THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

Présentée et soutenue par :
Adrien Champion

le mardi 7 janvier 2014

Titre :

Collaboration de techniques formelles pour la vérification de propriétés de sûreté sur des systèmes de transition

---

École doctorale et discipline ou spécialité :
ED MITT : Sureté de logiciel et calcul de haute performance

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

Mme Virginie Wiels (directrice de thèse)
M. Rémi Delmas (co-directeur de thèse)

Jury :
Pr. Yamine Aït-Ameur (INPT, France) -- Président
Pr. Cesare Tinelli (Univ. Iowa, USA) -- Rapporteur
Pr. Daniel Le Berre (Univ. d'Artois, France) -- Rapporteur
Dr. Steve Miller (Rockwell Collins USA) -- Examinateur
Ass. Pr. Viktor Kuncak (École Polytechnique Fédérale de Lausanne, Suisse) -- Examinateur
Dr. Virginie Wiels (ONERA) -- Directrice de thèse
Dr. Rémi Delmas (ONERA) -- Co-Directeur de thèse
Dr. Michaël Dierkes (Rockwell Collins France)-- Encadrant de thèse industriel

Acknoledgments

First, I would like to thank my supervisors Rémi Delmas and Michael Dierkes for their support, knowledge, time and energy. They provided me with the best environment I could wish for. Everything I did during these last three years, I owe to them.
I also thank the reviewers, Daniel Le Berre and Cesare Tinelli, for taking the time to read the manuscript and giving me constructive feedback, which lead to interesting discussions. Many thanks to all the members of the jury: Yamine Aït Ameur for making the defense possible and for his interest and insightful questions on my work; Virginie Wiels for making this PhD possible at all, along with Rémi Delmas and Michael Dierkes; Steve Miller for flying all the way to Toulouse, for his interest in my work, for his tricky questions on Boolean differential calculus and for all his reviews on the manuscript; Viktor Kuncak for following my work, and for the great discussions we had about both our research topics; Last, I thank again Daniel Le Berre and Cesare Tinelli for taking the time to come to Toulouse and attend the defense.

A few lines to thank the people close to me. I will only mention their names, the support they gave me cannot be put into words. Thanks, love and apologies to Yves et Marie-Claire, Claire, Yohann, $\mathcal{X}$, Emy et Salomé, Tonton Chips, Rémi, Tof, Foufou, St Pouf, Étienne et Maud et Chouquette.

Contents

Extended Abstract (French) 1

1 Introduction 5
1.1 Cas d'Étude . . . 5
1.2 Notations et Notions Mathématiques . . . 7
1.3 Vérification de Systèmes de Transition . . . 12

2 Découverte d'Invariants 17
2.1 Élimination de Quantificateurs . . . 18
2.2 Une Architecture Collaborative . . . 27
2.3 Découverte d'Invariants Potentiels par QE . . . 29
2.4 Le Framework Formel *Stuff* . . . 34

3 Conclusion 37
3.1 Conclusion . . . 37
3.2 Perspectives . . . 38

I Introduction 1

4 Problem 3
4.1 Lustre: A Synchronous Programming Language . . . 3
4.2 Safety Critical Avionics Systems Verification . . . 9

5 Mathematical Background and Notation 13
5.1 Many-Sorted First-Order Logic . . . 13
5.2 SAT and SMT Solving . . . 25
5.3 SMT: Standard, Implementations and Additional Features . . . 32

6 Transition Systems, State Invariant Verification 35
6.1 Transition Systems . . . 36
6.2 Bounded Model Checking and k-induction . . . 39
6.3 Invariant Discovery and Other Techniques . . . 47

6.4 Motivation and Outline of the Thesis . . . 53

II Invariant Discovery Powered By Quantifier Elimination **55**

7 SMT-Based Quantifier Elimination **59**
7.1 Monniaux's QE Algorithm . . . 59
7.2 Extension to Booleans and Integer Octagons . . . 60
7.3 Improving Extrapolation . . . 62
7.4 Benchmarking Results . . . 68

8 Invariant Discovery in a K-induction-based Framework **73**
8.1 A Proof Engine Architecture Allowing Continuous Invariant Integration . . . 73
8.2 Invariant Discovery as a Quantifier Elimination Problem . . . 75
8.3 Property Directed Reachability Modulo Theory . . . 76

9 HullQe: QE And Convex Hull Computation **83**
9.1 Extracting Potential Invariants From Pre-Images . . . 84
9.2 Exhaustive Exact Convex Hull Enumeration . . . 89
9.3 Applications . . . 95
9.4 End-to-end Verification Of A Functional Chain . . . 100

III The Stuff Formal Framework **103**

10 Architecture and Transition System Representation **105**
10.1 The Actor Formalism . . . 105
10.2 Stuff: Architecture . . . 110
10.3 Stream Systems and Unrolling . . . 113

11 Assumptio: an SMT-lib 2 Compliant Solver Wrapper **121**
11.1 Overview . . . 122
11.2 Architecture and Backend Solvers . . . 124
11.3 Source Code, User Manual and Integration in Stuff . . . 127

IV Conclusion **129**

12 Summary and Perspectives **131**
12.1 Conclusion . . . 131
12.2 Future Directions . . . 132

13 References 137

V Appendices 147

A Use Cases 149
A.1 The Rockwell Collins Triplex Voter 149
A.2 A Reconfiguration Logic System 151

List of Figures

1.1 A Single Computation Channel. 5
1.2 Triple Channel Functional Chain. 6

2.1 Règles de calcul de l'analyse structurelle. 24
2.2 Une architecture collaborative basée sur la k-induction. 28
2.3 Polyhedra examples. 31
2.4 ECH de polyhèdres entiers. 31
2.5 Découverte d'invariant relationnels sur le système de reconfiguration. 32
2.6 Logique de vote en dimension deux. 34
2.7 Une chaîne fonctionnelle simplifiée. 35

4.1 A simple Lustre program. 6
4.2 Network representation of the `bounded` node. 7
4.3 The observer pattern. 8
4.4 Bounded liveness property as a safety property. 9
4.5 A Single Computation Channel. 9
4.6 Triple Channel Functional Chain. 10
4.7 The shuffle mechanism. 11

5.1 Push and pop example, assertion stack on the right. 33

6.1 Reachable state space of $S_{DC}(4, 2)$, nodes show the values of x and y. 42

7.1 Activation conditions and cause propagation rules. 66
7.2 Benchmark results part 1. 69
7.3 Benchmark results part 2. 71
7.4 Experimental results. 72

8.1 k-induction based collaborative framework. 74
8.2 Asynchronous exact convex hull computation. 80

9.1 Polyhedra examples. 84
9.2 HullQe abstract view. 85
9.3 Discovering new relations using convex hulls. 86

9.4 ECH calculation on the double counter with $\mathbf{n_x} = 10$ and $\mathbf{n_y} = 6$. 86
9.5 Duplex voter equations. 88
9.6 Simple voting logic. 88
9.7 Hullification redundancy issues. 93
9.8 View of HullQe internal processes. 95
9.9 Reconfiguration subsystem with observer. 96
9.10 Triplex voter equations. 98
9.11 Running example. 100

10.1 Message handling. 111
10.2 A code extract of the `handleMessage` function of the k-induction `Method`. . 112
10.3 Actor hierarchy in Stuff. 113
10.4 A trace of a Stream System. 115
10.5 Traces of the double counter system. 117

11.1 Hi-level view of user interaction and internal architecture. 125

PART

Extended Abstract (French)

Les *systèmes critiques* sont des systèmes dont les défaillances ont des conséquences catastrophiques telles que la perte de vies humaines ou de fortes retombées économiques. On les trouve typiquement dans des domaines tels que l'aérospatiale, l'aéronautique, l'automobile, le ferroviaire, le nucléaire, les équipements médicaux, la conception de matériel informatique... Il est donc d'un grand intérêt *(i)* d'établir une spécification précise de ces systèmes, et *(ii)* de s'assurer de la correction du produit final par rapport à sa spécification, *i.e.* de *vérifier* le produit final. L'étude présentée dans ce mémoire se concentre sur la vérification de composants logiciels de systèmes critiques embarqués avioniques.

Il y a encore peu de temps, les industriels de notre domaine suivaient une approche dite d'*assurance qualité basée sur les processus*. Cette approche consiste à examiner minutieusement les processus menant à la création d'un produit. Si les processus sont jugés comme étant de confiance, le produit final est également jugé de confiance. En avionique par exemple, la conception ainsi que la production des composants matériels sont suivi de près au niveau processus. Des échantillons sont ensuite préleés afin de tester leur cohérence avec la spécification du produit.

Les récentes normes avioniques DO-178C –et sont supplément DO-333 concernant les méthodes formelles– publiées par la RTCA[1] – montrent une volonté de consolider cette approche basée sur les processus, par une assurance qualité davantage basée sur les produits. C'est à dire de vérifier mathématiquement l'adéquation entre le produit final et sa spécification. Dans le domaine du logiciel, il existe des méthodes automatisables permettant de prouver de telles propriétés. Ces méthodes mises au point par la communauté informatique sont appelées *méthodes formelles*. Contrairement aux composants matériels, les logiciels sont constitués de "0" et de "1" pouvant être dupliqués sans erreur, renforçant l'intérêt de les vérifier.

Les systèmes critiques embarqués avioniques sont généralement développés au moyen de language haut-niveau (niveau modèle) différant du language d'implémentation final (niveau code). Les méthodes formelles fournissent des outils pour toutes les étapes du

[1] http://www.rtca.org/

développement :

- la spécification formelle [71, 35] propose des languages à la sémantique mathématique permettant d'exprimer ce que le système doit faire ;
- la vérification des modèles par rapport à leur spécification formelle [16, 17, 23, 55, 12, 6] permettent de prouver que la spécification est respectée ;
- la génération de code à partir des modèles [13] produit du code respectant la sémantique du language de conception haut-niveau, préservant ainsi l'effort de vérification au moyen de compilateurs certifiés ou prouvés.

L'étude présentée dans ce mémoire se concentre sur la vérification automatique de modèles. En effet, même si les organismes de certification encouragent leur utilisation, les méthodes formelles ne sont pas capables à l'heure actuelle de résoudre les problèmes rencontrés dans l'industrie. En pratique la vérification de propriétés fonctionnelles arbitraires sur des systèmes réalistes demande souvent un travail d'expert autant sur le plan des systèmes analysés que sur la (les) technique(s) employée(s). L'essort des méthodes formelles dans le monde industriel est limité par ces interventions coûteuses financiérement et temporellement.

L'ambition de cette thèse est d'améliorer l'état de l'art de la vérification formelle dans notre domaine d'application en *(i)* améliorant son automatisation sur des systèmes correspondant à des patrons de conception rencontrés dans les systèmes avioniques ; *(ii)* trouvant des preuves pouvant être rapidement vérifiées par différents outils afin d'assurer leur véracité. Pour atteindre ces objectifs nous proposons d'étudier la collaboration de méthodes formelles existantes –telles que la résolution SMT, la k-induction, l'élimination de quantificateurs, *etc.*– afin d'introduire une méthodologie visant à la découverte automatique de lemmes facilitant la conduite d'une preuve. De plus, cette étude s'intéresse en particulier aux algorithmes parallèles, suivant la tendance actuelle des multi-cœurs.

Cette étude s'intéresse à l'analyse de *systèmes réactifs embarqués* [38]. Ces systèmes échantillonnent leur environnement au moyen de capteurs, et calculent une commande destinée à un actuateur. L'échantillonage est effectué à intervalles réguliers spécifiés par la fréquence du système. Plus particulièrement nous considérons des composants logiciels de systèmes réactifs, contribuant à la sûreté de *chaînes fonctionnelles*, *i.e.* de systèmes rassemblant des capteurs, des unités de calcul en réseau, des actuateurs, *etc.*

Nous introduisons plus précisemment ces systèmes dans le Chapitre 1, dans lequel nous introduisons également les notions sous-jacentes à leur représentation mathématique,

ainsi que l'état de l'art en vérification formelle. Nous introduisons ensuite notre contribution principale dans le Chapitre 2. Tout d'abord nous proposons des améliorations à l'algorithme d'élimination de quantificateurs de Monniaux [56] permettant de calculer des pré-images sur nos systèmes industriels. Nous exposons ensuite nos heuristiques de génération d'invariants potentiels dans le cadre d'un framework collaboratif basé sur la k-induction, et présentons nos expérimentation sur nos systèmes. La Section 2.4 présente Stuff[2], notre implémentation de l'architecture et des techniques présentées en Section 2.2. Enfin, le Chapitre 3 conclut sur les résultats de l'étude et offre quelques éléments de perspective.

[2] Suff's The Ultimate Formal Framework

CHAPTER 1

Introduction

Ce chapitre est un résumé en francais de la Partie I du mémoire. Comme mentionné précédemment, l'aspect logiciel de ces systèmes est tout d'abord dévelopé au niveau modèle avant que le code bas niveau destiné à des architectures temps réel ne soit généré automatiquement en préservant la sémantique du modèle. Nous nous intéressons en particulier au langage flot de données synchrone Lustre [36] décrit en détail dans le Chapitre 4.1. Un modèle Lustre a une sémantique de système de transition dans lequel des calculs instantannés sont effectués à une certaine fréquence afin de produire un signal de sortie en fonction des signaux d'entrée. Nous présentons les systèmes auquels nous nous intéressons, appelés *chaînes fonctionnelles*, en Section 1.1. Avant d'analyser un modèle Lustre, il est nécessaire d'en extraire sa sémantique sous forme d'un système de transition. Ce chapitre présente, en Section 1.2, les notions mathématiques nécessaires pour la représentation et l'analyse des systèmes de transitions. La Section 1.3 présente l'état de l'art des méthodes s'intéressant à leur vérification.

1.1 Cas d'Étude

La Figure 1.1 présente une chaîne de calcul correspondant à la fonction de "controle de l'angle de tangage de l'avion". Les capteurs sont tripliqués afin de réduire le taux de défaillance de l'échantillonnage. Lors de chaque cycle, la chaîne échantillonne les capteurs de

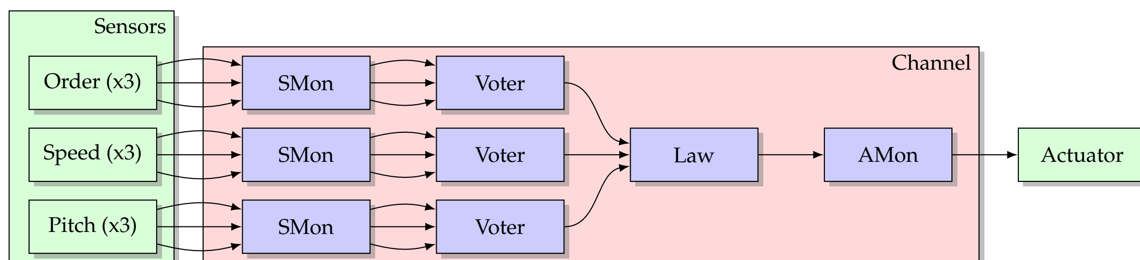


Figure 1.1: A Single Computation Channel.

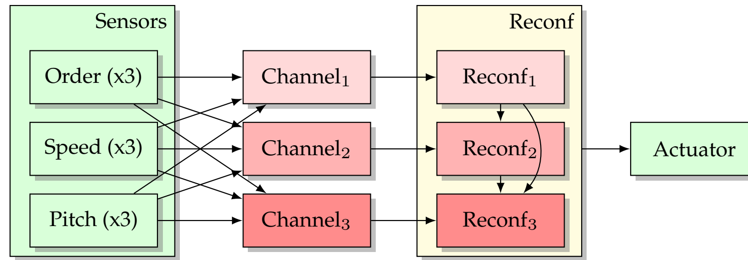


Figure 1.2: Triple Channel Functional Chain.

vitesse (Speed), de tangage (Pitch) ainsi que la commande produite par le pilote (Order). Des moniteurs de capteurs (SMon) examinent ces signaux afin de détecter des défaillances temporaires ou permanentes, telles que des valeurs en dehors de l'intervalle légal du capteur. Les signaux entrent ensuite dans un système de vote (Voter) chargé de produire un signal unique à partir de ses trois entrées. Le bloc suivant est la loi de commande (Law) de la chaîne qui calcule le signal commandant l'actuateur grâce à une unique valeur de vitesse, tangage et d'ordre du pilote. Le signal passe ensuite par un moniteur d'actuateur (AMon) s'assurant que la commande est correctement réalisée.

En pratique, les concepteurs de chaînes fonctionnelles tripliquent les canaux de calcul afin de réduire davantage le taux de défaillance. Comme présenté sur la Figure 1.2, les calculs sont effectués par trois chaînes redondantes évoluant sur des unités de calcul séparées et communiquant au moyen de câbles distincts. Un mécanisme de reconfiguration (Reconf) distribué sur les trois unités de calcul décide de la chaîne contrôlant effectivement l'actuateur, et est capable de reconfigurer le système dans le cas où une chaîne est jugée corrompue. Quels que soit les signaux produits par les différentes chaînes, la reconfiguration doit s'assurer qu'aucune commande dangereuse pour l'actuateur ou pour l'avion n'est produite par l'architecture tripliquée, et doit prévenir le pilote en cas de défaillance.

