n)$, we obtain for each location ℓ with abstract value $A_\ell = (B, P^J, I^K)$, a set $\Psi = \{(B, p, I^{K \cap U}) \mid p \in P^{J \cap U}\}$ where U are the numerical variables occurring in the flow relation V_ℓ. The elements of Ψ (viewed as predicates) are then used in the partitioning process (see §7.3.3). We apply this method to the following (purely numerical) example using the domain $\mathit{FdpInt}(\mathbb{R}^n)$:

Example 13.11 (Partitioning by numerically defined modes II) *We consider a system of two particles with positions* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *and velocities* $\boldsymbol{v}$ *and* $\boldsymbol{w}$ *respectively. With*



(a) Before partitioning.

(b) After partitioning.

Figure 13.5: Partitioning by numerically defined modes: Ex. 13.11 particle collision (The location corresponding to $q=\mathsf{above}$ is not depicted; it is not reachable anyway.)

the given initial state, they collide when $x_1=y_1$.

$$\mathcal{I}=\left(\boldsymbol{x}_0=\begin{pmatrix}0\\3\end{pmatrix}\wedge\boldsymbol{y}_0=\begin{pmatrix}6\\4\end{pmatrix}\wedge\boldsymbol{v}_0=\begin{pmatrix}2\\-3\end{pmatrix}\wedge\boldsymbol{w}_0=\begin{pmatrix}-4\\-4\end{pmatrix}\right)$$

$$\begin{cases}\dot{\boldsymbol{x}} &= \boldsymbol{v}\\ \dot{\boldsymbol{y}} &= \boldsymbol{w}\\ \boldsymbol{v}' &= \boldsymbol{w} \quad \textit{if } up(x_1-y_1)\\ \boldsymbol{w}' &= \boldsymbol{v} \quad \textit{if } up(x_1-y_1)\end{cases}$$

Fig. 13.5a depicts the hybrid automaton obtained by applying the translation of §11 and partitioning by q. Applying our analysis above, we get the following results in both locations:

$$\left(\underbrace{\begin{pmatrix}v_1\\v_2\\w_1\\w_2\end{pmatrix}}_{\in J},\underbrace{\begin{pmatrix}x_1\\x_2\\y_1\\y_2\end{pmatrix}}_{\in K}\right)\in\left(\underbrace{\left\{\begin{pmatrix}2\\-3\\-4\\-4\end{pmatrix},\begin{pmatrix}-4\\-4\\2\\-3\end{pmatrix}\right\}}_{P},\underbrace{\begin{pmatrix}\top\\\top\\\top\\\top\end{pmatrix}}_{I}\right)$$

After partitioning we obtain the hybrid automaton shown in Fig. 13.5b.

13.3 Conclusions and Perspectives

This chapter actually unified two extensions: on the one hand, we extended the logico-numerical analysis approach *from discrete to hybrid* systems, and on the other hand we extended hybrid analysis methods *from numerical to logico-numerical* hybrid systems.

In this spirit, we reused the technique for identifying modes according to equivalence classes of Boolean states, which we developed in the context of logico-numerical abstract acceleration, for detecting the continuous modes of a hybrid system. Besides, we complemented this technique by a method for detecting modes defined by numerical states, which is based on a particular disjunctive interval analysis.

Regarding analysis methods, we showed how to employ the basic techniques, for example to deal with non-convex conditions or to extract numerical relations from logico-numerical ones, for extending hybrid analysis methods from numerical to logico-numerical systems. We have already used these basic techniques in a similar way in the context of our discrete logico-numerical methods.

As a consequence of this unified approach, we could also use our analysis methods originally developed for discrete systems, logico-numerical abstract acceleration for instance, to treat the discrete transitions during the analysis of a hybrid system.

Perspectives. In the context of abstract interpretation, we are mainly interested in unbounded-time reachability. However, the principles of §13.1.1 are also applicable in the same manner to the set-simulation methods (§12.3) targeting bounded model-checking. Hence, they could also help to speed up these techniques when applied to logico-numerical systems.

Our techniques for turning numerical hybrid analysis methods into logico-numerical ones considered only the extreme techniques for convexifying flow transitions, *i.e.*, either splitting disjunctions or computing the convex hull: it should be investigated whether there are reasonable heuristics for an intermediate treatment allowing precision traded off for efficiency.

The abstract-interpretation-based analysis of logico-numerical hybrid systems in a more implicit way is a novel approach. In this work, we have shown its basic feasibility. However, further research and experimentation are necessary in order to assess and exploit the full potential of such methods.

Part IV

Implementation and Conclusion

Chapter 14

Implementation: The Tool ReaVer

We have implemented the verification methods developed throughout this thesis in a tool called ReaVer, REActive system VERifier.

In order to keep it as generic and extensible as possible, we separated the implementation into (1) a framework providing general facilities and defining generic concepts and (2) the implementations of these concepts making up the actual tool. The framework is built upon the logico-numerical abstract domain library BddApron. Implementations may additionally call other libraries, such as LP or SMT solvers. The tool is implemented using the OCaml language; the size of the code is approximately 4KLOC for (1) and 8KLOC for (2).

We will first briefly give an overview of the framework (§14.1) before describing which methods (§14.2) are implemented in the tool.

14.1 The Framework

Our framework provides three basic data structures:
- The *data-flow (DF) program* is the common intermediate representation of an input program. This representation is in fact the hybrid data-flow formalism (with zero-crossings) of §11.1, which subsumes the logico-numerical dynamical systems of §7.1.
- The *control-flow graph (CFG)* is the common representation used during analysis, which is actually a logico-numerical hybrid automaton (§11.2) subsuming discrete CFGs (§7.3.1).
- The *analysis result (AR)* holds for each CFG location the corresponding (logico-numerical) abstract value.

Moreover, our framework defines five interfaces for the operations and modules involved in the verification process:
- *Front ends* convert an input file into a DF program.
- *DF program to CFG transformations* convert a DF program into a CFG.
- *CFG transformations* transform a CFG.
- *Analyses* analyze a CFG and produce an analysis result.
- *Abstract domains* are used by the analyses.

The tool provides implementations of these interfaces and uses them to perform the flow of operations depicted in Fig. 14.1: The tool takes an input file and options, the *front*

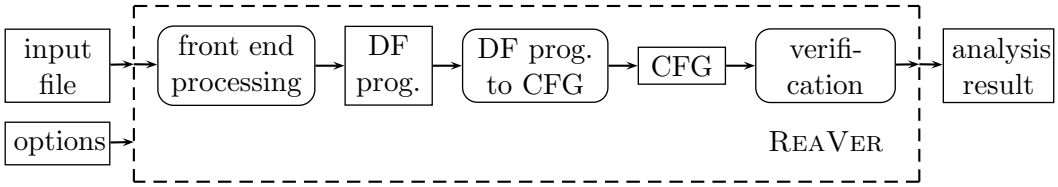


Figure 14.1: ReaVer: data and operation flow (rectangles... data, rounded rectangles... operations; see Fig. 14.2 for the internals of the verificaton block).

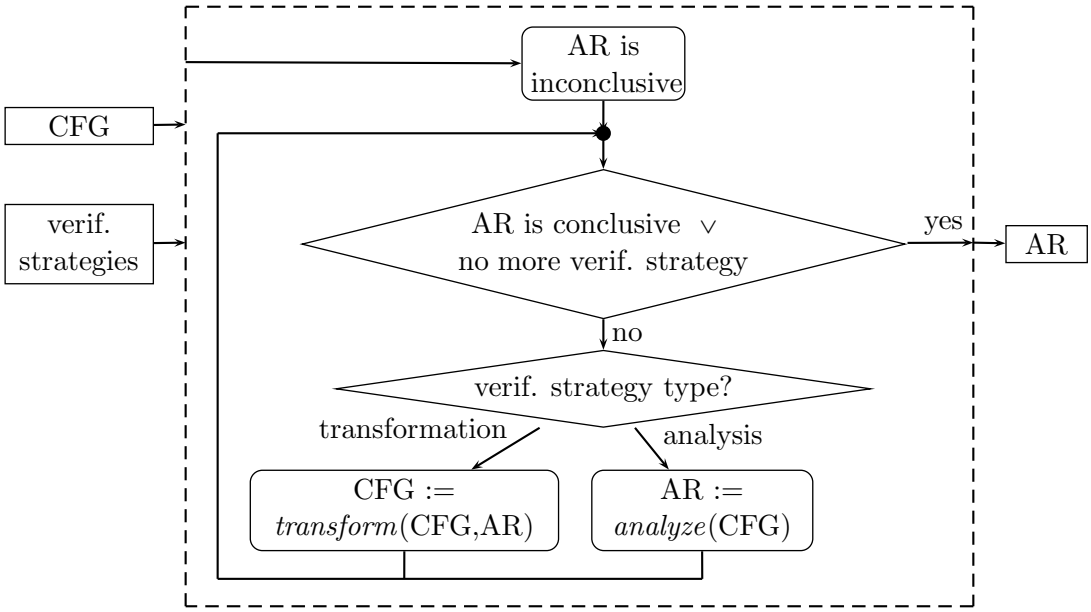


Figure 14.2: ReaVer: Zoom into the *verification* block of Fig. 14.1 (AR... analysis result).

end parses the input file and transforms the program do a *DF program*. The *DF program to CFG transformation* converts it into a *CFG*. Then the *verification block* (Fig. 14.2) performs the actual verification: it is given a list of *verification strategies* (via the tool options), *i.e.*, *CFG transformations* and *analyses*, and executes them sequentially until all verification strategies have been processed or a conclusive *analysis result* has been obtained.

The concept of verification strategies allows the user to compose analysis and partitioning methods in a flexible way. For example, we can start with a Boolean analysis, then we perform a first coarse partitioning followed by an interval analysis; then we refine the partition and finally, we apply some more expensive analysis, *e.g.*, using convex polyhedra.

The next section §14.2 gives an overview of the implementations of the five interfaces provided in ReaVer.

14.2 The Tool

The tool ReaVer[1] provides the following implementations of the framework's interfaces:

Front Ends

The tool supports the following input formats:
(1) NBac and Hybrid NBac formats,
(2) Lustre (after transformation into the NBac format using the tool Lus2NBac [Jea00]),
(3) subsets of Lucid Synchrone and Zelus.

NBac and Hybrid NBac. The NBac format is the input file format used by the tool NBac [Jea00]. It is a textual description of a discrete dynamical system (§2.1.1) together with a specification of a property $(\mathcal{A}, \mathcal{G})$. We have extended it with ODEs and a zero-crossing operator to the Hybrid NBac format.

The Hybrid NBac grammar is listed in Fig. 14.3; the expressions are those allowed by the BddApron library (see §7.2.2): besides the type definitions of enumerated types, the available types are `bool`, `real`, `int` and signed (`sint[`n`]`) and unsigned (`uint[`n`]` bounded integers represented by n bits.

The property is specified by the expressions following the keywords `assertion` $\mathcal{A}$ and `invariant` $\mathcal{G}$ (or alternatively by the error states: `final` $\mathcal{E}$).

Zelus and Lucid Synchrone. The front ends for a subset of Zelus (and Lucid Synchrone) are based on the front end of the Zelus compiler [BBCP11b] followed by a translation to the abstract syntax of the Hybrid NBac format. The top-level function must have two Boolean outputs `(assert,ok)`, which correspond to the two outputs $(\mathcal{A}, \mathcal{G})$ of the observers specifying the property.

By translating the abstract syntax of NBac/Hybrid NBac into BddApron formulas (§7.2.2) we obtain a *DF program*.

DF program to CFG

In case of a discrete program, the trivial *CFG* with a single self-loop (cf. §2.1.2) can be constructed from the DF program right away, whereas for hybrid programs with zero-crossings, we have to apply the *translation from hybrid data flow to logico-numerical hybrid automata* (§11).

Abstract Domains

BddApron provides an implementation of a *logico-numerical power domain* (§7.2.3) based on any numerical domain available in Apron, like convex polyhedra and octagons.

Based on this implementation, we provide the emulation of a *logico-numerical product domain* (§7.2.3) parametrized with any numerical domain available in Apron.

For experimental purposes, we have also implemented the product and power combination of a *logico-numerical linear template domain* emulated with the help of convex polyhedra, because template polyhedra are not yet available in Apron.

The partitioning by numerically defined continuous modes uses the (logico-numerical) *finitely disjunctive, partitioned interval domain* (§13.2.2).

[1] Current version ReaVer 0.9

⟨*prog*⟩ ::= [`typedef` ⟨*typedef*⟩$^+$] ⟨*vardecl*⟩ [`definition` ⟨*definition*⟩$^+$]
`transition` ⟨*transition*⟩$^+$
⟨*initial*⟩ [⟨*assertion*⟩] ⟨*invariant*⟩
⟨*typedef*⟩ ::= *type* `= enum{` ⟨*labels*⟩ `};`
⟨*labels*⟩ ::= *label* | *label* `,` ⟨*labels*⟩
⟨*vardecl*⟩ ::= `state` ⟨*varstype*⟩$^+$ [`input` ⟨*varstype*⟩$^+$] [`local` ⟨*varstype*⟩$^+$]
⟨*varstype*⟩ ::= ⟨*vars*⟩ `:` *type* `;`
⟨*vars*⟩ ::= *v* | *v* `,` ⟨*vars*⟩
⟨*definition*⟩ ::= *v* `=` ⟨*expr*⟩ `;`
⟨*transition*⟩ ::= ⟨*disctrans*⟩ | ⟨*conttrans*⟩
⟨*disctrans*⟩ ::= *v*`'` `=` ⟨*expr*⟩ `;`
⟨*conttrans*⟩ ::= `.`*v* `=` ⟨*expr*⟩ `;`
⟨*expr*⟩ ::= ⟨*BddApronExpr*⟩ | `up` ⟨*expr*⟩
⟨*initial*⟩ ::= `initial` ⟨*expr*⟩ `;`
⟨*assertion*⟩ ::= `assertion` ⟨*expr*⟩ `;`
⟨*invariant*⟩ ::= `invariant` ⟨*expr*⟩ `;` | `final` ⟨*expr*⟩ `;`

Figure 14.3: Hybrid NBac format.

CFG Transformations

These comprise *CFG preprocessing and simplifications* (§7.3.3, §8.1) as well as partitioning methods (and their variants) for discrete and hybrid programs:
– "manual" partitioning by a given set of predicates;
– partition by *initial* ($\mathcal{I}$), *error* ($\mathcal{E}$) and other ($\neg(\mathcal{I} \vee \mathcal{E})$) states (§7.3.2);
– *enumeration* of Boolean states (§7.3);
– partitioning by (Boolean-defined) *discrete numerical modes* (§8.3);
– partition refinement by *Boolean backward bisimulation* (§8.3);
– partitioning by *Boolean-defined continuous numerical modes* (§13.2.1);
– partitioning by *numerically defined continuous numerical modes* (§13.2.2).

Analyses

Besides a forward and backward *Boolean analysis* (§7.3.3), for discrete programs the following analyses are available:
– *logico-numerical standard analysis* (cf. §7.2.3, §7.3.1) in forward and backward direction parametrized with a logico-numerical product or power domain with a numerical domain from the Apron library;
– forward *logico-numerical abstract acceleration* (§8.2) with convex polyhedra in logico-numerical product or power combination;
– forward *logico-numerical max-strategy iteration* (§9.2) using a logico-numerical product or power domain with template polyhedra.