Il est à noter qu'une valeur de sortie incorrecte ne doit pas immédiatement faire basculer le statut de la chaîne la produisant à "corrompue". La défaillance peut n'être que temporaire, par exemple à cause d'une perturbation électromagnétique causée par le passage de la foudre à proximité de l'avion. C'est pourquoi la défaillance n'est considérée permanente que si la valeur de sortie reste incohérente pendant plus de n cycles.

Les chaînes fonctionnelles sont donc des systèmes extrêmement complexes, et il en est de même pour leur spécification [67]. La vérification de la partie logicielle d'une chaîne fonctionnelle complète est donc d'une grande difficulté pour les outils actuels. Même en suivant une approche modulaire, il loin d'être trivial de vérifier la correction des sous-systèmes séparément. L'objectif des travaux présentés dans ce mémoire est de fournir des

techniques génériques pour la vérification automatique de systèmes tels que ceux implémentant la logique de vote, ou la logique de reconfiguration. Ces systèmes utilisent largement l'arithmétique entière linéaire au travers des nombreux compteurs qu'ils utilisent pour confirmer les défaillances et transférer le contrôle de l'actuateur d'une chaîne à une autre. La logique de vote par exemple se doit de maintenir une certaine continuité dans les valeurs qu'elle produit : la loi de commande est pensée pour des valeurs en accord avec le monde réel et pourrait avoir un comportement imprévisible en cas de discontinuités irréalistes dans ses signaux d'entrée. Plus précisemment nous nous focaliserons sur le "triplex voter", un système fourni par Rockwell Collins implémentant une logique de vote. Le code de ce système est disponible dans l'annexe A. Les systèmes de loi de commande diffèrent du vote et de la reconfiguration car ils sont basés sur des équations différentielles. Ils peuvent être non-linéaires ou n'être analysable qu'au moyen de méthodes non-linéaires et sortent du cadre de cette étude.

1.2 Notations et Notions Mathématiques

1.2.1 Logique du Premier Ordre Multi-Sortée

La formalisation de la notion de système de transition s'appuie traditionellement sur le système formel appelé logique du premier ordre[1] multi-sortée, ou Many-Sorted First-Order Logic (MSFOL), défini en Section 5.1 de façon similaire au standard SMT-LIB 2 [4]. MSFOL permet de définir des logiques permettant à leur tour de construire des *termes* en combinant des symboles dans le cadre de règles assurant le typage des expressions obtenues.

Syntaxe. Les symboles représentent des fonctions n'ayant pas d'interprétation fixée, mais auquelles sont associées des *rangs* faisant intervenir les *types* ou *sortes* autorisés par la logique. Le symbole "+" par exemple aurait typiquement pour rangs $(\texttt{Int}, \texttt{Int}) : \texttt{Int}$ et $(\texttt{Real}, \texttt{Real}) : \texttt{Real}$. Le symbole "3", respectivement "3.0", peut avoir un unique rang $() : \texttt{Int}$, respectivement $() : \texttt{Real}$. Les sortes sont des noms et n'ont pas non plus, à ce niveau, d'interprétation. La notion de signature (Définition 5.1.1) déclare les sortes, les symboles, leurs rangs, ainsi que les *variables* –sur lesquelles nous reviendrons un peu plus tard– composant la logique. Les règles de typage définissent les termes légaux : avec les symboles et les rangs introduits ci-dessus par exemple, le terme $+(3, 3)$, c'est à dire "l'application du symbole + aux symboles 3 et 3" est bien typé, de sorte `Int`. Le terme $+(3, 3.0)$ au contraire n'est pas bien typé. Dans la suite de ce mémoire, un terme sera toujours supposé bien typé, sauf indication

[1] Les lecteurs initiés à la logique du premier ordre remarqueront que nous ne mentionnerons quasiment pas les quantificateurs dans cette section. En effet nous ne considérerons que des termes n'en utilisant pas par la suite. Les quantificateurs sont formellement traités dans la Section 5.1 de la partie anglaise du mémoire.

contraire. Il est à noter qu'en pratique, on impose que toutes les signatures contiennent au moins les sortes et les symboles –ainsi que leurs rangs– de la signature Core.

Core **signature.** Core est la signature composée du sorte `Bool` ainsi que des symboles et des signatures

$\vee$	(`Bool`, `Bool`) :	`Bool`	$\wedge$	(`Bool`, `Bool`) :	`Bool`	$\neg$	(`Bool`) :	`Bool`
$\top$	() :	`Bool`	$\bot$	() :	`Bool`	$=$	(`Bool`, `Bool`) :	`Bool`

Les termes de sorte `Bool` sont appelées *formules*, et les symboles "$\vee$", "$\wedge$", "$\neg$" sont appelés respectivement *disjonction* (ou), *conjonction* (et), *négation*. De plus, nous exigeons également que pour chaque sorte σ apparaissant dans une logique le symbole "$=$" ait, entre autres, le rang (σ, σ) : `Bool`.

Sémantique. Les notions introduites ci-dessus définissent la syntaxe de la logique. Pour donner une sémantique aux termes, il est nécessaire d'y associer une sémantique. Une *structure* (Définition 5.1.8) interprète les sortes d'une signature en ensembles de valeurs concrètes, et les symboles comme des fonctions sur ces ensembles de façon consistante avec leur rang. Une structure permet également d'interpréter des termes de la logique au moyen d'une fonction d'interprétation $\mathcal{I}$. Une application de symbole $s(terme_1, \ldots, terme_n)$ est évaluée récursivement :

$$\mathcal{I}(s)\Big(\mathcal{I}(terme_1), \ldots, \mathcal{I}(terme_n)\Big).$$

Une structure se doit de respecter les contraintes suivantes :

- l'interprétation du sorte `Bool` est l'ensemble $\{\mathbf{T}, \mathbf{F}\}$ (**T**rue et **F**alse) ;
- $\mathcal{I}(\top)$ est $\mathbf{T}$;
- $\mathcal{I}(\bot)$ est $\mathbf{F}$;
- $\mathcal{I}(\neg)(p)$ est $\mathbf{T}$ si et seulement si p est $\mathbf{F}$;
- $\mathcal{I}(\vee)(p, q)$ est $\mathbf{T}$ si et seulement si p **ou** q est $\mathbf{T}$;
- $\mathcal{I}(\wedge)(p, q)$ est $\mathbf{T}$ si et seulement si p **et** q est $\mathbf{T}$;
- $\mathcal{I}(=)(p, q)$ est $\mathbf{T}$ si et seulement si p est la même valeur concrète que q.

Étant donnée une signature Σ, une formule ϕ est *satisfiable* s'il existe une structure de Σ qui évalue ϕ à $\mathbf{T}$. Une telle structure est appelée un *modèle* de ϕ. Si toute structure de Σ est un modèle de ϕ, alors ϕ est une *tautologie* ou est *valide*. Validité et satisfiabilité sont

des notions duales. En effet, si ϕ est valide alors sa négation, $\neg\phi$, n'est pas satisfiable, ou *insatisfiable*. Inversement, si ϕ est insatisfiable alors $\neg\phi$ est valide.
Soit Σ un signature et ϕ et ψ deux formules. Alors

- ϕ et ψ sont *equisatisfiables* si ϕ est satisfiable si et seulement si ψ est satisfiable ;
- ψ est une *conséquence sémantique* de ϕ, noté $\phi \models \psi$, si tout modèle de ϕ est un modèle de ψ ;
- ψ est *équivalente sémantiquement* à ϕ, noté $\phi \equiv \psi$, si $\phi \models \psi$ et $\psi \models \phi$.

Satisfiabilité modulo théorie. La notion de sémantique introduite ci-dessus n'est pas suffisante pour les problèmes auquels nous nous intéressons. Malgré la possibilité d'obtenir des résultats tels que

- $\phi \wedge \neg\phi$ est insatisfiable,
- $\phi \vee \neg\phi$ est valide,
- $\neg(\phi \wedge \neg\phi) \equiv \phi \vee \neg\phi, \ldots$

Il n'est pas réaliste de raisonner au niveau arithmétique sans contraindre l'interprétation de symboles tels que "+", "42", "7.13" dans des signatures les autorisant. C'est pourquoi une logique est également composée d'une ou plusieurs théorie(s) restreignant l'interprétation des symboles et sortes non purement propositionnels, les symboles et sortes autres que ceux apparaissant dans Core. Une formule ϕ est alors *satisfiable modulo théorie* si elle admet un modèle autorisé par la ou les théories de la logique considérée. Dans la suite de ce mémoire, nous utiliserons majoritairement des fragments de MSFOL permettant de construire des termes arithmétiques linéaires réels ou entiers au moyen des symboles habituels "+", "$-$", "0", "1", ..., "0.0", "2.3", ... dont l'intepétation correspond à la sémantique usuelle de l'arithmétique. Nous dirons dans la suite de ce mémoire qu'une formule ϕ est satisfiable pour "satisfiable modulo théorie" pour des raisons de lisibilité.

Précisions sur la notion de variable d'une signature. Comme c'est le cas dans le standard SMT-LIB 2, les variables d'une signature ne peuvent apparaître dans des termes bien typés que si elles sont *liée* par un quantificateur ($\exists$ ou $\forall$) ou un *let-binding* (parallèle). Nous ne considérerons pas de termes quantifiés dans ce mémoire et omettons leur définition pour le moment ; davantage d'informations sont disponibles dans les sections 5.1.1 et 5.1.2 définissant formellement MSFOL, en particulier dans la Définition 5.1.3 pour la syntaxe abstraite, la Définition 5.1.4 pour le typage, et la Définition 5.1.10 pour leur sémantique. Nous utilisons

les let-binding-s pour définir formellement le déroulement de la relation de transition d'un système de transition en Section 6.1, mais nous n'en ferons pas usage dans ce résumé étendu en français par soucis de concision. Nous renvoyons donc le lecteur aux sections et définitions cîtées pour les quantificateurs traitant également de la syntaxe et de la sémantique des let-binding-s dans MSFOL.

Les solveurs SAT et SMT sont des outils extrêmement efficaces capables, étant donnée une formule ϕ exprimée dans un fragment de MSFOL, de répondre à la question "ϕ est-elle satisfiable?". Si c'est le cas, le solveur produit un modèle de ϕ. Par la suite nous abrégerons souvent "satisfiable" par "sat" et "insatisfiable" par "unsat". Le niveau de performance actuel de ces outils est le fruit de plusieurs décennies de recherche et d'optimisation déclenchées par les travaux précurseurs de [27, 28] mettant au point l'algorithme aujourd'hui standard, nommé DPLL en l'honneur de ses créateurs (Davis-Putnam-Logemann-Loveland). Les premiers solveurs ne s'intéressaient qu'au problème de la satisfiabilité de formules purement propositionnel, appelé "problème SAT". Les avancées considérables [9] dûes, entre autres, à l'équipe à l'origine du solveur SAT Chaff [59] ont à terme rendu possible d'adapter l'approche au problème de la satisfiabilité modulo théorie (SMT). La Section 5.2 de ce mémoire présente l'algorithme DPLL dans le cas SAT ainsi que dans le cas SMT au travers du formalisme *abstract DPLL* [60].
L'initiative SMT-LIB [4] vise a fédérer les différents solveurs en proposant des descriptions rigoureuses des théories qu'ils utilisent, ainsi qu'un langage d'entrée commun permettant la construction d'une librairie de benchmarks impressionnante. L'objectif est de faciliter l'utilisation et la comparaisons des solveurs disponibles. Les solveurs majeurs actuels sont compatible avec le récent standard SMT-LIB 2 [4], les principaux étant les suivants :

- Z3 [29], developpé à Microsoft Research,
- MathSAT 5 [20] de l'Université de Trento, et
- CVC4 [3], developpé conjointement par l'Université d'Iowa et l'Université de New York.

1.2.2 Systèmes de Transition

Les systèmes de transition encodent une sémantique de trace d'*états* partant d'états dits *initiaux*. Les états représentent l'état interne du système à un moment donné. Un système de transition encode les transitions légales pour le système lui permettant de passer d'un état à un autre. Les systèmes de transition ayant un nombre fini d'états peuvent, en théorie, être représentés de façon exhaustive en listant tous les états initiaux et tous les couples détats se succédant. En pratique malheureusement les systèmes finis que nous considérons ne sont

pas représentables de cette façon à cause du nombre gigantesque d'états qu'ils peuvent atteindre, sans parler des systèmes infinis. Il est donc plus réaliste de spécifier les états initiaux ainsi que la relation de transition au moyen de formules dans un fragment de MSFOL.

Étant donnée une logique $\mathcal{L}$, l'état interne du système de transition S est représenté par un vecteur de *variables d'états* $\mathsf{Var} \triangleq \langle v^1, \ldots, v^n \rangle$. Une fonction $\mathcal{D}$ associe à chaque variable un sorte de $\mathcal{L}$. Une *formule d'init* $I(\mathsf{Var})$ portant sur les variables d'états défini les états initiaux : ses modèles sont exactement les états initiaux du système. Un deuxième formule $T(\mathsf{Var}, \mathsf{Var}')$ porte sur Var vu comme "l'état courant" ainsi que sur une version primée de Var représentant "l'état suivant". Les modèles de cette *formule de transition* sont exactement les paires d'états du système se succédant.

Afin d'encoder des traces d'états semous forme de formules, nous augmentons la logique avec les symboles $v^1_{\mathbf{0}}, \ldots, v^n_{\mathbf{0}}, v^1_{\mathbf{1}}, \ldots, v^n_{\mathbf{1}}, \ldots$ que nous appelons des *variables d'état déroulées*. Nous notons la logique ainsi obtenue $\mathcal{L}(S)$. Par soucis de lisibilité, nous notons $s_{\mathbf{i}}$ le vecteur $\langle v^1_{\mathbf{1}}, \ldots, v^n_{\mathbf{1}} \rangle$. Il est à présent possible de construire une formule dont les modèles sont , par exemple, exactement les états atteignables en deux transitions à partir des états initiaux du système :

$$\phi \triangleq I(s_{\mathbf{0}}) \land T(s_{\mathbf{0}}, s_{\mathbf{1}}) \land T(s_{\mathbf{1}}, s_{\mathbf{2}}).$$

Si ϕ est satisfiable, un modèle de ϕ assigne des valeurs concrètes à $s_{\mathbf{0}}$, $s_{\mathbf{1}}$ et $s_{\mathbf{2}}$ telles que $s_{\mathbf{0}}$ est initial et les transition entre $s_{\mathbf{0}}$ et $s_{\mathbf{1}}$ et entre $s_{\mathbf{1}}$ et $s_{\mathbf{2}}$ sont légales.

Soit S_{cpt} le système de transition composé d'une seule variable d'état n de sorte `Int`, de la formule initiale $I(n) \triangleq (n = 0)$ et de la formule de transition $T(n, n') \triangleq (n' = n + 1)$. S_{cpt} représente un simple compteur initializé à zéro et s'incrémentant à chaque transition. Dans ce cas, la formule ϕ ci-dessus est donc

$$\phi \triangleq \overbrace{I(n_{\mathbf{0}})}^{n_{\mathbf{0}}=0} \land \overbrace{T(n_{\mathbf{0}}, n_{\mathbf{1}})}^{n_{\mathbf{1}}=n_{\mathbf{0}}+1} \land \overbrace{T(n_{\mathbf{1}}, n_{\mathbf{2}})}^{n_{\mathbf{2}}=n_{\mathbf{1}}+1}.$$

Il est alors possible de demander à un solveur SMT si ϕ est satisfiable, ce qui est le cas, puis d'en obtenir un modèle :

$$\{n_{\mathbf{0}} \mapsto 0,\ n_{\mathbf{1}} \mapsto 1,\ n_{\mathbf{2}} \mapsto 2\}.$$

Il est de plus possible d'ajouter des contraintes aux variables d'état déroulées, comme suit.

$$\overbrace{I(n_{\mathbf{0}})}^{n_{\mathbf{0}}=0} \land \overbrace{T(n_{\mathbf{0}}, n_{\mathbf{1}})}^{n_{\mathbf{1}}=n_{\mathbf{0}}+1} \land \overbrace{T(n_{\mathbf{1}}, n_{\mathbf{2}})}^{n_{\mathbf{2}}=n_{\mathbf{1}}+1} \land \overbrace{(n_{\mathbf{2}} \leq 0)}^{\text{contrainte sur } n_{\mathbf{2}}}$$

Le solveur SMT nous répond alors que la formule n'est pas satisfiable. Nous lui avons donc fait prouver qu'il n'était pas possible qu'un état à deux transitions d'un état initial soit négatif ou nul.