For hybrid programs, the following analyses are available (§13.1.2):
– forward *logico-numerical polyhedral time-elapse* parametrized with the logico-numerical product or power domain with convex polyhedra;

– forward *logico-numerical hybrid max-strategy iteration* using the logico-numerical product or power domain with template polyhedra;
– forward *relational abstractions* (currently restricted to abstractions not requiring a basis change) as preprocessing for a discrete analysis.

The purely numerical versions of the analyses (including backward abstract acceleration (§5.2)) are applicable after proper preprocessing of the CFG, *i.e.*, enumeration of the Boolean states.

The max-strategy iteration-based methods call the solver of Gawlitza [DG11a] (which is based on the LP solver QSopt_ex[2] and the SMT solver Yices[3]).

14.2.1 Analysis Example with ReaVer

In order to illustrate the basic output of the analyzer, we analyze the following small Lucid Synchrone program:

```
let node main i = (assert,ok) where
  rec assert = true
  and ok = true fby (ok && -10<=x && x<=10)
  and x = 0 fby (if i then -x else if x<=9 then x+1 else x)
```

We launch the analyzer with the command `reaver example.ls` and we get the output (compressed):

```
[0.020] INFO [Main] ReaVer, version 0.9.0
[0.028] INFO [Main] variables(bool/num):  state=(2/1), input=(1/0)
[0.038] INFO [Verif] CFG (3 location(s), 3 arc(s)):
LOC -1:  arcs(in/out/loop)=(0,1,0), def = init
LOC -3:  arcs(in/out/loop)=(1,0,0), def = not init and not p1_
LOC -4:  arcs(in/out/loop)=(1,1,1), def = not init and p1_
[0.039] INFO [Verif] analysis 'forward analysis with abstract acceleration'
[0.070] INFO [VerifUtil] analysis result:
LOC -1:  reach =  (init) and top
LOC -3:  reach = bottom
LOC -4:  reach =  (not init and p1_) and [|-p2_+10>=0; p2_+10>=0|]
[0.074] INFO [Main] variable mapping:
"p2_" in File "example.ls", line 4, characters 17-55:
>   and x = 0 fby (if i then -x else if x<=9 then x+1 else x)
>                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
"p1_" in File "example.ls", line 3, characters 21-42:
>   and ok = true fby (ok && -10<=x && x<=10)
>                       ^^^^^^^^^^^^^^^^^^^^
[0.075] INFO [Main] PROPERTY TRUE (final unreachable)
```

This tells us that:
– The program has two Boolean state variables and one numerical state variable and one Boolean input variable.
– After partitioning, the CFG has three locations with the displayed location definitions.

[2] http://www.dii.uchile.cl/~daespino/ESolver_doc/main.html
[3] http://yices.csl.sri.com

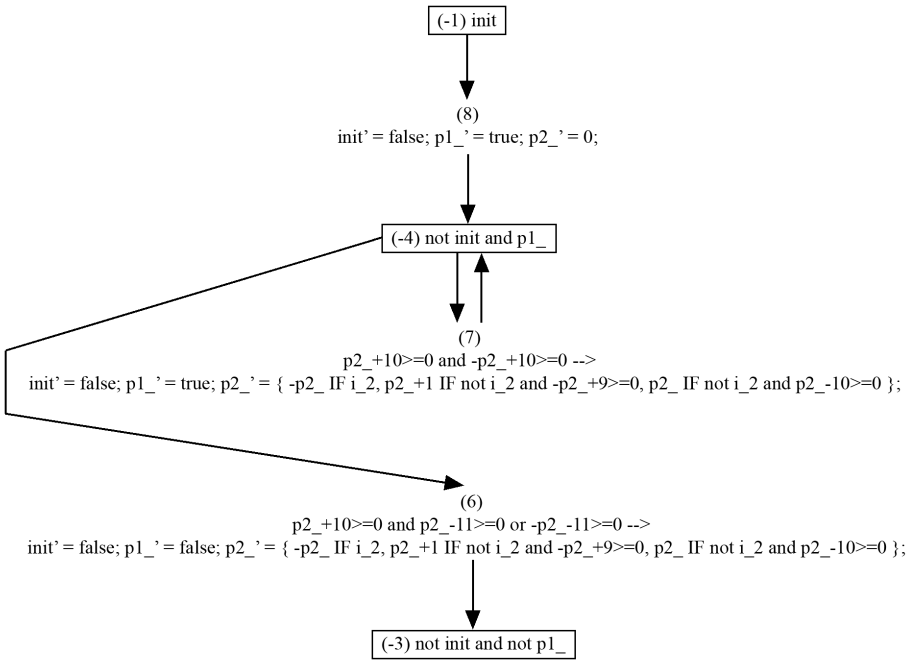


Figure 14.4: CFG printed to Dot: The locations are labeled with their location definitions. The arcs are labeled with "arc assertion → transition function".

– We analyzed the program using forward abstract acceleration, which inferred the displayed invariants in the locations.
– The variables occurring in the invariants correspond to the expressions in the source program listed after `variable mapping`.
– The analysis concluded with the result `PROPERTY TRUE`.

We can also display the CFG (using the Dot format, see Fig. 14.4).

We refer to the tool's website `http://pop-art.inrialpes.fr/people/schramme/reaver/` for additional information.

14.3 Conclusions and Perspectives

We briefly described the tool ReaVer for the verification of discrete and hybrid dataflow programs and its underlying framework, which makes the tool easy to extend. Throughout this work, the tool has proved its utility for experimenting various methods and their combinations as well as for automating benchmark series with different analysis options.

The wish list of features to be integrated into the tool is long. We describe here those that we think are the most important:

Broader support for high-level languages. The current connection of Zelus/Lucid Synchrone to our tool serves as proof-of-concept – *e.g.*, state machines (`automaton`) are not yet supported. In order to provide a broader support for the analysis of high-level synchronous (hybrid) programs, the integration of the Zelus/Lucid Synchrone

front end must be pushed further. Also, the direct integration of the Lustre (V6) front end is desirable.

Return to source. A practical issue when analyzing programs obtained by a translation from high-level languages is that there is no one-to-one correspondence between variables in the source code and the variables used in low-level representation during the analysis. For example, the inlining of nodes requires the duplication of variables, delay operators like `fby` introduce memory variables, and in the case of automata, a variable is added to hold the current state of the automaton. Thus, simply re-substituting corresponding expressions from the program source into the analysis result is not so easy.

Currently, we print the mapping between the state variables in the low-level representation and the corresponding expressions and equations together with their location in the source code. This should be improved to make the user's life easier w.r.t. interpretation of the computed invariants. A graphical user interface could probably help improving the usability in this respect. Moreover, semantic manipulations on the source language level will be necessary, *e.g.*, to simplify expressions.

Partitioning. In our translation to hybrid automata, we assumed that state machines in the Zelus program have been compiled to data-flow. However, it would be desirable to exploit the existing structure of these automata for partitioning the CFG used for analysis.

We have observed in our experimental comparison that the iterative partition refinement based on the heuristics of [Jea00] (cf. §7.4) is sometimes superior to our static partitioning methods (cf. §8.4). Since these refinement techniques are orthogonal to our partitioning and analysis methods, we could combine them with our approach.

Abstract domains. Concerning abstract domains, an efficient LP-solver based template domain implementation within Apron has highest priority. The logico-numerical product domain is currently implemented on top of the power domain: an optimized implementation will certainly speed up the analyses.

Some partitioning and analysis methods require quantifier elimination of numerical variables in formulas: currently, we use the operations for abstract domains available in BddApron, thus, we have to convert formulas to abstract domain values and convert the result back, which is not very efficient. Hence, it would make sense to use dedicated real quantifier elimination methods [ST11] for this purpose.