Il est important de noter à ce stade que les modèles de la formule de transition ne dénotent pas forcemment des états successeurs atteignables du système. Dans l'exemple décrit

ci-dessus, $\{n_{\mathbf{0}} \mapsto -10,\ n_{\mathbf{1}} \mapsto -9\}$ est un modèle de la formule $T(n_{\mathbf{0}}, n_{\mathbf{1}})$, mais ni l'un ni l'autre ne sont atteignable.

1.3 Vérification de Systèmes de Transition

Nous nous intéressons à la vérification de propriétés de sûreté sur des systèmes de transition. Ce problème correspond à une *analyse d'atteignabilité*. Étant donnés un système de transition S et un objectif de preuve PO, existe-t-il un entier $n \geq 0$ tel que

$$I(s_{\mathbf{0}}) \wedge T(s_{\mathbf{0}}, s_{\mathbf{1}}) \wedge \cdots \wedge T(s_{\mathbf{n\text{-}1}}, s_{\mathbf{n}}) \wedge \neg\mathrm{PO}(s_{\mathbf{n}}) \tag{1.1}$$

soit satisfiable ? Si c'est le cas un modèle peut être extrait et constitue un contre-exemple, c'est à dire une trace partant d'un état initial menant à une falsification de l'objectif de preuve PO. Si ce n'est pas le cas, PO est un *invariant* de S : aucun état atteignable du système ne falsifie PO. Un argument mathématique, une preuve en attestant doit être produit. Dans le reste de ce mémoire nous nous focaliserons sur la recherche de preuve dans le cas ou PO est effectivement un invariant de S.

1.3.1 K-induction

Pour ce faire nous utilisons la technique classique de la k-induction[2], consistant à trouver un entier $k \geq 1$ tel que deux formules soient unsat :

$$\begin{aligned}
Base_k(I, T, \mathrm{PO}) &\equiv \overbrace{I(s_{\mathbf{0}})}^{\text{Initial state}} \wedge \overbrace{\bigwedge_{i\in[0,k-2]} T(s_{\mathbf{i}}, s_{\mathbf{i+1}})}^{\text{trace of k-1 transitions}} \wedge \overbrace{\bigvee_{i\in[0,k-1]} \neg\mathrm{PO}(s_{\mathbf{i}})}^{\text{PO falsified on some state}} \\
Step_k(T, \mathrm{PO}) &\equiv \underbrace{\bigwedge_{i\in[0,k-1]} T(s_{\mathbf{i}}, s_{\mathbf{i+1}})}_{\text{trace of } k \text{ transitions}} \wedge \underbrace{\bigwedge_{i\in[0,k-1]} \mathrm{PO}(s_{\mathbf{i}})}_{\text{PO satisfied on first } k \text{ states}} \wedge \underbrace{\neg\mathrm{PO}(s_{\mathbf{k}})}_{\text{PO falsified by last state}} .
\end{aligned}$$

La formule $Base_k$ aussi appelée *instance de base* est satisfiable si une falsification de l'objectif de preuve peut être atteinte en $k-1$ transitions à partir des états initiaux. Un modèle de l'instance de base est donc un contre-exemple exhibant une falsification de l'objectif de preuve. Si $Base_k$ n'est pas satisfiable, alors tous les états atteignables en $k-1$ transitions à partir des états initiaux vérifient l'objectif de preuve.

La formule $Step_k$ aussi appelée *instance de step* n'est pas satisfiable si à partir du dernier état d'une trace de k états vérifiant PO, il n'est pas possible d'atteindre une falsification de PO en une transition.

Si les deux formules sont insatisfiable, toute trace partant des états initiaux ne peut atteindre une falsification de PO en $k-1$ transitions car l'instance de base n'est pas satisfiable. Le

[2]Introduite formellement dans la Section 6.2.

fait que $Step_k$ ne soit pas satisfiable permet de conclure qu'il n'est pas possible d'atteindre une falsification de PO en une transition à partir d'une trace initialisée de k états ($k-1$ transitions). Par récurrence, il n'existe donc pas d'entier $n \geq 0$ tel que l'Équation 1.1 soit satisfiable, et PO est bien un invariant de S.

En revanche, le fait que l'instance de step soit satisfiable ne permet pas de conclure. La formule de step ne prend pas en compte les états initiaux et il n'y a aucune garantie que ses modèles dénotent des états atteignables. On parle dans ce cas de *contre-exemple* fallacieux.

Il est cependant possible, en théorie, de chercher itérativement à établir une preuve de k-induction à partir de $k = 1$ en augmentant la valeur de k jusqu'à ce que les deux formules soient insatisfiable –ou que l'instance de base soit satisfiable, indiquant l'existence d'une falsification de PO.

Malheureusement, la complexité des formules augmente rapidement à chaque incrément de k et il n'est pas réaliste sur des systèmes réalistes de dérouler la formule de transition un grand nombre de fois. Les instances de base et de step deviennent rapidement trop complexe et le solveur SMT n'est plus capable de les traiter efficacement.

Pour remédier à ce problème il est nécessaire de *renforcer* l'objectif de preuve : en découvrant des invariants du système. L'intérêt est de contraindre l'instance de step dans le but de bloquer les contre-exemples fallacieux. Les invariants *non-relationels* sont en général faciles à découvrir car ils ne mentionnent qu'une seule variable d'état et correspondent à des bornes souvent présentes dans la description du système, comme par exemple la valeur maximale d'un compteur. Les invariants *relationels* en revanche expriment un lien entre plusieurs variables d'état et ne sont que rarement apparents ce qui rend difficile leur découverte. Ils sont en général plus informatifs que les invariants non-relationels et donc plus efficaces pour l'exclusion de contre-exemples de step fallacieux.

1.3.2 État de l'art

Les techniques formelles récentes se concentrent sur la recherche d'invariants du système permettant de renforcer un objectif de preuve afin de pouvoir le prouver par k-induction pour un k relativement petit.

L'interprétation abstraite (AI [23]) est une technique travaillant sur une abstraction de la sémantique du système qu'elle analyse. Cette approche est *correcte* dans la mesure ou tout invariant pour la sémantique absraite est également un invariant pour la sémantique concrète du programme. L'inverse n'est pas vrai cependant, AI n'est donc pas une technique *complète* car l'abstraction peut être trop grossière et donc rater des invariants pour la séman-

tique concrète. Les deux outils d'AI les plus intéressants dans notre cadre sont *(i)* Astrée [10] qui analyse du code C, et *(ii)* NBac [43] analysant des modèles Lustre en combinant des calculs de point fixes avants et arrières.
L'interprétation abstraite est une technique puissante, mais elle est loin d'être automatique. En effet de nombreux paramètres contrôle les techniques d'abstraction et doivent être spécifiés par un expert en AI ayant une bonne compréhension du système analysé.

Interpolation-based Model Checking (IMC) est une technique proposée par McMillan [55] basée sur les solveurs SMT et le calcul d'interpolants paramétré par un entier n. Cette approche consiste à calculer itérativement une suite de sur-approximations des états atteignables en n transitions excluant de plus en plus de contre-exemples de l'instance destep de n-induction à chaque itération. L'algorithme s'arrête quand il détecte que tous les contre-exemples de step sont bloqués, indiquant qu'un invariant du système permettant de prouver l'objectif de preuve par n-induction a été découvert.
Malheureusement IMC nécessite de dérouler la relation de transition un nombre relativement important de fois. Le méchanisme d'abstraction rend l'approche moins explosive que la k-induction, mais limite considérablement son passage à l'échelle sur des systèmes tels que les notres.

Property Directed Reachability (PDR, aussi appelé IC3 [12, 34]) ne construit pas de formule déroulant la formule de transition plus d'une fois. L'approche consiste a diviser la complexité de l'analyse en un grand nombre de problèmes plus simples. Une séquence de *frame*s bloquant des états dangereux est construite incrémentalement en mélangeant analyse arrière de la négation de l'objectif de preuve et propagation en avant prenant en compte les états initiaux. Cette technique a originellement été mise au point pour des systèmes purement propositionels ; des généralizations au cas SMT ont été proposées [40, 19] et montrent des résultats intéressants. Nous présentons notre propre généralisation dans la Section 8.3.

1.3.3 Motivation et Plan du Mémoire

La découverte d'invariants pour des systèmes industriels est un exercice difficile. Découvrir des invariants relationnels dirigés par l'objectif de preuve est cependant souvent indispensable au succès d'une analyse. Les méthodes décrites ci-dessus ne fonctionnent pas sur nos systèmes car ils ne sont pas capables d'inférer des invariants relationels, à l'exception de l'interprétation abstraite qui nécessite généralement une intervention experte pour les découvrir. Notre intuition est qu'il peut s'avérer plus efficace de les *deviner*, *i.e.* de produire des invariants potentiels grâce à une heuristique et de laisser à un moteur de k-induction le soin de séparer les invariants potentiels falsifiables de ceux étant prouvables.

Le moteur de preuve Kind [47] suit cette approche : des templates tirés d'une base de données sont instanciés sur les variables d'états et les constantes apparaissant dans le systèm [46]. Un moteur de k-induction est en suite chargé d'identifier les invariants. Cependant les invariants ainsi trouvés ne sont pas dirigés par l'objectif de preuve, il n'y a donc pas de garantie qu'ils aident à prouver l'objectif de preuve en bloquant des contre-exemples fallacieux. De plus, cette approche est problématique sur des systèmes complexe car le moteur de k-induction peut se retrouver noyé par le nombre d'invariants potentiels générés. Le fait que la base de données de templates grossit généralement avec le temps ne fait qu'augmenter ce risque.

Notre contribution se sépare en deux parties. Premiérement, nous définissons une architecture parallèle basée sur un moteur de k-induction permettant à des méthodes de découverte d'invariants de collaborer entre elles et avec la k-induction. Deuxiémement, nous proposons une heuristique de génération d'invariants potentiels dirigée par l'objectif de preuve. Nous démontrons sa capacité à découvrir des invariants relationnels renforçant les objectifs de preuve sur des systèmes standards dans le domaine des systèmes embarqués critiques avioniques.

Le Chapitre 2 décrit notre méthodologie. Nous commençons par améliorer l'algorithme d'élimination de quantificateurs (QE) introduit par Monniaux dans la Section 2.1, en termes de performance mais aussi de pouvoir d'expression des formules qu'il est capable de traiter. Nous introduisons ensuite notre architecture parallèle collaborative dans la Section 2.2 et présentons une technique de découverte d'invariants par QE dans ce cadre. Nous présentons HullQe dans la Section 2.3, notre heuristique générant des invariants potentiels dirigée par l'objectif de preuve combinant élimination de quantificateurs et calcul d'enveloppes convexes.
Enfin, nous concluons et donnons des perspectives pour des travaux poursuivant cette étude dans le Chapitre 3.

CHAPTER 2

Découverte d'Invariants par Élimination de Quantificateurs

L'élimination de quantificateurs est un problème fondamental en mathématiques, et constitue l'objet de recherches continues depuis des décennies. Cette section rapelle la chronologie des progrès majeurs en QE et présente les applications les plus fréquentes de la QE pour l'analyse de systèmes réactifs.

En 1951, Tarski prouve la décidabilité des corps algébriquement clos en exhibant une procédure d'élimination de quantificateurs [73]. Pour décider une formule, on élimine succsessivement toutes les variables quantifiées, chaque nouvelle formule obtenue étant equi-satisfiable à la précédente, pour obtenir une formule *ground*, dont la valeur de vérité peut facilement être réduite à *vrai* ou *faux*. L'algorithme de Tarski étant d'une complexité non élémentaire, il est d'une utilité pratiquement nulle pour toute application. En 1975, G. E. Collins publie la technique de *Décomposition Cylindrique Algébrique* (CAD) [21], avec une complexité de 2^{2^n} où n est le nombre de variables à éliminer, certes toujours élevée mais grandement améliorée par rapport à la technique de Tarski, et permettant d'envisager quelques applications. Les premières implémentations de CAD apparaissent au début des années 1980 et ont depuis été continuellement améliorées [41]. Un autre saut en performance est effectué par Weispfenning et Loos à la fin des années 1980 avec la méthode des *Substitutions Virtuelles* (VS) sur les réels, d'abord pour les systèmes linéaires [52], puis pour les systèmes au plus quadratiques [79] [80]. La méthode VS est très rapide mais tend à produire des formules dont la taille explose, si bien que récemment, certains travaux comme [72] proposent d'utiliser une combinaison de CAD, VS et de simplification de formule à la volée pour traiter des problèmes non linéaires de QE issus de problèmes de vérification difficiles. Aujourd'hui, des implémentations de ces techniques de QE générales sont disponibles dans des suites logicielles telles que QEPCAD, Redlog, Matlab, Maple, *etc.*.

Globalement, les méthodes de QE générales non linéaires restent très coûteuses, ce qui justifie l'intéret de la recherche sur les algorithmes de QE très efficaces pour l'arithmétique linéaire réelle, entière, ou combinée réelle-entière. En effet, le cadre linéaire suffit pour modéliser de nombreuses classes de systèmes très courants dans les applications critiques : les

automates hybrides linéaires, les automates temporisés, les programmes manipulant des compteurs entiers, les programmes synchrones mêlant Booléens, entiers et réels *etc.* La QE linéaire s'utilise principalement de la même manière que la QE générale : calcul de préimage [26], synthèse d'invariants par patrons [50], abstraction automatique de portions de programmes [57]. Donc, se doter de techniques efficaces pour ces fragments peut être extrêmement gratifiant dans les applications de vérification. La plus significative et récente des avancées dans ce domaine est celle de la *QE par énumération paresseuse et projection polyédrique* publiée dans [56], et reprise dans [58]. Cette méthode capitalise les derniers progrès de la *satisfiabilité modulo théories* (SMT), et apporte un gain de performance notable sur les approches préexistantes.

Cette méthode de QE basée SMT se trouve au cœur de notre cadre logiciel pour l'analyse de systèmes réactifs synchrones : dans [16], un calcul de préimage itéré est utilisé pour alimenter un générateur de lemmes potentiels afin de prouver une propriété sur un système de transitions. Dans cette application, c'est l'étape de QE qui constituait le principal goulot d'étranglement, ce qui nous conduisit à améliorer l'algorithme original de [56], en modifiant la phase d'*extrapolation* de l'algorithme.

2.1 Élimination de Quantificateurs

Definition 2.1.1 (Élimination de quantificateurs) Étant donnée une formule ϕ de QF_LRIA et une collection V de variables de ϕ, *l'élimination de quantificateurs* consiste à produire une autre formule $\mathcal{O}$ de QF_LRIA telle que $\mathcal{O}$ soit équivalente à $\exists V,\ \phi$ en *éliminant* les variables V de ϕ. Cette opération est notée $QE(V, \phi)$.

Par exemple, si $\phi \equiv (x = 39 \ \wedge \ y \geq 3 + x) \vee (x \ \geq 7 \ \wedge \ y = 11 - x)$ alors un résultat possible pour $QE(x, \phi)$ serait $\mathcal{O} \equiv y \geq 42 \ \vee \ y \leq 4$.

L'algorithme d'*élimination de quantificateurs par énumération paresseuse de modèles* de [56] est donné dans l'algorithme 1. Dans l'article original, ϕ est supposée appartenir au fragment QF_LRA et V ne contient donc que des variables réelles. Dans cet algorithme, chaque itération comporte trois phases. Premièrement, *l'extraction de modèle*, qui utilise un solveur SMT pour trouver un modèle de $\phi \wedge \neg\mathcal{O}$ tant qu'il en existe, sinon l'algorithme termine. Ensuite, la phase d'*extrapolation* qui produit, à partir d'un modèle M de ϕ, une nouvelle formule $\mathcal{E}$ qui se veut plus générale que le modèle dont elle est issue et qui implique ϕ. Une définition formelle de la notion d'extrapolant, tirée de [62], est donnée ci-après :

Definition 2.1.2 (Extrapolant) Soient deux formules A et B. Une formule C est un extrapolant de A et B si elle est telle que :

- si $A \wedge B$ est **unsat** alors $C \equiv \bot$
- si $A \wedge B$ est **sat** alors $A \wedge C$ est **sat** et $C \wedge \neg B$ est **unsat**.

Algorithm 1 QE par énumération paresseuse de modèles.

Require: ϕ : une formule QF_LRIA
Require: V : collection de variables à éliminer de ϕ.
Ensure: $\mathcal{O}$: une formule en forme DNF telle que $\mathcal{O} \equiv \exists V, \phi$
  function $QE(V, \phi)$
    $\mathcal{O} \leftarrow \bot$
    while *isSatisfiable*$(\phi \land \neg\mathcal{O})$ **do**
      $M \leftarrow getModel(\phi \land \neg\mathcal{O})$ ▷ génération d'un modèle
      $\mathcal{E} \leftarrow extrapolate(\phi, M)$ ▷ extrapolation du modèle
      $P \leftarrow project(\mathcal{E}, V)$ ▷ projection de l'extrapolant
      $\mathcal{O} \leftarrow \mathcal{O} \lor P$ ▷ construction incrémentale du résultat
    end while
    return $\mathcal{O}$ ▷ retrouner la formule DNF
  end function

Cette définition est instanciée de la manière suivante dans le cadre de la QE : si M est un modèle de ϕ, en prenant $A \equiv (\bigwedge_{v \in Vars(\phi)} v = M(v))$ pour caractériser le modèle et $B \equiv \phi$, un extrapolant $\mathcal{E}$ est tel que :

- $A \land \mathcal{E}$ est satisfiable, *i.e.* $M(\mathcal{E}) \equiv M(\phi) \equiv \top$, *i.e.* $M \models \mathcal{E}$;
- $\mathcal{E} \land \neg\phi$ est insatisfiable, *i.e.* $\mathcal{E} \implies \phi$.

Par extension, le calcul d'un tel $\mathcal{E}$ est appelé *"extrapoler M par rapport à ϕ"*.

Dans la QE de [56], l'extrapolation de M est réalisée par la technique appelée *SMT-test*, par laquelle on considère l'ensemble des atomes de ϕ noté $Atoms(\phi)$, et l'on construit *l'ensemble des littéraux impliqués par le modèle* :

$$\mathcal{L} \equiv \{\text{si } M(a) \text{ alors } a \text{ sinon } \neg a \mid a \in Atoms(\phi)\} \tag{2.1}$$

On a alors $(\bigwedge_{l \in \mathcal{L}} l) \implies \phi$, *i.e.* $(\bigwedge_{l \in \mathcal{L}} l) \land \neg\phi$ est insatisfiable. Ensuite, on minimise $\mathcal{L}$ en en retirant successivement chaque littéral l' tel que $((\bigwedge_{l \in \mathcal{L},\, l \neq l'} l) \land \neg\phi)$ devienne satisfiable. Enfin, on retourne l'extrapolant $(\bigwedge_{l \in \mathcal{L}_{min}} l)$.

Enfin, la phase de *projection* élimine de $\mathcal{E}$ les variables de V par une projection polyédrique, fournie par exemple par la Parma Polyhedra Library [2]. Le résultat de chaque projection nourrit la construction de la formule DNF résultat $\mathcal{O}$, qui est équivalente à $(\exists V,\ \phi)$ et ne contient plus aucune variable de V lorsque l'algorithme termine, *i.e.* lorsque $\phi \land \mathcal{O}$ n'a plus de modèle.

Il est important de retenir le rôle clef joué par l'extrapolation dans l'algorithme de QE. C'est en effet sa capacité à généraliser le modèle qui va permettre de limiter le nombre d'itérations de la boucle principale. Plus un extrapolant est général, plus sa projection est

générale, et sa négation bloque davantage de modèles de la formule ϕ. Dans l'algorithme original, la plus grande partie de l'exécution est passée à généraliser les modèles. C'est ce qui justifie notre travail sur l'extrapolation afin d'améliorer l'algorithme de QE.

2.1.1 Projection combinée d'atomes booléens, entiers et réels

L'algorithme de QE original imposait que les formules appartiennent au fragment linéaire réel QF_LRA (variables réelles, relations et combinateurs booléens uniquement). Nous généralisons ici l'approche à QF_LRIA, ce qui implique d'éliminer des variables pouvant être booléennes, entières ou réelles. C'est la méthode de projection qu'il faut adapter ici. En effet, en supposant qu'un solveur SMT compatible QF_LRIA soit utilisé, les phases d'énumération de modèles et de génération d'extrapolants restent inchangées, l'essentiel du raisonnement spécifique aux théories arithmétiques et booléennes étant délégué au solveur SMT.

Un extrapolant n'est maintenant plus un polyèdre de variables réelles, mais un cube c contenant des littéraux booléens, entiers et réels. Compte tenu de l'absence d'intersection de types entre entiers et réels dans le langage QF_LRIA, ce cube peut être divisé en trois cubes c_{bool}, c_{int} et c_{real} tels que c_{bool} (*resp.* c_{int}, c_{real}) ne contient que des littéraux booléens (*resp.* entiers, réels), et $c \equiv c_{bool} \wedge c_{int} \wedge c_{real}$.

L'ensemble V des variables à éliminer peut lui aussi être divisé en trois sous-ensembles disjoints V_{bool}, V_{int} et V_{real} tels que V_{bool} (*resp.* V_{int}, V_{real}) ne contient que des variables booléennes (*resp.* entières, réelles). Enfin, la propriété suivante nous permet d'obtenir trois problèmes de projection distincts : projection des variables booléennes, entières et réelles, qui peuvent être traitées en parallèle avec des méthodes dédiées.

Property 2.1.1 (Séparation de quantificateurs) Soient deux formules ϕ_1 et ϕ_2, et deux ensembles de variables V_1 et V_2 tels que $V_1 \cap FV(\phi_2) = \emptyset$, $V_2 \cap FV(\phi_1) = \emptyset$, et $FV(\phi_1) \cap FV(\phi_2) = \emptyset$, alors $(\exists V_1 \cup V_2,\ \phi_1 \wedge \phi_2) \equiv (\exists V_1,\ \phi_1) \wedge (\exists V_2,\ \phi_2)$.

Cette adaptation de l'algorithme de QE à QF_LRIA permet d'aborder des systèmes typiques du domaine avionique, dans lesquels les booléens supportent la logique de contrôle, les entiers supportent l'implémentation de diverses primitives temporelles telles que les timers, les watchdogs, *etc.* et les réels supportent les flots de données.

Ces systèmes s'expriment dans la théorie des octogones entiers et nous permettent d'éviter d'avoir à recourir à l'opérateur modulo lors de l'élimination de quantificateurs. En effet, $QE(y, x = 7 * y)$ où x et y sont des entiers devrait normalement renvoyer $x\%7 = 0$ où $\%$ est l'opérateur "modulo". En nous limitant aux octogones, nous pouvons donc projeter les atomes entiers comme des atomes réels. Dans un contexte plus général le résultat de la projection pourrait être faux. Avec notre méthode, $QE(x, \{0 < x,\ x < 1\})$ où x est un entier

aurait pour résultat $\top$; cependant, l'algorithme de QE de Monniaux ne projetant que des atomes satisfiables modulo théorie, ce genre de cas ne peut donc pas se présenter au sein de la procédure.

2.1.2 Extrapolation : utilisation des cœurs insatisfiables

Le but de l'extrapolation est de dériver une formule $\mathcal{E}$ à partir d'un modèle $M \models \phi$ qui soit plus générale que M, tout en impliquant ϕ. Si nous considérons tous les extrapolants pouvant être construits à partir des atomes de ϕ, un *meilleur extrapolant* pour un modèle M aurait un *nombre minimal de littéraux*. Un tel extrapolant peut être obtenu en prenant la conjonction du sous-ensemble de littéraux $\mathcal{U} \subseteq \mathcal{L}$ impliqué dans tout cœur insatisfiable minimal de la formule $((\bigwedge_{l \in \mathcal{L}} l) \wedge \neg\phi)$, où $\mathcal{L}$ est l'ensemble défini dans l'équation (7.1).

Calculer des extrapolants très généraux permet de limiter le nombre total d'itérations de la boucle principale de QE. Cependant, l'extraction de cœurs insatisfiables minimaux pour déterminer $\mathcal{U}$ se révèle trop coûteuse en pratique. Il est bénéfique pour la performance globale de la QE de se contenter d'approximer les meilleurs extrapolants. Le coût des quelques itérations supplémentaires de la boucle principale, du à la moindre qualité des extrapolants, est amorti par le gain sur l'extrapolation plus rapide.

L'extrapolation *SMT-test* décrite en section 2.1 a de bonnes capacités de généralisation, mais reste coûteuse en raison des tests SMT utilisés pour décider, pour chaque littéral de la formule, de l'éliminer ou de le garder dans l'extrapolant. Le nombre de tests SMT peut être réduit en utilisant les informations découvertes par le solveur SMT sur les formules insatisfiables. Lorsqu'un littéral $l' \in \mathcal{L}$ tel que $G \equiv ((\bigwedge_{l \in \mathcal{L},\, l \neq l'} l) \wedge \neg\phi)$ est insatisfiable est considéré, le solveur peut être interrogé pour savoir quels littéraux ont été utilisés dans le graphe du dernier conflit. Ces littéraux font partie d'un cœur unsat de la formule G, pas nécessairement minimal, et on peut éliminer de $\mathcal{L}$ tous les littéraux n'étant pas dans ce graphe. On élimine ainsi plus d'un littéral de l'extrapolant par itération. Cette idée est détaillée dans l'algorithme 2.

2.1.3 Extrapolation par analyse structurelle

Limiter l'utilisation du solveur SMT dans l'extrapolation est un progrès, mais la supprimer totalement pourrait être encore mieux. Nous présentons ici une analyse structurelle permettant d'identifier rapidement un sous-ensemble de littéraux expliquant la satisfiabilité d'une formule par rapport à un modèle, et d'en générer un extrapolant. Cette méthode est inspirée de [11], où les modèles découverts par un model-checker pour Simulink sont simplifiés avant d'être présentés à l'utilisateur. Les portions du réseau à flots de données impliquées dans la falsification de l'objectif de preuve sont identifiées par des règles structurelles. De [11], nous retenons principalement les notions *d'activité* et de *cause*. L'idée de

Algorithm 2 SMT-test avec cœurs insatisfiables.

Require: ϕ une formule QF_LRIA, M un modèle de ϕ.
Ensure: Un extrapolant de M par rapport à ϕ sous la forme d'un ensemble de littéraux.

```
function smtTestCores(φ, M)
    toCheck ← ℒ                                  ▷ Ensemble ℒ de littéraux à tester (cf. Eq 7.1)
    toKeep ← ∅
    while toCheck ≠ ∅ do                          ▷ ensemble des littéraux de l'extrapolant
        solver.assert(¬φ)
        literal ← toCheck.first                   ▷ prendre le prochain littéral à tester
        toCheck ← (toCheck \ {literal})           ▷ supprimer le littéral de l'ensemble
        for lit ∈ toKeep ∪ toCheck do
            solver.assert(lit)                    ▷ supposer les littéraux à garder/tester
        end for
        if solver.checksat() = sat then
            toKeep ← (toKeep ∪ {literal})         ▷ ajouter le littéral dans l'ensemble résultat
        else                    ▷ supprimer les littéraux n'étant pas dans le cœur unsat
            toCheck ← solver.getUnsatCore() ∩ toCheck
        end if
        solver.restart()              ▷ réinitialiser le solveur pour la prochaine itération
    end while
    return toKeep             ▷ retourner l'ensemble de littéraux formant l'extrapolant
end function
```

base ici est de ne pas juste considérer l'ensemble des littéraux de la formule "à plat", mais plutôt de garder une vision arborescente de la formule et de définir des règles permettant d'identifier les sous-arbres expliquant la valeur de la racine.

Definition 2.1.3 (Activité d'une sous-formule) Soit f une formule et g une de ses sous-formules directes. On dit que g est *active* par rapport à f dans M si et seulement si la valeur de g dans M est telle que la changer en préservant les valeurs des autres sous-formules directes de f change la valeur de f, ou bien si cette valeur permet de déterminer complètement la valeur de f dans M.

L'activation d'une sous-formule peut être résumée à une condition logique sur les valeurs impliquées par un modèle pour la sous-formule et ses soeurs. Par exemple, si on considère que $a \wedge b$, une sous-formule de ϕ, est active, alors une des situations suivantes se produit : (a) si $M(\neg a)$ et $M(b)$, la valeur de a dans M suffit à déterminer que $a \wedge b$ est fausse dans M, mais pas la valeur de b. Donc a est active et b ne l'est pas ; (b) si $M(a)$ et $M(\neg b)$, la valeur de b dans M suffit à déterminer que $a \wedge b$ est fausse dans M, mais pas la valeur de a. Donc b est active et a ne l'est pas ; (c) si $M(\neg a)$ et $M(\neg b)$, alors la valeur de a ou celle de b dans M permet indépendamment de déterminer que $a \wedge b$ est fausse dans M. Donc a et b sont

actives, c'est un cas *combinaison de causes indépendantes* ; (d) si $M(a)$ et $M(b)$, la connaissance des valeurs de a et b est nécessaire pour déterminer que $a \wedge b$ est vraie dans M. Donc a et b sont actives, c'est un cas de *combinaison de causes dépendantes*. La condition d'activation de a peut être réduite à $M(\neg a \vee b)$, et celle de b à $M(\neg b \vee a)$.

Definition 2.1.4 (Causes) Soient une formule ϕ et un de ses modèles M, une *cause* pour une sous-formule g active par rapport à ϕ est un cube k sur les variables libres de ϕ tel que $M \models k$ et pour tout M' tel que $M' \models k$, $M'(g) = M(g)$. Puisqu'il peut exister plusieurs causes pour g dans un même modèle, l'ensemble des causes de g est noté $K_M(g)$, ou bien $K(g)$ quand M est évident d'après le contexte.

L'ensemble des causes d'une formule peut être construit inductivement à partir des ensembles de causes de ses sous-formules. Pour l'opérateur unaire $\neg a$, les causes sont celles de a. Pour un opérateur binaire $\langle op \rangle \in (\wedge, \vee)$, $K_M(a \langle op \rangle b)$ est égal à :

- $K_M(a)$ si seulement a est active ;
- $K_M(b)$ si seulement b est active ;
- $K_M(a) \cup K_M(b)$ si a et b sont actives et leurs causes sont indépendantes ;
- $\{C_a \wedge C_b \mid (C_a, C_b) \in K_M(a) \times K_M(b)\}$ si a et b sont actives et leurs causes sont dépendantes.

Ces règles de construction et la définition d'une cause font que si a (resp. b) était active par rapport à $a \langle op \rangle b$ dans M, alors elle l'est toujours dans tout $M' \models k$ où $k \in K_M(a \langle op \rangle b)$. Donc, si $M \models \phi$, tout $k \in K_M(\phi)$ est en fait un extrapolant de M par rapport à ϕ, car $M \models k$ et tout $M' \models k$ est tel que $M'(\phi) = M(\phi) = \top$, ce qui est équivalent à $k \implies \phi$, on retrouve bien la définition 7.1.1 instanciée dans le cadre de la QE.

Il est donc possible de calculer des extrapolants pour M par rapport à ϕ en calculant des causes pour ϕ dans M. Nous avons constaté expérimentalement que les combinaisons de causes indépendantes augmentent la taille des ensembles de causes (par union), menant indirectement à une explosion du nombre de causes lors du traitement des causes dépendantes (produit cartésien). Nous avons donc décidé de n'extraire qu'un seul extrapolant par modèle, en ne traitant que les combinaisons de *causes dépendantes* et en ne calculant que la cause de la sous-formule active la plus à gauche lors de combinaisons de *causes indépendantes*. L'algorithme 7 donne le parcours récursif de l'arbre de la formule et la combinaison des causes utilisés pour générer des extrapolants.

Le Tableau 2.1 résume toutes les règles définissant les fonctions $active$ et $combine$. La colonne *Terme* spécifie l'opérateur en question. La colonne *Condition* spécifie la condition logique à évaluer. La colonne *Sous-formule active* donne le comportement de la fonction

Algorithm 3 Extrapolation par méthode structurelle.

Require: f une formule, M une fonction d'évaluation.
Ensure: Si $M \models f$ alors le résultat est un extrapolant de M par rapport à f.
```
function struct(f, M)
    activeSubFormulas ← active(f, M)                  ▷ sous-formules actives (cf. Tableau 2.1)
    subCauses ← activeSubFormulas map {x → struct(x, M)}            ▷ appel récursif
    return combine(f, subCauses, M)          ▷ combinaison des sous-causes (cf. Tableau 2.1)
end function
```

active, qui retourne la liste des sous-formules actives d'une formule. La colonne *Causes générées/Combinaison* donne le comportement de la fonction *combine*, qui retourne soit une cause fraîchement générée dans les cas d'arrêt de la récursion (quand *active* renvoie une liste vide), ou bien une cause obtenue par combinaison. Par exemple, la cinquième ligne du tableau se lit : si la formule courante est de la forme $x = y$, la récursion s'arrête et alors si $M(x)$ est égal à $M(y)$, la cause $x = y$ est retournée, sinon si $M(x) < M(y)$, alors la cause $x < y$ est retournée, sinon si $M(y) < M(x)$, la cause $y < x$ est retournée.