We are convinced that ReaVer could serve as a platform for connecting various source languages and integrating a variety of abstract interpretation methods for discrete and hybrid systems, which would considerably simplify the automization of experimental comparisons.

Chapter 15

Conclusions and Perspectives

15.1 Summary and Discussion

We started this thesis with the motivation to verify safety-critical embedded systems, and more precisely, systems with Boolean and numerical variables and exhibiting discrete and continuous behavior.

We pursued the objective of improving the precision of inferred properties while enhancing scalability w.r.t. the Boolean combinatorics of logico-numerical systems. In addition, we considered the analysis of embedded systems modeled in hybrid simulation languages.

Our verification approach is based on *logico-numerical analysis methods*, *i.e.*, abstract interpretation methods that are capable of handling both, Boolean and numerical, variables in an implicit way, and *state space partitioning*. Within this framework, we adapted numerical methods that are known to improve the precision to the logico-numerical context. Regarding hybrid simulation languages, we proposed a translation to logical-numerical hybrid automata, and we showed that our logical-numerical analysis approach also applies to such systems.

Our experiments have shown that these methods can improve the efficiency of the analysis by at least one order of magnitude compared to purely numerical approaches, while improving the precision of the discovered invariants.

Verification of discrete numerical systems using abstract acceleration. The first numerical method for discrete systems we considered was abstract acceleration – a precise polyhedral extrapolation operator for some types of affine transformations – to which we contributed the following extensions:

We made it handle *numerical inputs*, which was less straightforward than expected due to the observation that resetting a variable to an input occurring without restrictions in the guard of the transition can emulate a general transformation that we do not (yet) know how to accelerate. In our experiments, we perceived that abstract acceleration was always at least as precise as standard Kleene iteration with widening and descending iterations, and in most cases faster than the latter.

We extended abstract acceleration to *co-reachability analysis* which is slightly different from the forward direction because of the asymmetry of the type of transitions (guarded actions) we considered.

However, in these extensions we only considered translations and translations with resets, which are not the only transitions types that are worth being accelerated. This led

us to *revisit linear accelerability* and to provide a characterization of linearly accelerable linear transformations based on the Jordan canonical form of the homogenized, linear form of affine transformations, which takes into account the initial set. This criterion turned out to be slightly more general than the finite monoid characterization used in exact acceleration. We used this result to formulate an additional abstract acceleration formula. Yet, in general, this requires a change of basis, which is difficult to implement in practice. Thus, we do not yet know how to exploit these results to their full extent in practice.

Verification of discrete logico-numerical systems using abstract acceleration and max-strategy iteration. The adaptation of numerical analysis methods to the logico-numerical context has to cope with the tight coupling between Boolean and numerical variables in logico-numerical transitions.

Our approach to *logico-numerical abstract acceleration* consisted in relaxing this coupling without losing too much precision. The method is supported by suitable partitioning techniques: we proposed a *partitioning method by numerical modes*, which proved very effective in our experiments.

Our *logico-numerical max-strategy iteration* algorithm solves the issue of the coupling between Boolean and numerical variables by exploring increasingly larger Boolean subsystems by alternating propagation and max-strategy iteration phases. While a previous attempt for logico-numerical strategy iteration relied on Kleene iteration and widening, our algorithm relies on LP solving, and thus, it is indeed able to compute the least fixed point for affine programs w.r.t. logico-numerical domains with linear templates.

Our experiments have shown that these techniques can improve the efficiency of the analysis by at least one order of magnitude (w.r.t. system size or computation time) compared to purely numerical approaches, while enhancing the precision of the inferred invariants.

Translation of the hybrid synchronous language Zelus to logico-numerical hybrid automata. Another goal was the verification of high-level languages for embedded systems. We considered the hybrid synchronous language Zelus, which enables tightly integrated programming of systems with discrete and continuous behavior with a semantically clear interaction based on zero-crossings. However, we do not know how to verify systems with zero-crossings using existing hybrid analysis methods.

For this reason, we proposed a sound *translation* from an intermediate *hybrid dataflow formalism* with zero-crossings to *logico-numerical hybrid automata*. This translation entails over-approximations because the semantics of zero-crossings cannot be expressed exactly in hybrid automata. There are several choices for defining the semantics of zero-crossings, which result in qualitatively quite different translations. Moreover, our translation introduces a new finite-type variable for each zero-crossing, which makes the discrete state space larger.

Unbounded-time verification of logico-numerical hybrid automata. These logico-numerical hybrid automata could be verified using existing hybrid verification tools by resorting to Boolean state space enumeration. We showed how to *extend the concept of logico-numerical analyses to hybrid systems* in order to avoid this enumeration:

Although flow transitions in hybrid systems and self-loops in discrete systems raise similar problems, the situation is easier with flow transitions, because only numerical

variables evolve continuously, and thus, Boolean and numerical variables are already decoupled. We proposed a method that dynamically derives flow relations that can be handled by existing hybrid analysis methods, and we exemplified this technique for three unbounded-time hybrid analysis methods. Moreover, we proposed techniques for partitioning the system by its *continuous numerical modes*.

Verification tool ReaVer. We implemented a verification tool ReaVer for abstract interpretation of discrete and hybrid programs. We used logico-numerical hybrid automata as a common internal representation and the BddApron and Apron abstract domain libraries for representing formulas and abstract values. The tool is designed as an extensible framework: new front ends, domains, partitioning and analysis techniques can be added. The current version supports a front end for a simple hybrid data flow format and a subset of Zelus, logico-numerical product and power domains of the available Apron domains, as well as the numerical and logico-numerical versions of the techniques proposed in this thesis.

Summing up, this work proposes a unified approach to the verification of discrete and hybrid logico-numerical systems based on abstract interpretation, which is capable of integrating sophisticated numerical abstract interpretation methods while successfully trading precision for efficiency.

15.2 Perspectives

We conclude by summarizing the most important enhancements, extensions and research directions w.r.t. the proposed methods.

Generalization of the abstract acceleration concept. The currently used graph-expansion to deal with multiple loops in abstract acceleration could be a possible bottleneck in scaling up further. It would be an interesting approach to experiment heuristics for reducing the number of loops by joining them, similarly to the derivative closure method [ACI10]. Moreover, we have only dealt with self-loops. Monniaux and Gonnord [MG11] propose an improvement of abstract acceleration of cycles supported by an SMT solver. We could take advantage of this method, although an adaptation will be necessary to deal with numerical inputs of synchronous programs.

We assume that the potential of abstract acceleration is far from being fully exploited. In this work, we have only made a first step in generalizing the abstract acceleration concept. Further research will be necessary in order to enable the computation of polyhedral approximations that are able to discover complex invariants *e.g.*, of general linear transformations. Such methods are not only of interest for discrete loop analysis but also for computing continuous evolutions in hybrid systems.

Template inference and a logico-numerical max-strategy solver. A related topic is template inference: although max-strategy iteration computes the least fixed point, it cannot discover complex invariants, because the template has to be fixed in advance. Abstract acceleration could be used to guess a better template before computing the precise fixed point using max-strategy iteration.

Our logico-numerical max-strategy is based on a simple, generic algorithm. It would be desirable to develop a more integrated logico-numerical max-strategy solver in order to boost the performance of this approach, possibly by making more extensive use of SMT solvers.

Further improving scalability. Our experimental results showed that partitioning heuristics are crucial for making analyses tractable. In this thesis we considered only static partitioning. Yet, in our experiments the dynamic partitioning technique of Jeannet et al [JHR99] proved to be superior to our static partitioning in some cases. We believe that partition refinement techniques are indispensable in further improving the scalability of our verification tool.