Terme	Condition	Sous-formules actives	Causes générées/ Combinaison
$\langle id \rangle$ booléen	$\mathbf{T}$	[]	$\langle id \rangle$ si $M(\langle id \rangle)$, $\neg\langle id \rangle$ sinon
$\neg a$	$\mathbf{T}$	$[a]$	$K(a)$
$a \wedge b$	$M(\neg a)$	$[a]$	$K(a)$
	$M(a \wedge \neg b)$	$[b]$	$K(b)$
	$M(a \wedge b)$	$[a, b]$	$K(a) \wedge K(b)$
$a \vee b$	$M(a)$	$[a]$	$K(a)$
	$M(\neg a \wedge b)$	$[b]$	$K(b)$
	$M(\neg a \wedge \neg b)$	$[a, b]$	$K(a) \wedge K(b)$
$x = y$	$M(x = y)$	[]	$(x = y)$
	$M(x < y)$	[]	$(x < y)$
	$M(y < x)$	[]	$(y < x)$
$x < y$	$M(x < y)$	[]	$(x < y)$
	$M(\neg(x < y))$	[]	$\neg(x < y)$

Figure 2.1: Règles de calcul de l'analyse structurelle.

2.1.4 Combinaison de techniques d'extrapolation

L'analyse structurelle peut être intégrée dans l'algorithme 2, en remplaçant simplement la première ligne $toCheck \leftarrow \mathcal{L}$ par $toCheck \leftarrow struct(\phi, M)$ où *struct* est la fonction de

Algorithm 4 $hybrid(\phi, M)$ ▷ Extrapolation hybride.

Require: ϕ une formule, M un modèle de ϕ.
Ensure: Un extrapolant de M par rapport à ϕ, sous la forme d'un ensemble de littéraux.

```
toCheck ← struct(φ, M)            ▷ ensemble de littéraux par analyse structurelle (cf. Alg. 7)
toKeep ← ∅
solver.assert(¬φ)                 ▷ insérer ¬φ au niveau 0
solver.push(1)                    ▷ à niveau 1 (littéraux à garder)
while toCheck ≠ ∅ do
    literal ← toCheck.first       ▷ prendre un littéral dans l'ensemble toCheck
    toCheck ← (toCheck \ {literal})   ▷ le supprimer de l'ensemble
    solver.push(1)                ▷ à niveau 2 (littéral à tester)
    for lit ∈ toCheck do
        solver.assert(lit)        ▷ insérer le littéral à tester à niveau 2
    end for
    if solver.checksat() = sat then
        toKeep ← (toKeep ∪ {literal})
        solver.pop(1)
        solver.assert(literal)    ▷ insérer le littéral à garder à niveau 1
    else                          ▷ éliminer les littéraux hors du cœur insatisfiable
        solver.pop(1)
        toCheck ← (solver.getUnsatCore() ∩ toCheck)
    end if
end while
return toKeep
```

l'algorithme 7. Pour encore améliorer la performance, on peut utiliser la pile d'assertions du solveur SMT sous-jacent. L'algorithme 8 illustre cette combinaison. La formule $\neg\phi$ est insérée au bas de la pile d'assertions, les littéraux à garder pour former l'extrapolant sont placés au niveau 1 de la pile, le littéral à tester est inséré en haut de la pile. Un simple *pop* nous ramène au niveau 1, où l'on peut insérer le littéral à garder et passer au prochain littéral à tester après un *push*, préservant ainsi au sein du solveur la majorité du contexte de l'algorithme.

2.1.5 Évaluation de l'Approche

Nous avons implémenté chacune des techniques d'extrapolation discutées ci-dessus ainsi qu'un algorithme de QE générique paramétré par la fonction d'extrapolation. Le solveur SMT Microsoft Z3 [29] est utilisé pour couvrir les besoins SMT. Pour la projection sur contraintes arithmétiques, décrite en section 2.1.1, nous avons utilisé la Parma Polyhedra Li-

brary [2][1], la projection de Booléens étant réalisée par nos soins. Dans le tableau 7.4, nous présentons les résultats obtenus avec les techniques d'extrapolation suivantes :

- *SMT-test* la technique de [56], fournie comme référence ;
- *struct* la technique purement structurelle de l'algorithme 7 ;
- *hybrid* la technique mêlant analyse structurelle, *SMT-test* et cœurs insatisfiables de l'algorithme 8.

Les formules de test sont des problèmes de vérification générés par notre outil Stuff [16], et correspondent à des itérations de calcul de préimage symbolique à partir de la négation d'un objectif de preuve sur un système de transition. Plus précisément, si $T(s, s')$ est la relation de transition du système, appliquée à l'état courant s et à un successeur s', le calcul de préimage itéré est défini ainsi :

$$\mathcal{G}_1(s) \equiv QE(s', PO(s) \wedge T(s, s') \wedge \neg PO(s'))$$
$$\mathcal{G}_{i+1}(s) \equiv QE(s', PO(s) \wedge T(s, s') \wedge \mathcal{G}_i(s'))$$

La formule de chaque nouvelle itération contient donc le résultat de l'itération précédente et grossit rapidement.

Comparaison de performances expérimentales.

		Reconf					Triplex		
		1	2	3	4	5	1	2	3
Durée totale ms	SMT-test	53 172	114 371	99 953	131 855	318 257	13 855	30 163	119 771
	struct	**800**	**883**	**2 063**	**5 587**	**10 078**	**1 627**	**4 785**	**23 413**
	hybrid	25 078	27 837	31 024	50 160	51 913	7 005	16 978	42 063
Nb. cubes dans $\mathcal{O}$	SMT-test	55	106	125	119	264	33	38	84
	struct	37	**38**	84	153	243	26	31	51
	hybrid	**37**	39	**45**	**70**	**74**	**23**	**28**	**48**
Durée ms / itération	SMT-test	966	1 078	799	1 108	1 205	419	793	1 425
	struct	**21**	**23**	**24**	**36**	**41**	**62**	**154**	459
	hybrid	677	713	689	716	701	304	606	876
Extrapol. ms / itération	SMT-test	938	1 049	768	1 075	1 163	369	618	919
	struct	**1.810**	**0.684**	**0.595**	**0.483**	**0.427**	**0.115**	**0.290**	**0.117**
	hybrid	652	687	661	686	667	261	361	392
Projection ms / itération	SMT-test	11	9	6	8	9	**8**	**11**	**14**
	struct	**5**	**6**	**5**	**5**	**5**	19	30	42
	hybrid	10	10	9	9	8	**8**	**11**	**14**
Check-sat ms / itération	SMT-test	16	19	24	23	32	42	163	491
	struct	**14**	**16**	18	31	35	43	**123**	**416**
	hybrid	**14**	**16**	**17**	**21**	**25**	**34**	234	469

Nous donnons ici les résultats sur deux systèmes[2] : le premier est une logique triplex fournie par Rockwell-Collins, qui mêle Booléens et réels, pour laquelle nous calculons les

[1]Outil disponible à `https://cavale.enseeiht.fr/redmine/projects/smt-qe/files`
[2]Disponibles à `https://cavale.enseeiht.fr/redmine/projects/smt-qe/files`

trois premières préimages ; le second est une logique de reconfiguration distribuée mêlant Booléens et entiers, pour laquelle nous calculons les cinq premières préimages. Nous ne les détaillons pas plus ici et dirigeons le lecteur intéressé vers [16].

Le tableau 7.4 montre la performance des techniques d'extrapolation au travers de différentes métriques. D'abord, la durée totale d'exécution en millisecondes. Ensuite, le nombre d'itérations de la boucle principale de QE, qui correspond aussi au nombre de cubes dans la formule DNF résultat. Nous donnons ensuite la durée moyenne d'une itération de la boucle principale de QE, en détaillant les contributions de : (i) l'extrapolation, (ii) la projection, (iii) l'obtention du modèle.

Les tests nous ont permis de remarquer que la durée d'exécution et le nombre total d'itérations de la QE avec *SMT-test* pouvait varier grandement, jusqu'à $\pm 50\%$ d'une exécution à l'autre, sur une même formule. Ceci est dû à la sensibilité à l'ordre dans lequel *SMT-test* considère chaque littéral et – comme pour les méthodes *struct* et *hybrid*, à l'ordre dans lequel les modèles sont découverts. La méthode *struct* est la plus stable de toutes avec des variations de durée d'exécution de seulement $\pm 5\%$.

Les bénéfices de l'analyse structurelle sans tests SMT sont clairement mis en avant : la méthode de QE avec extrapolation *struct* est de loin la plus rapide de toutes les techniques, parfois de plusieurs ordres de grandeur (cf. Reconf 5, *struct* termine en 10 secondes contre 300 secondes pour *SMT-test*, en produisant un nombre de cubes comparable). La contrepartie est aussi apparente : en ne considérant pas finement la sémantique des littéraux manipulés, mais seulement leur rôle structurel dans la formule, des littéraux inutiles peuvent s'accumuler dans l'extrapolant et le rendre moins général. Mais, dans les rares cas où les DNF produites contiennent plus de cubes (cf. Reconf 4), elles sont malgré tout produites beaucoup plus rapidement qu'avec les autres techniques.

Si la taille de la formule produite est importante pour l'application en question, le meilleur compromis entre rapidité d'exécution et pouvoir de généralisation semble être atteint avec la méthode d'extrapolation *hybrid* : elle produit des formules DNF jusqu'à trois fois plus petites en nombre de cubes que *struct* ou *SMT-test* (cf. Reconf 5, 74 cubes en 50 secondes contre 243 cubes en 10 secondes avec *struct*, et 264 cubes en 300 secondes pour *SMT-test*), montrant donc une capacité de généralisation supérieure. En outre, *hybrid* s'est avérée beaucoup plus rapide que la technique *SMT-test* de référence dans tous les tests conduits, et la variance des temps d'exécution est aussi beaucoup plus faible qu'avec *SMT-test*, de l'ordre de $\pm 10\%$.

2.2 Une Architecture Collaborative

Notre architecture est paramétrée par un entier n indiquant la profondeur maximale à laquelle le moteur de k-induction est autorisé à dérouler la formule de transition. La Figure 2.2 illustre le fonctionnement de l'architecture à haut niveau. Des méthodes génèrent

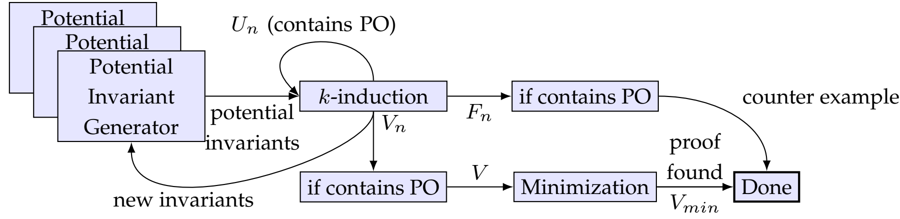


Figure 2.2: Une architecture collaborative basée sur la k-induction.

des invariants potentiels et les envoient au moteur de k-induction en charge d'identifier ceux étant effectivement des invariants. Pour ce faire la k-induction construit de manière incrémentale trois ensembles F_n, U_n et V_n à partir d'un ensemble d'invariants potentiels s contenant l'objectif de preuve comme suit. L'ensemble F_n contient les invariants potentiels falsifiables par n-induction, *i.e.* ceux pour lesquels l'instance de base est satisfiable. L'ensemble U_n contient ceux pour lesquels l'instance de base est insatisfiable mais l'instance de step est satisfiable : leur statut est non-défini (Undefined). Enfin, l'ensemble V_n contient les invariants potentiels prouvables par n-induction.

Si l'objectif de preuve PO est dans l'ensemble V_n, alors l'analyse est terminée, PO a été renforcé et est prouvable par n-induction. Si PO est dans l'ensemble F_n, alors un contre-exemple existe et PO n'est pas un invariant. Si PO est dans U_n alors la k-induction recommence l'analyse décrite ci-dessus avec les nouveaux invariants potentiels générés par les autres méthodes de l'architecture. Les invariants découverts durant une analyse –les éléments de V_n– sont quant à eux communiqués aux méthodes de génération d'invariants potentiels dans le but de raffiner les analyses qu'elles effectuent.

Nous illustrons à présent la découverte d'invariants par instanciation de templates *via* élimination de quantificateurs bénéficiant des améliorations apportées à l'algorithme de Monniaux présentées en Section 2.1. Cette méthode, appelée BrutalIQe, est fortement inspirée des travaux de Kapur [49, 50]. Nous supposons l'existence d'une base de données de fonctions de la forme $template(vars)(params)$ retournant une formule, comme par exemple

$$template(var_1, var_2)(min, max) \triangleq (min \leq var_1 + var_2) \wedge (var_1 + var_2 \leq max).$$

L'argument $vars$ de la fonction représente des variables d'état du système et $params$ les paramètres de la template. L'idée est de trouver des valeurs p pour ces paramètres. Notons tout d'abord que le fait que la formule produite est (1-)inductive est équivalent à l'insatisfiabilité de la formule

$$\overbrace{(I(s) \wedge \neg template(sv)(p))}^{\text{instance de base}} \quad \vee \quad \overbrace{(template(sv)(p) \wedge T(s, s') \wedge \neg template(sv')(p))}^{\text{instance de step}}.$$

Nous construisons ensuite la formule suivante, dont les modèles sont exactement les valeurs des paramètres pour lesquelles la template est inductive :

$$\forall s,\ s',\ \neg\Big(\quad (I(s) \wedge \neg template(sv)(fresh)) \\ \vee \quad (template(sv)(fresh) \wedge T(s,s') \wedge \neg template(sv')(fresh))\Big)$$

où *fresh* est un vecteur de nouvelles variables. Cette formule peut être réécrite en

$$\neg\Big(\exists s, s', \quad (I(s) \wedge \neg template(sv)(fresh)) \\ \vee \quad (template(sv)(fresh) \wedge T(s,s') \wedge \neg template(sv)(fresh))\Big).$$

Il est à présent possible d'utiliser l'élimination de quantificateurs pour éliminer les variables d'état du système pour obtenir un caractérisation sans quantificateurs des paramètres :

$$\mathcal{C} \triangleq \neg QE\Big(s \cup s', \quad (I(s) \wedge \neg template(sv)(fresh)) \\ \vee \quad (template(sv)(fresh) \wedge T(s,s') \wedge \neg template(sv')(fresh))\Big).$$

$\mathcal{C}$ ne porte que sur les variables *fresh* ; si elle est insatisfiable, alors aucune valeur de *fresh* ne peut rendre la template inductive sans davantage de renforcement. Si au contraire $\mathcal{C}$ est satisfiable, un modèle fournit des valeurs pour les paramètres de la template permettant d'obtenir des invariants inductifs par construction. La formule obtenue est alors envoyée au moteur de k-induction dans l'espoir de renforcer l'objectif de preuve mais également afin de communiquer l'invariant découvert aux autres méthodes du framework.

En pratique, nous utilisons BrutalIQe pour découvrir des bornes sur les variables détat du système. Cela permet de réduire fortement l'espace d'états considéré par la k-induction et les autres méthodes, leur permettant de raffiner leurs analyses.

2.3 Découverte d'Invariants Potentiels par QE

Nous abordons à présent la contribution principale de ce mémoire, une heuristique de génération d'invariants potentiels combinant le calcul de préimages par élimination de quantificateurs avec le calcul d'enveloppes convexes. L'approche, nommée HullQe, bénéficie grandement des améliorations apportées à l'algorithme de QE de Monniaux présentées dans la Section 2.1 et se décompose en deux parties. Premièrement, un calcul de pré-images de la négation de l'objectif de preuve PO construit itérativement une caractérisation des *états gris*. Les états gris sont des états vérifiant l'objectif de preuve desquels une falsification

de PO peut être atteinte en un certain nombre de transitions. Dans un second temps, une analyse des partitionnements de la représentation partielle des états gris essaie d'infèrer des relations entre les variables envoyées au moteur de k-induction en tant qu'invariants potentiels.