The ASTRÉE tool [BCC+03] uses grouping ("packing") of variables by syntax-based heuristics in order to scale up the analysis of C programs. Packing induces product domains that enable the analysis of groups of variables with distinct precision and thus, it focuses the use of expensive domains like convex polyhedra to presumably relevant sets of variables. Such techniques are orthogonal to ours and could be integrated in our tool. However, their effectiveness in the context of data flow programs remains to be evaluated.

Another point we have not yet exploited until now is the modular structure of high-level programs: in designing complex systems, modularity has proved to be the key to success. We suppose that this is also true for verification: interprocedural analyses have shown that modular approaches help to scale up in static analysis.

All these techniques ultimately trade precision for efficiency.

More complete support for high-level languages. The current version of our tool supports only a small subset of the ZELUS language. We should push our development further to a more complete support. Moreover, a tighter integration with the front end will be necessary in order to implement a more practical return-to-source.

Most existing benchmarks for hybrid systems aim at assessing the precise computation of the continuous behavior, whereas the discrete controller part is rather simplistic. Since it is much easier to program complex examples using these high-level programming languages, we assume that this will give us access to a larger range of relevant benchmarks.

Further experimental evaluation and comparison of methods. We have not yet performed a thorough experimental evaluation of our hybrid system analysis methods. We can only compare the numerical with the logico-numerical version of the hybrid analysis methods we considered, since, to our knowledge, there is no other logico-numerical, unbounded-time verification tool for hybrid systems. It is difficult to reasonably compare unbounded-time methods with bounded-time model checkers like HYSAT and iSAT, as well as with set-simulation-based methods like SPACEEX, because their goal is primarily to find bugs and not to prove their absence. However, in the discrete case, a comparison with tools based on k-induction, *e.g.*, KIND, which are able to provide unbounded-time proofs, would be interesting.

Finally, there is a plethora of proposed abstract interpretation methods often implemented in prototype tools with incompatible input formats, which makes their comparison cumbersome and time-consuming. A comparison of a wide range of methods would be possible by plugging these techniques into our tool framework.

Further applications and combining approaches. The hybrid programs we consider are general event-triggered systems that can emulate time-triggered systems. However, time-triggered systems can probably be analyzed more precisely when taking into account the strictly periodic structure of their discrete transitions associated to the controller. Zutshi et al [ZST12] propose an analysis method for time-triggered (hybrid)

systems based on relational abstractions. It would be worthwhile to extend their method to logico-numerical systems.

Traditionally, methods developed for verification have their applications in related areas like controller and parameter synthesis [EA10, FK11], test case generation [DB99, JJ05, JJRZ05] and debugging [GJJM03, Gau03]. It would be interesting to investigate to which extent we could exploit our methods in these applications.

Recently, theorem provers have been used in static analysis [ZLKR04, MV06], SMT-based k-induction is enhanced by abstract interpretation [RDG10] and abstract interpretation methods are boosted by SMT solvers (*e.g.*, [MG11, GM11]). Moreover, SAT solving is viewed as deductive proving [FJOS03] and the analogies of SMT solving, and abstract interpretation are pointed out [DHK12, CCM11]. This tendency to a convergence of these different approaches opens an interesting field of research.

Bibliography

[AB08] Vincent Acary and Bernard Brogliato. *Numerical Methods for Nonsmooth Dynamical Systems: Applications in Mechanics and Electronics*. Springer, 2008.

[ABDM00] Eugene Asarin, Olivier Bournez, Thao Dang, and Oded Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 21–31. Springer, 2000.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[ACHH92] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, Lecture Notes in Computer Science, pages 209–229. Springer, 1992.

[ACI10] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. In *Numerical and Symbolic Abstract Domains*, volume 267 of *Electronical Notes in Theoretical Computer Science*, pages 3–16. Elsevier, 2010.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[ADE+03] Rajeev Alur, Thao Dang, Joel M. Esposito, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, 2003.

[ADG07] Eugene Asarin, Thao Dang, and Antoine Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Informatica*, 43(7):451–476, 2007.

[ADI02] Rajeev Alur, Thao Dang, and Franjo Ivancic. Reachability analysis of hybrid systems using counter-example guided predicate abstraction. Technical report, University of Pennsylvania, 2002.

[AGG10a] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2010.

[AGG10b] Xavier Allamigeon, Stéphane Gaubert, and Eric Goubault. The tropical double description method. In *Theoretical Aspects of Computer Science*, volume 5 of *LIPIcs*, pages 47–58. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[AKRS08] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Embedded Software*, pages 89–98. ACM, 2008.

[AMP95] Eugene Asarin, Oded Maler, and Amir Pnueli. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 138(1):35–65, 1995.

[ASK04] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *Electronical Notes in Theoretical Computer Science*, 109:43–56, 2004.

[AW85] Edward A. Ashcroft and William W. Wadge, editors. *LUCID, the Data-Flow Programming Language*. Academic Press, 1985.

[BBBC09] Anna Beletska, Denis Barthou, Wlodzimierz Bielecki, and Albert Cohen. Computing the transitive closure of a union of affine integer tuple relations. In *Combinatorial Optimization and Applications*, volume 6508 of *Lecture Notes in Computer Science*, pages 98–109. Springer, 2009.

[BBBS02] Yannis Bres, Gérard Berry, Amar Bouali, and Ellen M. Sentovich. State abstraction techniques for the verification of reactive circuits. In *Designing Correct Circuits*, volume 531 of *Lecture Notes in Computer Science*, pages 197–203. Springer, 2002.

[BBCP11a] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *Languages, Compilers, and Tools for Embedded Systems*, pages 61–70. ACM, 2011.

[BBCP11b] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code. In *Embedded Software*, pages 137–148. ACM, 2011.

[BBCP12] Albert Benveniste, Timothy Bourke, Benoît Caillaud, and Marc Pouzet. Non-standard semantics of hybrid systems modelers. *Journal of Computer and System Sciences*, 78(3):877–910, 2012.

[BBLS92] Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 260–273. Springer, 1992.

[BC85] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, Lecture Notes in Computer Science, pages 389–448. Springer, 1985.

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, pages 196–207. ACM, 2003.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[BCP10] Albert Benveniste, Benoît Caillaud, and Marc Pouzet. The fundamentals of hybrid systems modelers. In *Conference on Decision and Control*, pages 4180–4185. IEEE, 2010.

[BFH91] Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. Minimal model generation. In *Computer-Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, pages 197–203. Springer, 1991.

[BFL04] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. FASTer acceleration of counter automata in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590, 2004.

[BFLP03] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast: Fast acceleration of symbolic transition systems. In *Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BFLP08] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: acceleration from theory to practice. *Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008.

[BFLS05] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Philippe Schnoebelen. Flat acceleration in symbolic model checking. In *Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.

[BG97] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *Formal Methods in System Design*, 14(3):237–255, 1997.

[BGI09] Marius Bozga, Codruţa Gîrlea, and Radu Iosif. Iterating octagons. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2009.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[BGL00] Tevfik Bultan, Richard Gerber, and Christopher League. Composite model-checking: verification with type-specific symbolic representations. *Transactions on Software Engineering Methodologies*, 9(1):3–50, 2000.

[BGP97] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 1997.

[BHvMW09] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185. IOS Press, 2009.

[BHZ04] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004.

[BHZ05] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17(2):222–257, 2005.

[BIK10] Marius Bozga, Radu Iosif, and Filip Konecný. Fast acceleration of ultimately periodic relations. In *Computer-Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2010.

[Bil87] J. P. Billon. Perfect normal forms for discrete programs. Technical report, BULL, 1987.

[BJ10] Xavier Briand and Bertrand Jeannet. Combining control and data abstraction in the verification of hybrid systems. *Computer-Aided Design of Integrated Circuits and Systems*, 10:1481–1494, 2010.

[BK09] Simon Bliudze and Daniel Krob. Modelling of complex systems: Systems as dataflow machines. *Fundamenta Informaticae*, 91(2):251–274, 2009.

[Bli06] Simon Bliudze. *Un cadre formel pour l'étude des systèmes industriels complexes: un exemple basé sur l'infrastructure de l'UMTS*. PhD thesis, Ecole Polytechnique, 2006.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[Boi98] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, University of Liège, 1998.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993.

[Bou08] Olivier Bouissou. *Analyse statique par interprétation abstraite de systèmes hybrides*. PhD thesis, École polytechnique, 2008.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *Transactions on Computers*, 35(8):677–692, 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *Transactions on Computers*, 40(2):205–213, 1991.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BT00] Oleg Botchkarev and Stavros Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In *Hybrid Systems: Computation and Control*, Lecture Notes in Computer Science, pages 73–88. Springer, 2000.

[BV04] Stephen Boyd and Lieven Vandenberghe, editors. *Convex optimization*. Cambridge University Press, 2004.

[BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *Transactions on Computers*, 45(9):993–1002, 1996.

[CBM89] Olivier Coudert, Christian Berthet, and Jean C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1989.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CC76] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295, 1992.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does ASTRÉE scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *Foundations of Software Science and Computation Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer, 2011.

[CGG+05] Alexandru Costan, Stephane Gaubert, Éric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer-Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005.

[CGHP04] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT*, pages 230–239. ACM, 2004.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, pages 84–97. ACM, 1978.

[Che65] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[CHP08] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming: The Lucid Synchrone experiment. In Nicolas Navet and Stephan Merz, editors, *Real-Time Systems: Description and Verification Techniques: Theory and Tools*. Hermes, 2008.

[CJ98] Hubert Comon and Yan Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.

[CK98] Alonkrit Chutinan and Bruce H. Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *Conference on Decision and Control*, pages 2089–2094. IEEE, 1998.

[CK99] Alongkrit Chutinan and Bruce H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 1999.

[CK01] Alonkrit Chutinan and Bruce H. Krogh. Verification of infinite-state dynamic systems using approximate quotient transition systems. *IEEE Transactions on Automatic Control*, 46(9):1401 –1410, 2001.

[CK03] Alonkrit Chutinan and Bruce H. Krogh. Computational techniques for hybrid systems verification. In *Transactions on Automatic Control*, volume 48, pages 64–75. IEEE, 2003.

[CMZ+93] Edmund M. Clarke, Kenneth L. McMillan, Xudong Zhao, Masahiro Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation Conference*, pages 54–60, 1993.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[Cou77] Patrick Cousot. Asynchronous iteration methods for solving a fixed point system of monotone equations in a complete lattice. Rapport de Recherche No. 88, Laboratoire d'Informatique, Grenoble, France, 1977.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

[CP96] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, pages 226–238. ACM, 1996.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *Principles of Programming Languages*, pages 178–188. ACM, 1987.

[CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Embedded Software*, pages 173–182. ACM, 2005.

[DB99] L. Du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, Sept 1999.

[DG11a] Thao Dang and Thomas Martin Gawlitza. Discretizing affine hybrid automata with uncertainty. In *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 473–481. Springer, 2011.

[DG11b] Thao Dang and Thomas Martin Gawlitza. Template-based unbouded time verification of affine hybrid automata. In *Asian Symposium on Programming Languages and Systems*, Lecture Notes in Computer Science, pages 34–49. Springer, 2011.

[DGM09] Thao Dang, Colas Le Guernic, and Oded Maler. Computing reachable states for nonlinear biological models. In *Computational Methods in Systems Biology*, volume 5688 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2009.

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[DHK12] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analyzers. In *Static Analysis Symposium*, Lecture Notes in Computer Science, page to appear. Springer, 2012.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 351–368. Springer, 2011.

[DM98] Thao Dang and Oded Maler. Reachability analysis via face lifting. In *Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 96–109. Springer, 1998.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[DT97] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: Introduction*. Springer, 1997.

[EA10] Étienne André. *Timed Parametrization of Embedded System Components*. PhD thesis, École Normale Supérieure de Cachan, Dec 2010.

[EFH08] Andreas Eggers, Martin Fränzle, and Christian Herde. SAT Modulo ODE: A direct SAT approach to hybrid systems. In *Automated Technology for Verification and Analysis*, volume 5311 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2008.

[ERNF11] Andreas Eggers, Nacim Ramdani, Nedialko Nedialkov, and Martin Fränzle. Improving SAT Modulo ODE for hybrid systems analysis by combining different enclosure methods. In *Software Engineering and Formal Methods*, volume 7041 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2011.

[FE98] Peter Fritzson and Vadim Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1998.

[FGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer-Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[FH07] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.

[Fil60] Aleksei F. Filippov. Differential equations with discontinuous right-hand sides. *Mathematicheskii Sbornik*, 51(1), 1960.

[FJOS03] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2003.

[FK11] Laurent Fribourg and Ulrich Kühne. Parametric verification and test coverage for hybrid automata using the inverse method. In *Reachability Problems*, volume 6945 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2011.

[FL02] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[FO97] Laurent Fribourg and Hans Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 1997.

[FP95] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer, 1995.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Principles of Programming Languages*, pages 191–202. ACM, 2002.

[Fre05] Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.

[Gau03] Fabien Gaucher. *Étude du débogage de systèmes réactifs et application au langage synchrone Lustre*. PhD thesis, Grenoble INP, Nov 2003.

[Gaw09] Thomas M. Gawlitza. *Strategieverbesserungsalgorithmen für exakte Programmanalysen*. PhD thesis, Technische Universität München, 2009.

[GG09] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In *Computer-Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 540–554. Springer, 2009.

[GGTZ07] Stephane Gaubert, Éric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[GIB+12] Khalil Ghorbal, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 235–250. Springer, 2012.

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2005.

[GJJM03] F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *Automated and Algorithmic Debugging*, 2003.

[GJS95] Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Hybrid cc, hybrid automata and program verification. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 1995.

[GM99] Mark R. Greenstreet and Ian Mitchell. Reachability analysis using polygonal projections. In *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*, pages 103–116. Springer, 1999.

[GM11] Thomas . Gawlitza and David Monniaux. Improving strategies via SMT solving. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 236–255. Springer, 2011.

[GMP02] Éric Goubault, Matthieu Martel, and Sylvie Putot. Asserting the precision of floating-point computations: A simple abstract interpreter. In *European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[Gon] Laure Gonnord. The ASPIC tool: Accelerated symbolic polyhedral invariant computation. `http://laure.gonnord.org/pro/aspic/aspic.html`.

[Gon07] Laure Gonnord. *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, Oct 2007.

[GP06] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.

[GR98] Roberto Giacobazzi and Francesco Ranzato. Optimal domains for disjunctive abstract intepretation. *Science of Computer Programming*, 32(1-3):177–210, 1998.

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In *Static Analysis Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GS07a] Thomas M. Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2007.

[GS07b] Thomas M. Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.

[GS08] Thomas M. Gawlitza and Helmut Seidl. Precise interval analysis vs. parity games. In *Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 342–357. Springer, 2008.

[GS10] Thomas M. Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *Static Analysis Symposium*, volume 6337 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2010.

[GS11] Thomas M. Gawlitza and Helmut Seidl. Solving systems of rational equations through strategy iteration. *Transactions on Programming Languages and Systems*, 33(3):11, 2011.

[GSA+12] Thomas M. Gawlitza, Helmut Seidl, Assalé Adjé, Stephane Gaubert, and Éric Goubault. Abstract interpretation meets convex optimization. *Journal of Symbolic Computation*, 47(12):1512–1532, 2012.

[GT08] Sumit Gulwani and Ashish Tiwari. Constraint-based approach for analysis of hybrid systems. In *Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 190–203. Springer, 2008.