2.3.1 Calcul de Pré-images

Cette première étape de notre heuristique de génération d'invariants potentiels consiste à calculer une caractérisation de tous les états gris pouvant atteindre une falsification de l'objectif de preuve PO en une transition. Ce calcul est itéré afin d'identifier des états gris de plus en plus éloignés des états falsifiant PO. En pratique, nous nous intéressons à la suite de formules suivante :

$$\begin{aligned} \mathcal{G}_1(s) &\triangleq QE(s', \mathrm{PO}(s) \wedge T(s,s') \wedge \neg\mathrm{PO}(s')) \\ \mathcal{G}_i(s) &\triangleq QE(s', \mathrm{PO}(s) \wedge T(s,s') \wedge \mathcal{G}_{i-1}(s')) \quad (\text{for } i > 1). \end{aligned}$$

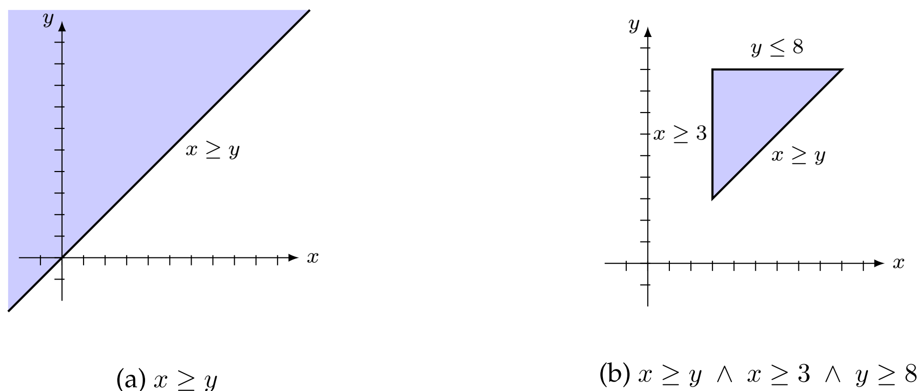
Ainsi, $\mathcal{G}_i(s)$ caractérise tous les états pouvant atteindre une falsification de l'objectif de preuve en i transitions ou moins. Si pour un certain i, $\mathcal{G}_{i+1}(s)$ ne caractérise pas davantage d'états que $\mathcal{G}_i(s)$, un point fixe est atteint et tous les états gris ont été découverts. Si aucun d'eux n'intersecte les états initiaux, *i.e.* $(\bigwedge_{0\leq j\leq i} \mathcal{G}_j(s)) \wedge I(s)$ n'est pas satisfiable, alors PO est un invariant du système.

Malheureusement, il n'est pas réaliste de s'attendre à atteindre un point fixe sur des système complexe tels que ceux qui nous intéressent. Les caractérisations partielles des états gris sont donc envoyées au fur et à mesure au second composant de notre heuristique en charge de faire apparaitre des relations entre les variables par calcul d'enveloppes convexes.

2.3.2 Extraction d'Invariants Potentiels par Calcul d'Enveloppes Convexes

Notons tout d'abord que l'algorithme que nous utilisons pour le calcul de pré-images a la propriété de produire des formules en forme normale disjonctive ou DNF, *i.e.* des disjonctions de conjonctions de littéraux. Dans cette section nous utiliserons couramment le terme *polyhèdre* pour nous référer à une conjonction de littéraux. D'un point de vue géometrique, une formule en DNF correspond à une union de polyhèdres tels que ceux représenté sur la Figure 2.3a et la Figure 2.3b.

En raison du grand nombre de polyhèdres apparaissant dans les pré-images calculées sur nos systèmes, il n'est pas réaliste de calculer toutes les enveloppes convexes possible. De plus, il est important de préserver la localité des polyhèdres. Il est donc nécessaire d'établir des critères conditionnant le calcul des enveloppes covexes dans le but d'identifier les groupes de polyhèdres proches les uns des autres.



(a) $x \geq y$ (b) $x \geq y \;\wedge\; x \geq 3 \;\wedge\; y \geq 8$

Figure 2.3: Polyhedra examples.

Enveloppes Convexes Exactes

Le premier critère que nous proposons consiste à ne valider l'enveloppe convexe h de deux polyhèdres p_1 et p_2 que si celle-ci est exacte ; c'est à dire si elle décrit exactement les mêmes points que l'union des deux polyhèdres dont elle est issue. Plus précisemment, cela signifie que h a exactement les mèmes modèles que $p_1 \vee p_2$. Ce critère est particulièrement intéressant pour des polyhèdres dont les littéraux sont dans l'arithmétique entière linéaire. La Figure 2.4 illustre les nouvelles relations pouvant apparaître lors du calcul d'ECH sur des polyhèdres entiers. Dans le cas réel, aucune de ces enveloppes convexes ne serait exacte, et de façon plus générale il n'est pas possible de faire apparaître de nouvelles relations entre des variables réelles au moyen d'ECH. Nous verrons que nous utilisons un critère différent pour les traiter.

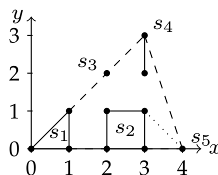


Figure 2.4: ECH de polyhèdres entiers.

Nous proposons un algorithme hautement optimisé décrit dans la Section 9.2.1 permettant de calculer efficacement toutes les ECHs calculables à partir d'un ensemble non redondant de polyhèdres. Cela nous permet de faire apparaître de nouvelles relations entre les variables d'états dirigées par le calcul de pré-images présenté dans la section précédente.

Cependant, les enveloppes convexes ainsi générées caractérisent des états gris, c'est à dire des états que nous souhaitons bloquer lors d'une tentative de preuve par k-induction.

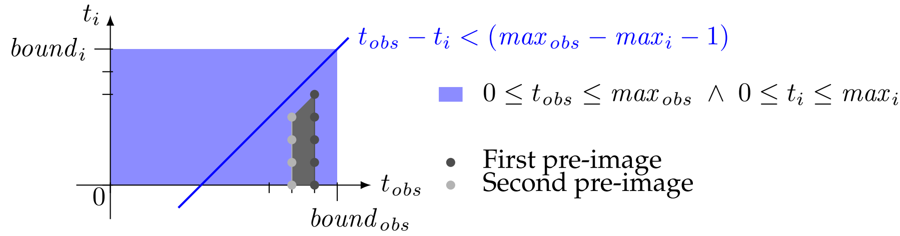


Figure 2.5: Découverte d'invariant relationnels sur le système de reconfiguration.

C'est pourquoi l'extraction d'invariants potentiels considère l'ensemble des littéraux découverts indépendamment de l'enveloppe convexe à laquelle ils appartiennent. Les invariants potentiels envoyés à la k-induction sont finalement l'ensemble des négations de chacun de ces littéraux. Les éléments de cet ensemble confirmés comme étant des invariants par le moteur de k-induction sont garantis de bloquer au moins une partie des contre-exemples de step fallacieux apparaissant dans un tentative de preuve par k-induction.

Nous illustrons à présent l'intérêt de cette approche sur notre système de reconfiguration. L'objectif de preuve PO de ce système est que "aucune chaîne n'est en contrôle de l'actuateur" ne peut pas être vrai pendant plus de n cycles. Pour détecter une falsification de PO un observateur du système utilise un compteur s'incrémentant tant qu'aucune chaîne de calcul n'est en contrôle, et est remis à zéro sinon. Aucun outil de vérification formelle n'est capable, à notre connaissance, de conclure sur ce système. La difficulté de l'analyse réside dans le besoin d'identifier des relations entre les compteurs du système pour la confirmation des défaillances, ainsi que le transfert de contrôle d'une chaîne à une autre.

Grâce au calcul exhaustif d'enveloppes convexes exactes, notre outil est capable de renforcer l'objectif de preuve en quelques secondes. Après le calcul de la deuxième pré-image, 150 invariants potentiels sont envoyés à la k-induction. 70 sont prouvés et renforcent PO. Il s'avère que seulement trois sont nécessaire au renforcement de l'objectif de preuve, exprimant des relations entre le compteur de l'observateur et ceux apparaissant dans le système de reconfiguration. La Figure 2.5 illustre le processus de découverte de ces relations.

Il est à noter que les résultats obtenus ne dépendent pas de la valeur des bornes des compteurs, auxquelles sont sensibles la plupart des autres techniques formelles. Ces compteurs peuvent être amenés à compter des centaines, voire des milliers de cycles suivant la spécification du système. La k-induction, IMC, ainsi que PDR doivent dérouler la relation de transition un nombre de fois proportionnel ce qui provoque la divergence de l'analyse.

Enveloppes Convexes Modulo Intersection

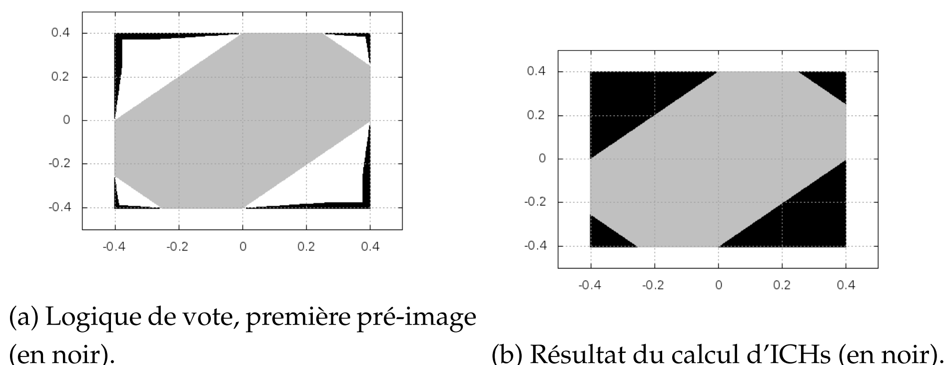
Le second critère que nous proposons consiste à n'accepter l'enveloppe convexe h de deux polyhèdres p_1 et p_2 que si p_1 et p_2 ont au moins un point en commun, c'est à dire si $p_1 \wedge p_2$ est exacte. Si h est légale par ce critère, nous dirons que h est l'ICH de p_1 et p_2.

Ce critère n'est pas plus permissif que ECH. En effet, les ECH présentées ci-dessus dans le cadre de la vérification du système de reconfiguration ne sont **pas** légales par rapport à ce second critère. Les ICHs diffèrent également des ECH de par le fait qu'elle font apparaître de nouveaux points dans les enveloppes convexes qu'elles calculent. Leur but est de sur-approximer des régions de l'espace d'états gris tout en évitant de fusionner des polyhèdres distants.

Sur nos systèmes, le nombre d'ICH calculable est en général beaucoup trop grand pour adopter une approche exhaustive. C'est pourquoi nous proposons, dans la Section 9.2.4, un algorithme consistant à éliminer les polyhèdres fusionnables et à les remplacer par leur ICH avant de rechercher d'autres combinaisons de polyhèdre légales. Ce faisant, une abstraction des zones disjointes de l'espace d'états gris est calculée. Nous appliquons ensuite la même stratégie d'extraction d'invariants potentiels que pour les ECHs : L'ensemble des négations de chaque littéral apparaissant dans le résultat de l'algorithme est envoyé à la k-induction.

Nous illustrons l'intérêt de cette approche sur le système de logique de vote fournit par Rockwell Collins. L'objectif de preuve que nous considérons est que si les entrées du système sont bornées, alors c'est également le cas de sa sortie. Des invariants relationels renforçant la propriété ont été trouvés à la main [32] au prix de longues simulations, une approche inadaptée à la vérification de systèmes critiques dans le cadre d'un processus de développement. À notre connaissance, aucun outil de vérification formelle n'est capable de vérifier ce système. La difficulté de l'analyse réside dans la nécessité de trouver des invariants reliant les trois variables d'égalisation du système desquelles dépend directement la valeur du signal de sortie. Grâce au calcul d'ICHs, notre outil est capable de vérifier ce système industriel en quelques secondes. Environ 60 invariants potentiels sont générés, dont 30 sont prouvés par k-induction et renforcent l'objectif de preuve. Il s'avère que seulement deux sont nécessaires. La Figure 2.6 illustre le processus de découverte de ces invariants, les invariants trouvés par simulation sont représentés en gris clair. La Figure 2.6a montre la première préimage de la négation de l'objectif de preuve en noir, alors que la Figure 2.6b montre le résultat du calcul d'ICH.

Comme c'est le cas pour le calcul d'ECHs sur le système de reconfiguration présenté dans la section précédente, l'analyse menée ici n'est pas sensible à la taille des intervalles de validité supposés sur les entrées du système de vote ni de celui à prouver sur sa sortie. Ce n'est pas le cas de la k-induction, ni d'IMC ou de PDR qui doivent dérouler la relation de



(a) Logique de vote, première pré-image (en noir).

(b) Résultat du calcul d'ICHs (en noir).

Figure 2.6: Logique de vote en dimension deux.

transition un nombre de fois proportionnel à la taille de ces intervalles, et donc ne parviennent pas à conclure.

2.4 Le Framework Formel *Stuff*

Nous avons implémenté l'architecture collaborative parallèle décrite dans la Section 2.2 dans notre framework formel Stuff (**S**tuff's **T**he **U**ltimate **F**ramework). Ce dernier est écrit en Scala et utilise le framework Akka qui propose une API, basée sur le formalisme des acteurs [39], pour le développement de logiciels parallèles. Nous détaillons notre utilisation des acteurs dans la Section 10.1 du mémoire. Le framework Akka confère à Stuff la possibilité de spécifier, après compilation, comment déployer ses composants sur plusieurs coeurs, voire même sur plusieurs machines au travers d'un réseau.

De plus, Stuff est également extensible dans l'implémentation de nouvelles techniques qui se fait sans impact sur le code existant. Comme décrit dans la Section 10.2, les techniques sont gérées à l'exécution au moyen du patron de conception des chaînes de responsabilité, permettant au framework de gérer des techniques avec un minimum de connaissance sur la façon dont elles fonctionnent.

Enfin, les solveurs SMT utilisés par Stuff peuvent être spécifiés après compilation, grâce à une librairie générique implémentée par nos soins. Cette librairie, appelée *Assumptio*, est détaillée dans le Chapitre 11. Assumptio est compatible avec les principaux solveurs SMT respectant le standard SMT-LIB 2 : Z3, CVC4 et MathSat 5.

Stuff s'est montré très utile dans nos récents travaux visant à la vérification d'une chaîne fonctionnelle complète. En le combinant avec un interprétateur abstrait [65] développé à l'Onera, nous avons pu vérifier la stabilité en boucle ouverte de la chaîne représentée en Fig-

ure 2.7. Bien que considérablement plus simple que les chaînes fonctionnelles industrielles, la vérification de cet exemple exécutable est un premier pas vers l'analyse de systèmes plus complexe. Les moniteurs de capteurs (Sat) sont vérifiés par interprétation abstraite en utilisant des intervalles en tant que domaines abstraits. Stuff se charge de vérifier la correction des systèmes de vote au moyen de l'analyse basée sur les enveloppes convexes modulo intersection présentée en Section 2.3.2. Enfin, l'interpréteur abstrait conduit une analyse utilisant des ellipsoides permettant de prouver la stabilité de la loi de commande. La stabilité de la chaîne est ainsi vérifiée de bout en bout.

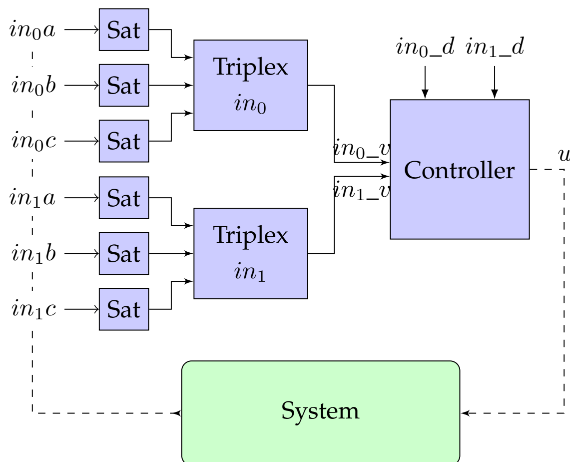


Figure 2.7: Une chaîne fonctionnelle simplifiée.

CHAPTER 3

Conclusion

3.1 Conclusion

La vérification de logiciels est un domaine de recherche très actif, mais malgré les impressionnantes avancées de ces dernières décennies certains systèmes restent hors de portée des techniques actuelles. Le but de cette étude était d'améliorer l'état de l'art sur le problème de la vérification automatique des composants logiciels issus de systèmes critiques embarqués avioniques, présentés dans le Chapitre 1. Nous nous sommes en particulier concentrés sur la découverte d'invariants relationels dans le cadre d'un framework collaboratif centré autour d'un moteur de k-induction. Nous espérons que nos travaux aideront à la diffusion de la vérification formelle dans la communauté industrielle.

Tout d'abord, nous avons formalisé la notion de systèmes de transition en termes de prédicats dans MSFOL (logique du premier ordre multi-sortée) en Section 1.2.1 et Section 1.2.2. La formalisation complète est donnée en Chapitre 5 et Chapitre 6. Nous avons défini une catégorie de signatures MSFOL dans laquelle les notions de trace, d'objectif de preuve, de validité, *etc.* sont précisemment formalisés. Ce type de représentation est très répandu dans la littérature, mais les détails mathématiques et en particulier la sémantique précise du déroulement de la relation de transition sont souvent omis.