[Hag08] George Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. Phd dissertation, University of Iowa, Dec 2008.

[Hal79] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. Thèse du 3ème cycle, Institut National Polytechnique de Grenoble, Mar 1979.

[Hal93a] Nicolas Halbwachs. Delay analysis in synchronous programs. In *Computer-Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 1993.

[Hal93b] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[Hal98] Nicolas Halbwachs. Synchronous programming of reactive systems - a tutorial and commented bibliography. In *Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[HBG+05] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *Transactions on Mathematical Software*, 31(3):363–396, 2005.

[Hen95] Thomas A. Henzinger. Hybrid automata with finite bisimulations. In *International Colloquium on Automata, Languages and Programming*, volume 944 of *Lecture Notes in Computer Science*, pages 324–335. Springer, 1995.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science*, page 278. IEEE, 1996.

[HH94] Thomas A. Henzinger and Pei-Hsin Ho. Model checking strategies for linear hybrid systems. Technical report, Cornell University, 1994.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[HHWT98] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *Transactions on Automatic Control*, 43:225–238, 1998.

[HK09] Jacob M. Howe and Andy King. Logahedra: A new weakly relational domain. In *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[HLR93] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology And Software Technology*, Workshops in Computing, pages 83–96. Springer, 1993.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science*, pages 394–406. IEEE, 1992.

[Ho95] Pei-Hsin Ho. *Automatic Analysis of Hybrid Systems*. PhD thesis, Cornell University, 1995.

[HPR97] 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, 1997.

[HRP94] Nicolas Halbwachs, Pascal Raymond, and Yann-Erick Proy. Verification of linear hybrid systems by means of convex approximations. In *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 1994.

[HT98] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1998.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, 2008.

[IFS+95] Yumi Iwasaki, Adam Farquhar, Vijay A. Saraswat, Daniel G. Bobrow, and Vineet Gupta. Modeling time in hybrid systems: How fast is "instantaneous"? In *International Joint Conference on Artificial Intelligence*, pages 1773–1781, 1995.

[Jea] Bertrand Jeannet. BDDAPRON: A logico-numerical abstract domain library. `http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/`.

[Jea00] Bertrand Jeannet. *Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones*. Thèse de doctorat, Grenoble INP, Sept 2000.

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JHR99] Bertrand Jeannet, Nicolas Halbwachs, and Pascal Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 1999.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JJRZ05] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 349–364. Springer, 2005.

[JM09] Bertrand Jeannet and Antoine Miné. APRON: A library of numerical abstract domains for static analysis. In *Computer-Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–396, 1984.

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[KPRS96] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International Journal of Parallel Programming*, 24(6):579–598, 1996.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[KV00] Alex Kurzhanskiy and Pravin Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 202–214. Springer, 2000.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[LCJG11] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 492–502. Springer, 2011.

[Ler03] Jérôme Leroux. *Algorithmique de la vérification des systèmes à compteurs – Approximation et accélération – Implémentation dans l'outil FAST*. Thèse de doctorat, École Normale Supérieure de Cachan, Dec 2003.

[Lin88] Tom Lindstrøm. An invitation to nonstandard analysis. In N. Cutland, editor, *Nonstandard Analysis and its Applications*, pages 1–105. Cambridge University Press, 1988.

[Löh88] Rainald Löhner. *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben und Anwendungen*. PhD thesis, Universität Karlsruhe, 1988.

[LS05] Jérôme Leroux and Grégoire Sutre. Flat counter automata almost everywhere! In *Automated Technology for Verification and Analysis*, volume 3707 of *Lecture Notes in Computer Science*, pages 489–503. Springer, 2005.

[LS07] Jérôme Leroux and Grégoire Sutre. Acceleration in convex data-flow analysis. In *Foundations of Software Technology and Theoretical Computer Science*, volume 4855 of *Lecture Notes in Computer Science*, pages 520–531. Springer, 2007.

[LT84] Peter Lancaster and Miron Tismenetsky. *The Theory of Matrices*. Academic Press, 2nd edition, 1984.

[LZ05] Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 25–53. Springer, 2005.

[Mau96] Christophe Mauras. Calcul symbolique et automates interprétés. Technical Report No. 10, IRCCyN, 1996.

[Mau98] Laurent Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31(1):91–112, 1998.

[MG11] David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In *Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 369–385. Springer, 2011.

[Min67] Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.

[Min01a] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.

[Min01b] Antoine Miné. The octagon abstract domain. In *Working Conference on Reverse Engineering*, pages 310–319. IEEE, 2001.

[MMBC11] Karthik Manamcheri, Sayan Mitra, Stanley Bak, and Marco Caccamo. A step towards verification and synthesis from Simulink/Stateflow models. In *Hybrid Systems: Computation and Control*, pages 317–318. ACM, 2011.

[Mon09] David Monniaux. Automatic modular abstractions for linear constraints. In *Principles of Programming Languages*, pages 140–151. ACM, 2009.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice Hall, 1966.

[MRTT53] T.S. Motzkin, H. Raiffa, G.L. Thompson, and R.M. Thrall. The double description method. In H.W.Kuhn and A.W. Tucker, editors, *Contributions to the theory of games, Vol. 2*. Princeton University Press, 1953.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD Foundations and Applications*. Springer, 1998.

[MT00] Ian Mitchell and Claire Tomlin. Level set methods for computation in hybrid systems. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 310–323. Springer, 2000.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MV06] Panagiotis Manolios and Daron Vroon. Integrating static analysis and general-purpose theorem proving for termination analysis. In *International Conference on Software Engineering*, pages 873–876. ACM, 2006.

[NJC99] Nedialko S. Nedialkov, Kenneth R. Jackson, and George F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21–68, 1999.

[NN07] Masaoud Najafi and Ramine Nikoukhah. Implementation of hybrid automata in SCICOS. In *16th IEEE International Conference on Control Applications*, pages 819–824, Singapore, Oct 2007.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[OEM99] Martin Otter, Hilding Elmqvist, and Sven Erik Mattsso. Hybrid modeling in Modelica based on the synchronous data flow principle. In *Computer-Aided Control System Design*, pages 22–27, 1999.

[PJ04] Stephen Prajna and Ali Jadbabaie. Safety verification of hybrid systems using barrier certificates. In *Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 2004.

[Pla08] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[Pla10] André Platzer. Differential-algebraic dynamic logic for differential-algebraic programs. *Journal of Logic and Computation*, 20(1):309–352, 2010.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pou02] Marc Pouzet. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Paris, France, Nov 2002. Habilitation à diriger les recherches.

[Pou06] Marc Pouzet. *Lucid Synchrone Release version 3.0: Tutorial and Reference Manual*, 2006.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In *International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 171–178. Springer, 2008.

[Pre30] Mojżezs Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu matematyków krajów s lowiańskich, Warszawa 1929*, pages 92–101, 1930.

[Ray91] Pascal Raymond. *Compilation efficace d'un langage déclaratif synchrone: Le générateur de code LUSTRE-V3*. Thèse de doctorat, Grenoble INP, Nov 1991.

[RDG10] Pierre Roux, Remi Delmas, and Pierre-Loïc Garoche. SMT-AI: an abstract interpreter as oracle for k-induction. *Electronical Notes in Theoretical Computer Science*, 267(2):55–68, 2010.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems*, 29(5):26, 2007.

[Rob96] Abraham Robinson. *Non-Standard Analysis*. Princeton Landmarks in Mathematics, 1996.

[RRJ08] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008.

[RS03] William C. Rounds and Hosung Song. The phi-calculus: A language for distributed control of reconfigurable embedded systems. In *Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2003.

[RS07] Stefan Ratschan and Zhikun She. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *Transactions on Embedded Computing Systems*, 6(1), 2007.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.

[SB02] J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer, 3rd edition, 2002.

[Sca] SCADE. `http://www.esterel-technologies.com/products/scade-suite/`.

[SH11] Kohei Suenaga and Ichiro Hasuo. Programming with infinitesimals: A While-language for hybrid system modeling. In *International Colloquium on Automata, Languages and Programming*, volume 6756 of *Lecture Notes in Computer Science*, pages 392–403. Springer, 2011.

[Sim] Simulink: Simulation and model-based design. `http://www.mathworks.com/products/simulink/`.

[SISG06] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.

[SJ10] Peter Schrammel and Bertrand Jeannet. Extending abstract acceleration to data-flow programs with numerical inputs. In *Numerical and Symbolic Abstract Domains*, volume 267 of *Electronical Notes in Theoretical Computer Science*, pages 101–114. Elsevier, 2010.

[SJ11] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

[SJ12a] Peter Schrammel and Bertrand Jeannet. Applying abstract acceleration to (co-)reachability analysis of reactive programs. *Journal of Symbolic Computation*, 47(12):1512–1532, 2012.

[SJ12b] Peter Schrammel and Bertrand Jeannet. From hybrid data-flow languages to hybrid automata: A complete translation. In *Hybrid Systems: Computation and Control*, pages 167–176. ACM, 2012.

[SJ12c] Peter Schrammel and Bertrand Jeannet. From hybrid data-flow languages to hybrid automata: A complete translation. Research Report RR-7859, INRIA, Jan 2012.

[SJVG11] Pascal Sotin, Bertrand Jeannet, Franck Védrine, and Éric Goubault. Policy iteration within logico-numerical abstract domains. In *Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2011.

[SK03] Olaf Stursberg and Bruce H. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 482–497. Springer, 2003.

[SK06] Axel Simon and Andy King. Widening polyhedra with landmarks. In *Asian Symposium on Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2006.

[SS13] Peter Schrammel and Pavle Subotic. Logico-numerical max-strategy iteration. In *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 414–433. Springer, 2013.

[SSM04] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Principles of Programming Languages*, pages 318–329. ACM, 2004.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2005.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[ST11] Sriram Sankaranarayanan and Ashish Tiwari. Relational abstractions for continuous and hybrid systems. In *Computer-Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 686–702. Springer, 2011.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2), 1955.

[Tiw03] Ashish Tiwari. Approximate reachability for linear systems. In *Hybrid Systems: Computation and Control*, volume 2623 of *Lecture Notes in Computer Science*, pages 514–525. Springer, 2003.

[Tiw08] Ashish Tiwari. Abstractions for hybrid systems. *Formal Methods in System Design*, 32(1):57–83, 2008.

[Tom98] Claire. J. Tomlin. *Hybrid Control of Air Traffic Management Systems*. PhD thesis, University of California, Berkeley, 1998.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *Transactions on Embedded Computing Systems*, 4(4):779–818, 2005.

[Utk92] Vadim I. Utkin. *Sliding modes in Control and optimization*. Springer, 1992.

[vBMR+06] Dirk A. van Beek, Ka L. Man, Michel A. Reniers, Jacobus E. Rooda, and Ramon R. H. Schiffelers. Syntax and consistent equation semantics of Hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129–210, 2006.

[Ver92] Hervé Le Verge. A note on Chernikowa's algorithm. Research report 1662, IRISA, 1992.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 1998.

[Wil86] H. P. Williams. Fourier's method of linear programming and its dual. *American Mathematical Monthly*, 93:681–695, 1986.

[Wil88] J. H. Wilkinson, editor. *The algebraic eigenvalue problem.* Oxford University Press, 1988.

[WT94] Howard Wong-Toi. *Symbolic approximations for verifying real-time systems.* PhD thesis, Stanford University, Dec 1994.

[WYGI07] Chao Wang, Zijiang Yang, Aarti Gupta, and Franjo Ivancic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Computer-Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 352–365. Springer, 2007.

[YKTB01] Tuba Yavuz-Kahveci, Murat Tuncer, and Tevfik Bultan. A library for composite symbolic representations. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2001.

[ZLA06] Haiyang Zheng, Edward A. Lee, and Aaron D. Ames. Beyond Zeno: Get on with it! In *Hybrid Systems: Computation and Control*, volume 3927 of *Lecture Notes in Computer Science*, pages 568–582. Springer, 2006.

[ZLKR04] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *Workshop on Software Verification and Validation*, 2004.

[ZST12] Aditya Zutshi, Sriram Sankaranarayanan, and Ashish Tiwari. Timed relational abstractions for sampled data control systems. In *Computer-Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2012.

Résumé

Cette thèse étudie la vérification automatique de propriétés de sûreté de systèmes logico-numériques discrets ou hybrides. Ce sont des systèmes ayant des variables booléennes et numériques et des comportements discrets et continus. Notre approche est fondée sur l'analyse statique par interprétation abstraite.

Nous traitons les problèmes suivants : les méthodes d'interprétation abstraite numériques exigent l'énumération des états booléens, et par conséquent, elles souffrent du problème d'explosion d'espace d'états. En outre, il y a une perte de précision due à l'utilisation d'un opérateur d'élargissement afin de garantir la terminaison de l'analyse. Par ailleurs, nous voulons rendre les méthodes d'interprétation abstraite accessibles à des langages de simulation hybrides.

Dans cette thèse, nous généralisons d'abord l'accélération abstraite, une méthode qui améliore la précision des invariants numériques inférés. Ensuite, nous montrons comment étendre l'accélération abstraite et l'itération de max-stratégies du contexte numérique au contexte logico-numérique, ce qui aide à améliorer le compromis entre l'efficacité et la précision. En ce qui concerne les systèmes hybrides, nous traduisons le langage de programmation synchrone et hybride Zelus vers les automates hybrides logico-numériques, et nous étendons les méthodes d'analyse logico-numérique aux systèmes hybrides. Enfin, nous avons mis en œuvre les méthodes proposées dans un outil nommé ReaVer et nous fournissons des résultats expérimentaux.

En conclusion, cette thèse propose une approche unifiée à la vérification de systèmes logico-numériques discrets et hybrides fondée sur l'interprétation abstraite, qui est capable d'intégrer des méthodes d'interprétation abstraite numériques sophistiquées tout en améliorant le compromis entre l'efficacité et la précision.

Mots-clés: vérification, programmes logico-numériques, langages synchrones, systèmes hybrides, analyse statique, interprétation abstraite, accélération abstraite, itération de stratégies.

Abstract

This thesis studies the automatic verification of safety properties of logico-numerical discrete and hybrid systems. These systems have Boolean and numerical variables and exhibit discrete and continuous behavior. Our approach is based on static analysis using abstract interpretation.

We address the following issues: Numerical abstract interpretation methods require the enumeration of the Boolean states, and hence, they suffer from the state space explosion problem. Moreover, there is a precision loss due to widening operators used to guarantee the termination of the analysis. Furthermore, we want to make abstract interpretation-based analysis methods accessible to simulation languages for hybrid systems.

In this thesis, we first generalize abstract acceleration, a method that improves the precision of the inferred numerical invariants. Then, we show how to extend abstract acceleration and max-strategy iteration to logico-numerical programs while improving the trade-off between efficiency and precision. Concerning hybrid systems, we translate the Zelus hybrid synchronous programming language to logico-numerical hybrid automata and extend logico-numerical analysis methods to hybrid systems. Finally, the proposed methods are implemented in ReaVer, a REActive System VERification tool, and we provide experimental results.

To conclude, this thesis proposes a unified approach to the verification of discrete and hybrid logico-numerical systems based on abstract interpretation, which is capable of integrating sophisticated numerical abstract interpretation methods while successfully trading precision for efficiency.

Keywords: verification, logico-numerical programs, synchronous languages, hybrid systems, static analysis, abstract interpretation, abstract acceleration, strategy iteration.