Nous avons présenté notre contribution principale dans le Chapitre 2, exposée en détail dans la Partie II. Nous avons commencé par améliorer l'algorithme d'élimination de quantificateurs (QE) de Monniaux sur l'arithmetique réelle linéaire en Section 2.1 (correspondant au Chapitre 7). Cela nous a permis de gagner plusieurs ordres de grandeur dans le calcul de pré-images sur des systèmes de transition, gain mis en évidence par nos expérimentations sur un grand nombre de benchmarks. Nous avons également adapté l'algorithme aux formules mélangeant arithmétique réelle, octogones entiers et booléens. Nous avons ensuite proposé des méthodes de découverte d'invariants profitant des améliorations apportées à l'algorithme de QE. Premièrement, une méthode basée sur des templates, appelée BrutalIQe, s'inspirant des travaux de Kapur [49] en Section 2.2 (*cf.* Section 8.2). En pratique cette technique est utilisée pour decouvrir des invariants sous forme d'intervalles en-

cadrant les variables d'état du système. Dans un deuxième temps, nous avons proposé, HullQe, une nouvelle heuristique de découverte d'invariants en Section 2.3 –correspondant au Chapitre 9. HullQe calcule des pré-images d'états dangereux et étudie leurs partitionnements pour en extraire des invariants potentiels relationels. Enfin, nous avons proposé une architecture de moteur de preuve parallèle collaboratif centré autour de la k-induction en Section 2.2 (voir Section 8.1). Cette architecture permet aux techniques de découverte d'invariants de coopérer grâce à l'intégration continue des invariants découverts.

Stuff est l'implémentation de cette architecture, présentée en Section 2.4 –et détaillée dans la Partie III. Nous avons présenté le formalisme des acteurs sur lequel Stuff est bâti, ainsi que son caractère extensible, notamment par rapport à l'ajout de technique sans impact sur les techniques déjà implémentées, et par rapport aux solveurs SMT utilisés pouvant être spécifiés après compilation.

Nous avons évalué l'intêret de notre approche collaborative proposée dans ce mémoire sur un système de reconfiguration et un système de vote fourni par Rockwell Collins. Stuff conclut en quelques secondes grâce à la collaboration entre les différentes méthodes du framework.

Enfin, nous avons étudié la combinaison de notre outil avec un interpréteur abstrait [65] dans l'optique de la vérification d'une chaîne fonctionnelle complète. Grâce à cette combinaison nous avons pu vérifier automatiquement une partie d'une chaîne fonctionnelle comportant des entrées correspondant à des capteurs tripliqués, des moniteurs de capteurs, des systèmes de vote et une loi de commande contrôllant un actuateur. Les résultats obtenus sur un système exécutable représentatif sont une avancée vers la vérification de chaînes fonctionnelles industrielles.

3.2 Perspectives

3.2.1 Élimination de Quantificateurs

L'élimination de quantificateurs (QE) est une brique de base utilisée par toutes les techniques de découverte d'invariants présentées dans ce mémoire. Une direction naturelle de recherche est donc d'étendre notre technique de QE à des fragments plus généraux, tels que l'arithmétique entière linéaire et l'arithmétique réelle non-linéaire. De telles extensions impacteraient l'ensemble du framework et permettraient d'évaluer la pertinence de notre approche sur des systèmes s'exprimant dans des fragments plus complexes. De ce point de vue, les limitations se situent au niveau de la phase de projection. En effet, les solveurs SMT actuels sont déjà capables de traiter ces fragments, et notre analyse structurelle est tout à fait compatible avec eux.

Pour ce qui est de la projection de (conjonction de) littéraux, les techniques Cylindrical Algebraic Decomposition (CAD [21]) et Virtual Substitution (VS [79, 80]) sont capables de manipuler l'arithmétique entière non-linéaire. Malheureusement, elles sont loin de passer à l'échelle sur des problèmes d'élimination de quantificateurs extraits de la vérification de systèmes industriels tels que les notres. Augmenter le pouvoir d'expression des formules d'entrée de la procédure de QE ne doit pas se faire au détriment de ses performances : toutes nos techniques, excepté la k-induction, s'appuient fortement sur la rapidité de la procédure de QE. Changer la phase de projection pour CAD ou VS causerait une baisse en performance conséquente, et pourrait rendre le framework dans son ensemble inutilisable. Nous pensons que davantage de recherche sur ce sujet est nécessaire, par exemple en identifiant des sous-fragments de l'arithmétique non-linéaire pour lesquels des algorithmes de QE efficaces existent. Une autre direction d'intêret est la combinaison des approches susnommées [72].

3.2.2 HullQe

La méthode HullQe est composée de deux parties : un calcul incrémental de pré-image et les heuristiques étudiant celles-ci afin d'en extraire des invariants potentiels. Des invariants relationel pertinents sont découverts en fusionnant les différents polyhèdres apparaissant dans les pré-images. L'explosion du nombre d'invariants potentiels générés est limitée par des critères conditionnant la fusion de deux polyhèdres. Une première heuristique n'autorise une fusion que si celle-ci est exacte, la deuxième impose que les deux polyhèdres aient au moins un point en commun. Il serait intéressant d'évaluer des critères plus permissifs sur des systèmes réalistes.

De plus, introduire un mécanisme d'abstraction dans le calcul de pré-image pourrait accélérer le calcul des états gris et permettre de découvrir de nouveaux invariants. Des travaux récents [22] introduise une technique d'abstraction adaptable *a priori* dans le calcul de pré-images : si une trace contre-exemple partant des états initiaux est découverte, un algorithme détermine quelles approximations la rendent possible –dans le cas où cette trace ne constitue pas un vrai contre-exemple– et les annule.

3.2.3 Property-Directed Reachability

PDR est une technique très prometteuse, conçue à l'origine pour des systèmes purement propositionnels. Généraliser cette approche au cas SMT n'est pas simple : les auteurs de [19] proposent une adaptation de PDR à l'analyse de programmes sous la forme de Control Flow Graphs (CFG) contenant de l'arithmétique. Ils proposent une version "ramifiée" de PDR qui réduit l'explosion combinatoire de l'espace de recherche en bloquant des cubes pour des branches abstraites du propramme au lieu d'essayer de les bloquer pour le système entier. Il serait très intéressant d'adapter cette approche à l'analyse de programmes Lustre.

3.2.4 Stuff

Les travaux présentés dans ce mémoire omettent le cas où l'objectif de preuve n'est pas un invariant du système considéré. Il est alors important de produire un ou plusieurs contre-exemple(s) de manière efficace, ce qui peut s'avérer problématique pour des systèmes réalistes. PDR prend en compte les états initiaux tout au long de l'analyse, et effectue une exploration en arrière des antécédents de la négation de l'objectif de preuve. Cette stratégie rend PDR capable de trouver des contre-exemples bien plus rapidement que le BMC et la plupart des autres méthodes formelles, *a fortiori* dans sa version ramifiée.

La recherche de contre-exemples est d'un grand intêret pour les industriels de notre domaine et dans le cadre de la coûteuse phase de détection et correction de bugs. Malheureusement les traces contre-exemple produites par les méthodes formelles sont généralement constituées d'états concrets trouvés par des solveurs SMT. Il est donc souvent difficile pour les dévelopeurs d'identifier l'origine du bug dans le modèle, en particulier pour des traces de centaines ou de milliers d'états. PDR diffère de la plupart des autres méthodes et produit, non pas une unique trace, mais une classe de contre-exemples sous la forme d'une séquence de cubes. Il serait intéressant d'évaluer l'intêret de ces contre-exemples abstraits pour les concepteurs de systèmes critiques lors de la phase de correction de bugs dans un contexte industriel.

Enfin, nous projetons de continuer nos travaux visant à la vérification de chaînes fonctionnelles complètes grâce à la combinaison de notre framework et de l'interpréteur abstrait introduit dans [65]. Nous insistons sur le besoin de cas d'étude, idéalement industriels. Nous avons étudié avec succès une famille importante de propriété en boucle ouverte, mais davantage de recherche est nécessaire pour étendre et améliorer ces résultats. En particulier pour automatiser la décision de la méthode à employer pour analyser chaque sous-système. Il est également primordial d'assurer le passage à l'échelle de l'approche aux système réels, pouvant utiliser plus de dix fois plus de variables d'état.

PART I

Introduction

Critical systems are systems the failure of which has catastrophic consequences such as human casualties, economic loss, *etc.* They are typically encountered in fields such as aerospace, avionics, automotive, rail transport, nuclear plants, medical equipment, hardware design... There is thus a strong interest in *(i)* establishing precise specifications of those systems, and *(ii)* making sure that the final product is correct with respect to the specifications. The work presented in this thesis focuses on the latter in the context of software components of avionics critical embedded systems.

Until recently, the industry in our domain is mostly relying on a *process-based quality assurance* approach. The introduction of the DO-178C avionics norm and of its formal methods supplement DO-333 published by RTCA[1] shows a will to consolidate this process-based approach with a *product-based quality assurance* one.

Process-based quality assurance consists in a thorough inspection of the process leading to the creation of the product. If the process is deemed trustworthy, then so is the resulting product. For avionics hardware parts for example the design and production of components are closely monitored at the process level. They are then tested against their specification by sampling the global production rather than testing all of them.

Product-based quality assurance on the other hand aims at verifying, in a sound way, that the final product itself meets its specification. This especially applies to software, for which there exists methods amenable to automation to prove mathematically that the software itself respects its specification. These methods, established by the research community in computer science, are called *formal methods*. The interest of the approach is strengthened by the fact that software is made of zeros and ones which can be easily duplicated without any margin of error, which is not the case of hardware.

Avionics critical embedded systems are traditionally developed in a high-level language (model level) different from the final implementation language (code level). Formal methods are available for each phase of this development process:

- formal specification [71, 35] provides languages with mathematical semantics and al-

[1] http://www.rtca.org/

lows one to express what the system should do;

- verification of models against high-level specifications [16, 17, 23, 55, 12, 6] proves mathematically that the design of the system respects its specification;
- code generation from high-level models [13] produces code respecting the semantics of the high-level language, thus preserving the verification effort thanks to certified / proved compilers.

The work presented in this thesis focuses on the automatic verification of models against their specification. Indeed, while certification organisms acknowledge the existence of formal methods and advise to use them, there is a gap to bridge between their capabilities and the systems and specifications encountered in the industry. In practice verification of arbitrary functional properties on realistic systems often requires expert knowledge about the systems analyzed and the verification technique(s) used. The spread of formal verification in the industrial community is hindered by this need for costly and time-consuming expert intervention.

The goal of this thesis is to improve on the state of the art of formal verification in our application domain by *(i)* improving its automation on common design patterns found in avionics systems; *(ii)* finding proofs that can be quickly re-checked and are thus trustworthy. To achieve these goals we propose to study the collaboration of existing formal methods – such as SMT solving, k-induction, Quantifier Elimination, *etc.* – to propose a methodology for the automatic discovery of lemmas easing the process of concluding the proof. Also, this work focuses on parallel algorithms, consistently with the current trend of multi-core and many-core computers.

In this part of the thesis, we expose further the problem of formal verification of avionics critical systems at the model level in Chapter 4. We discuss languages used in model level conception in Section 4.1 and present common avionics design patterns in Section 4.2. Chapter 5 introduces basic formal notions to model the verification problems in Section 5.1 and presents building blocks to work on said verification problems, SAT and SMT solvers, in Section 5.2. Then, Chapter 6 formalizes the representation of critical systems thanks to transition systems in Section 6.1 and reviews the state of the art of transition system verification in Section 6.2 and Section 6.3. Part II presents our main contribution, a parallel collaborative proof engine architecture as well as invariant discovery methods to incorporate in this framework. Part III presents the most interesting aspects of our implementation of this proof engine, called Stuff.

CHAPTER 4

Problem

In this thesis, we focus on the analysis of embedded *reactive systems* [38]. These systems sample their environment using sensors, and compute an output used as a command for an actuator. Sampling is performed at regular intervals specified by the frequency of the system. We consider embedded reactive software functions which contribute to the safe operation of collections of hardware sensors, networked computers, actuators, moving surfaces, *etc.* called *functional chains*. A functional chain can for instance be in charge of "controlling the aircraft pitch angle", and must meet both qualitative and quantitative safety requirements depending on the effects of its failure. Effects are ranked from MIN (minor effect with no casualties) to CAT (catastrophic effect with casualties). For instance, the failure of a pitch control function is ranked CAT, and it shall be robust to at least a double failure and have an average failure rate of at most 10^{-9} per flight hour. In order to meet these requirements, engineers must introduce hardware and software redundancy and implement several fault detection and reconfiguration mechanisms in their software.

Software development is hard, even more so when developing real-time systems. To address this issue, programmers first design their programs in a language abstracting away real-time constraints. A code generator then produces low-level code targeted in real-time operating systems. We present design languages, and in particular the Lustre language, in Section 4.1. We then present our use case in Section 4.2, a typical avionics functional chain. We will focus in particular on the reconfiguration and voting subsystems.

4.1 Lustre: A Synchronous Programming Language

Embedded systems in general and functional chains in particular have to comply with real-time constraints such as release dates and deadlines on the computations, assuming a non-faulty hardware. Low-level Application Programming Interfaces (API) such as POSIX [37] are now widely used for development of small real-time systems. To develop complex functional chains however, using such a low-level approach is tedious and error-prone, and overall unrealistic. This is why during the eighties *synchronous languages* such as Lustre [36], Esterel [7] and Signal [5] were proposed. The idea behind the synchronous paradigm is to

provide designers with a language which *(i)* allows one to design real-time systems at a higher level of abstraction; *(ii)* is formal, *i.e.* the semantics of a program is not ambiguous. In the following we will refer to this level of abstraction as the *model level*, as opposed to *code level* which is the code generated from a synchronous language. At the model level, real-time constraints are abstracted to the notion of *steps* or *cycles*, and the following hypothesis is assumed.

Definition 4.1.1 (Synchronous Hypothesis) The *Synchronous Hypothesis* consists in viewing time as a succession of discrete instants at which computations take place instantly. Each of these instants defines a *global state*, yielding a *state machine* or *transition system* semantics.

System design the at model level has many benefits. The most immediate from an industrial point of view is that it shortens development processes by letting the code generator handle the difficult part of implementing the high-level design by generating the low-level code. It is thus not surprising that synchronous languages have been rapidly adopted by embedded system designers. Most notably, the Esterel Technologies company[1] provides them with a complete suite of tools, the SCADE Suite®, including the KCG code generator certified/qualified according to international rail transport, nuclear energy, automotive and avionics standards. The SCADE suite is centered around the SCADE language, which is similar to Lustre. Verification, debug, coverage analysis... techniques are applied at the model level, on SCADE programs. Verification at the model level is a rewarding approach: as the code generator is trustworthy, a proof at the model level ensures the code generated respects the specification.

Let us now give a brief overview of the Lustre language. It is the input language of the verification tool in which the techniques discussed in this thesis have been implemented. In addition to being a synchronous language, Lustre is a *dataflow* language: every variable or expression is an infinite flow, or *stream*, of values following the rhythm of a clock. Every Lustre program has a *base clock*, which represents the smallest, indivisible discrete steps for the program. All the other clocks are streams of Boolean values for each instant defined by the base clock and sample streams at a different, slower rate than the base clock. Consider for instance the following Lustre program defining a *node*, *i.e.* a set of equations. Counter `c1` counts the number of cycles of the base clock and `c2` counts the number of times `clock`, the Boolean input of the system, is `true`.

[1] http://www.esterel-technologies.com/

```
node counters(clock: bool) returns (c1,c2: int);
let
  c1  = 0 -> pre(c1)  + 1;
  c2  = 0 -> if clock then (pre(c2) + 1) else pre(c2);
tel
```

The equations linking the counters to the input are between the `let` and `tel` keywords. Output `c1` is initially zero, and at each step after that – operator `->` – is equal to its value in the preceding step plus one. Keyword `pre` is a *unit delay operator* defining a *memory* or *latch*. The memory contains the value of the expression given as argument at the previous step. So `pre(c1)` represents the value of stream `c1` at the previous step. Stream `c2` is also initially zero but must stay the same when `clock` is `false`, and be equal to its previous value plus one otherwise. Now, this node is observationally equivalent to the following one.

```
node counters(clock: bool) returns (c1,c2: int);
var
  c2C: int;
let
  c1  = 0 ->  pre(c1)  + 1;
  c2C = (0 -> (pre(c2C) + 1)) when (true -> clock);
  c2  = current(c2C);
tel
```

A local variable, `c2C`, has been introduced and follows a different clock than the base one. The `when` keyword indicates that `c2C` is *present* – its clock ticks and its value is defined – when `(true -> clock)` is `true`, *i.e.* in the initial state and whenever `clock` is true. When it is not present, `c2C` is *absent*: it has no value. An example of *trace* – sequence of legal values for the flows of the system – follows. With the exception of the line corresponding to `c2C`, it is also a trace for the first version of the `counters` node since they are equivalent.

base clock	true	true	true	true	true	true	true	...
pre(c1)	–	0	1	2	3	4	5	...
c1 = 0 -> pre(c1) + 1	0	1	2	3	4	5	6	...
clock	false	false	true	true	false	false	true	...
true -> clock	true	false	true	true	false	false	true	...
c2C	0	–	1	2	–	–	3	...
c2	0	0	1	2	2	2	2	...

Notice that Lustre is a declarative language: "=" denotes a constraint *via* an equation, and is not an assignment. Hence the body of the node is defined in a declarative manner by a set of equations. Now, consider the Lustre program on Figure 4.1. Node `middleValue` returns the middle value of its three inputs. It also outputs a Boolean flag `faulty` which

```
-- Returns the absolute value of its input.
node abs(n: int) returns (result: int);
let
  result = if (n < 0) then -n else n;
tel

-- "ok" becomes and stays false as soon as "in" escapes
-- the -"bound", +"bound" interval.
node bounded(in, bound: real) returns (ok: bool);
let
  assert(bound > 0);
  ok = true -> pre(ok) and (abs(in) < bound);
tel

-- Returns the middle value of "in1", "in2" and "in3", and
-- "faulty" which is true as soon as an input escapes its interval.
node middleValue(in1, in2, in3, bound: real) returns (middle:
   real; faulty: bool);
var
  c12, c13, c23, toProve: bool;
let
  assert(bound > 0);
  c12 = in1 > in2;
  c23 = in2 > in3;
  c31 = in3 > in1;
  middle = if (c12 = c23) then in2 else
           if (c23 = c31) then in3 else in1;
  faulty = not (bounded(in1,bound) and
                bounded(in2,bound) and
                bounded(in3,bound));
tel
```

Figure 4.1: A simple Lustre program.

indicates whether the inputs have **always** been in a specified interval or not, *i.e.* it becomes true as soon as an input escapes its bounds, and stays that way forever. This nodes use the `assert` keyword specifying a constraint on the flows.

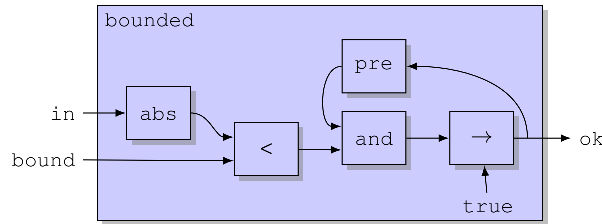


Figure 4.2: Network representation of the `bounded` node.

A Lustre program is a hierarchical network of nodes and operators connected by dataflow variables. Figure 4.2 for instance represents the `bounded` node in SCADE-like representation. Note the *instantiation* of the `abs` node, which means that an instance of the network corresponding to `abs` should be placed here. As another example, the `middleValue` node instantiates the `bounded` node three times, each with their different `pre` memories. The result is clear and unambiguous semantics for the Lustre programs.

Verification. The node `observer` on Figure 4.3 illustrates the usual approach to the verification of Lustre programs. Its inputs correspond to the inputs and outputs of the `middleValue` node. It produces the value `true` at steps for which the value of `result` is within a certain range, and at steps where `faulty` is `true`. Node `top` instantiates nodes of the system and returns the status of the observers – only one in this example. The verification challenge is to prove that the observers always produce the value `true`. If this is the case on our example, then in all possible executions, whenever an input escapes its legal boundaries the faulty flag is raised and stays that way. Such properties are called *safety* properties: they specify the correct behavior of the system and should hold in any state of the system.

They differ from *liveness* properties which express that "if <a certain condition> is true, then eventually it must be true that <another condition>". For instance, it could be the case that the `faulty` flag is not raised immediately when an input escapes its bound. Instead, the system could wait for the input to stay out of its bounds for n steps to confirm that the input is indeed faulty. The specification could then be the *bounded* liveness property "if an input stays out of bounds for n consecutive steps, then the `faulty` flag is raised". Bounded liveness properties can be seen as safety properties as illustrated on Figure 4.4 using a `counter` node to count the number of cycles the inputs have been out of bounds for. This is not the case for *unbounded* liveness properties, such as "if an input stays out of bounds (for an arbitrary long time), then eventually the `faulty` flag will be raised", unless the system has a finite number of states [8], or for some special classes of infinite state systems [68].

```
node observer(in1, in2, in3, bound, result: real; faulty: bool)
   returns(
  toProve: bool
);
let
  assume(bound > 0);
  -- As long as "faulty" is false, "result" is bounded by "bound".
  toProve = bounded(result,bound) or faulty;
tel

node top(in1, in2, in3, bound: real) returns (ok: bool)
var
  result: real;
  faulty: bool;
let
  assume(bound > 0);
  (result,faulty) = middleValue(in1,in2,in3,bound);
  ok = observer(in1,in2,in3,bound,result,faulty);
tel
```

Figure 4.3: The observer pattern.

The last category of properties are those of *fairness*, which express the fact that if something is *always* possible then it must be done infinitely often. These properties are typically used to specify that when some concurrent processes run in parallel and try to access a resource concurrently, then if a process can *always* gain access to it then at some point it will. In the following however we will only consider safety properties.

As we shall see in the presentation of our use case, industrial critical embedded systems are very complex: this results in what is known as the *combinatorial explosion* of the state space. The *reachable state space*, the set of all states the system can reach from its initial states, is so huge that it cannot be represented, or is simply infinite. Chapter 6 discusses a compact representation to conduct formal analyses on.

```
node counter(condition: bool; max: int) returns (maxed: bool);
var cpt: int;
let
  assume(max > 0);
  cpt = 0 -> if (condition) pre(cpt)+1 else 0;
  maxed = cpt > max;
tel

node observer(
  in1, in2, in3, bound, result: real; faulty: bool
) returns(ok: bool)
let
  assume(bound > 0);
  problemConfirmed = counter(not bounded(result,bound),42);
  ok = (not problemConfirmed) or faulty;
tel
```

Figure 4.4: Bounded liveness property as a safety property.

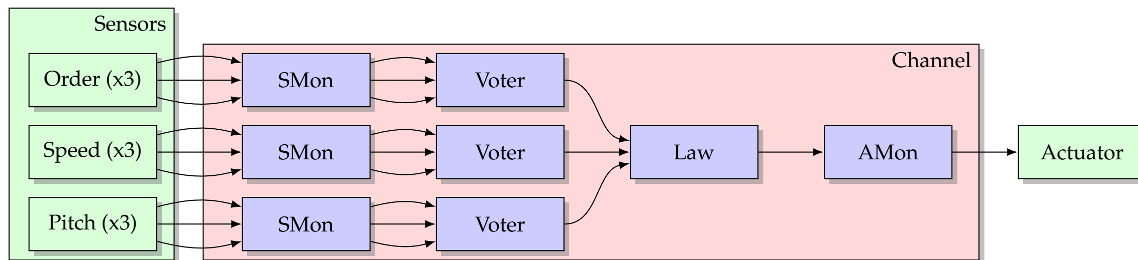


Figure 4.5: A Single Computation Channel.

4.2 Safety Critical Avionics Systems Verification

Figure 4.5 presents a computation channel corresponding to the function "controlling the aircraft pitch angle" as usually found in avionics functional chains. Note that the sensors are triplicated to tolerate sensor faults and hence reduce the failure rate of the sampling. At each cycle, the channel samples the Speed, Pitch, and Order[2] sensors. All the values sampled go through sensor monitors (SMon) in charge of detecting temporary or permanent failures such as out-of-bounds values. The signals then enter a triplex voter (Voter); the voter

[2]Order is the command from the pilot.

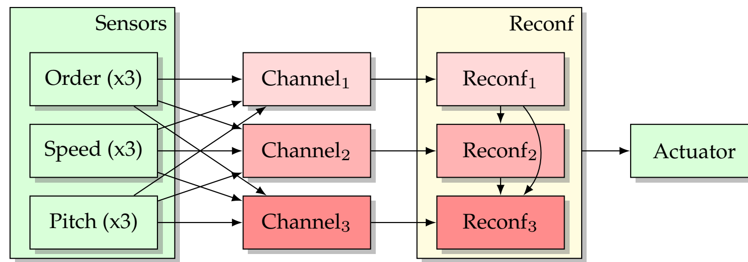


Figure 4.6: Triple Channel Functional Chain.

performs additional failure detection and outputs one single signal for, say, speed based on the three speed values sampled – the simple Lustre program from Figure 4.1 can be seen as a toy example of voting logic. The next block is the control law (Law) which computes the signal to send to the actuator using a single input value for order, speed and pitch. The signal also goes through an actuator monitor (AMon) which makes sure that the command is indeed enforced by the actuator. If it is not the case, a failure flag is raised.

In order to further reduce the failure rate of this critical function of the aircraft, computation channels are triplicated (Figure 4.6): the computations are performed by three redundant channels running on separate hardware, each using distinct communication wires. To decide which channel actually controls the actuator, a *reconfiguration mechanism* (Reconf) is distributed over the three channels. It is responsible of the safety aspect of the triplication, *reconfiguring* the system in case a channel is considered corrupted because it either consumes or produces invalid data. In fact, whatever the output of each channel is, the reconfiguration mechanism should ensure that no command dangerous for the actuator or the aircraft is output by the triplicated architecture, and that proper flags are raised to inform the pilot whenever a failure is detected.

Note that an incorrect output should not immediately mark a channel as corrupted. It could be that the disturbance is temporary, *e.g.* because of a thunderbolt nearby causing an electromagnetic perturbation. So, only if the output is incoherent for more than n cycles will the failure be considered permanent. To detect incoherent outputs and be able to produce a correct command while a timer counts the n cycles, functional chain designers can use a technique called *shuffle*. It typically takes place after the control law computation, as illustrated on Figure 4.7: each channel has access to the control law output computed by the other channels. The actual output of a channel is the result of a vote between those three values. This allows a channel to detect that it is incoherent on its own and count the n cycles while still producing a correct output. There is also a timer used before switching between a corrupted channel and a safe one; it can be the case that because of a temporary perturbation the channel in control stops sending a command for a short time. Also, channels

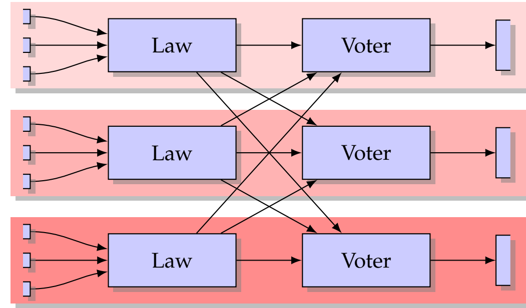


Figure 4.7: The shuffle mechanism.

usually need to perform several checks before taking control to assess precisely the state of the actuator.

As mentioned above, each channel runs on separate hardware, but it can be the case that the different parts of one channel also run on different hardware components. In that case functional chain designers can go further and allow the system to deactivate only part of a channel. For instance, assume channel 1 has control but realizes that the voter after the control law does not produce any output anymore because of a hardware failure. If the control law of channel 1 – running on different hardware than the voter – is still operative then it could be decided that control is transferred to channel 2, but that the control law of channel 1 keeps computing and participates in the voting now taking place on channel 2.

Functional chains are extremely complex, and so is their specification [67]. Verification of the software of a full functional chain is thus a challenge for current techniques and tools. Even with a modular approach, individual verification problems are far from trivial. The objective of this thesis is to provide generic techniques for the verification of systems such as voting and reconfiguration logic implementations. These systems make extensive use of linear integer arithmetic because of the many timers they rely on for fault confirmation and transition between faulty channels, as well as linear real arithmetic when handling a signal. The voting logic for instance has to maintain a certain continuity in its output: the control law expects values consistent with the real world and unexpected behavior could arise, were its inputs to show discontinuities because of sensor failures. In particular, we will focus on the Rockwell Collins triplex voter system and an implementation of reconfiguration logic, the code of which is available in Appendix A. Control law systems differ from voting and reconfiguration in that they are based on differential equations. They can be non-linear or require a non-linear analysis and are outside the scope of this study.

CHAPTER 5

Mathematical Background and Notation

This chapter gives notation and notions in order to define the problem of Boolean satisfiability (SAT) and Satisfiability Modulo Theory (SMT). We will use SMT solvers extensively in this thesis. We begin by defining many-sorted first-order logic in Section 5.1. We then present the modern approach to solving SAT and SMT problems in Section 5.2. Last, we discuss the SMT-LIB 2 standard and additional features encountered in most state of the art solvers in Section 5.3.

5.1 Many-Sorted First-Order Logic

This section formalizes the concepts required to define *many-sorted first-order logic*. The *syntax* of the logic allows the construction of sequences of symbols and is defined in Subsection 5.1.1 with the notions of many-sorted first-order signature and formula. In Subsection 5.1.2, *semantics* is given by structures over a many-sorted signature. Semantics gives meaning to the sequences of symbols allowed by the syntax. We then formally introducing logics in Subsection 5.1.3 before briefly discuss proof systems in Subsection 5.1.4.

5.1.1 Syntax

Definition 5.1.1 (Many-sorted signature) A *many-sorted first-order signature* Σ is a tuple $\langle$ $\mathsf{Sorts}, \mathsf{Var}, \mathsf{Fun}, \mathit{Rank}$ $\rangle$ consisting of:

- a set of *sorts* Sorts containing at least the sort `Bool`;
- a countably infinite set of *variable symbols* $\mathsf{Var} \triangleq \{v_1, v_2, \dots\}$, usually implicit when defining an actual signature;
- a set of *function symbols* Fun such that $\mathsf{Fun} \cap \mathsf{Var}$ is empty and Fun contains $\wedge$, $\vee$, $\neg$, $\bot$, $\top$, and $=$;
- a left-total relation Rank on $\mathsf{Fun} \times \mathsf{Sorts}^+$.

Each sort sequence associated to f by Σ is a *rank* of f.

Intuitively, the sorts of a signature are types used by the function symbols to specify their ranks. Note that overloading is supported: a function can have multiple ranks. We will rely heavily on the following notation to discuss function symbols ranks. Given a signature Σ, we write $f(\sigma_1, \ldots, \sigma_n) : \sigma$ for $(f, \sigma_1 \ \ldots \ \sigma_n \ \sigma) \in Rank$ and $f : \sigma$ if $n = 0$, in place of $f() : \sigma$.

Core **signature.** Core is the many-sorted signature $\langle \{\texttt{Bool}\}, \emptyset, \mathsf{Fun}_{core}, Rank_{core} \rangle$ composed of the following function symbols:

$\vee$	$(\texttt{Bool}, \texttt{Bool}):$	$\texttt{Bool}$	$\wedge$	$(\texttt{Bool}, \texttt{Bool}):$	$\texttt{Bool}$	$\neg$	$(\texttt{Bool}):$	$\texttt{Bool}$
$\top$	$():$	$\texttt{Bool}$	$\bot$	$():$	$\texttt{Bool}$	$=$	$(\texttt{Bool}, \texttt{Bool}):$	$\texttt{Bool}$

Definition 5.1.2 (Legal many-sorted signatures) A *legal many-sorted signature* $\langle \mathsf{Sorts}, \mathsf{Var}, \mathsf{Fun}, Rank \rangle$ is a many-sorted signature such that $\texttt{Bool} \in \mathsf{Sorts}$, $\mathsf{Fun} \in \mathsf{Fun}_{core}$, $Rank \in Rank_{core}$, and for all $\sigma \in \mathsf{Sorts}$: $= (\sigma, \sigma) : \texttt{Bool} \in Rank$. In the following we consider only legal signatures. For the sake of readability we will omit the word "legal".

Definition 5.1.3 (Abstract syntax) Given a many-sorted signature Σ, the *abstract syntax* of Σ-*terms* is given by the following Extended Backus-Naur Form (EBNF [53]):

$$T ::= v \mid f\ T^* \mid f^\sigma\ T^* \mid \exists (x:\sigma)^+\ T \mid \forall (x:\sigma)^+\ T \mid \textbf{let}\ (x = T)^+\ \textbf{in}\ T \mid (T)$$

where $f \in \mathsf{Fun}$, $\sigma \in \mathsf{Sorts}$ and $v \in \mathsf{Var}$. $f\ T^*$ is called a *function application*, and so is $f^\sigma\ T^*$. **let** $\ldots$ **in** $\ldots$ is called a *(parallel) let-binding*. Symbols $\forall$ and $\exists$ are called *quantifiers*.

Function application f^σ distinguishes between applications of function symbols with more than one rank. This is made explicit in the rule *(sfun)* in the following rules of well-sortedness.

Definition 5.1.4 (Rules of well-sortedness) Given a many-sorted signature Σ, an environment $\mathcal{E}$ is a set of statements $v : \sigma$, read as *"v has sort σ"*, where $v \in \mathsf{Var}$ and $\sigma \in \mathsf{Sorts}$. We say that *"term t is well-sorted of sort σ in $\mathcal{E}$"*, written $\mathcal{E} \vdash t : \sigma$, if t is derivable by the following sorting rules:

$$\frac{}{\mathcal{E} \vdash v : \sigma}\ \textbf{(var)} \quad \text{if } v : \sigma \in \mathcal{E}$$

$$\frac{\mathcal{E} \vdash t_1 : \sigma_1 \quad \ldots \quad \mathcal{E} \vdash t_n : \sigma_n}{\mathcal{E} \vdash (f\ t_1\ \ldots\ t_n) : \sigma}\ \textbf{(fun)} \quad \text{if} \begin{cases} f(\sigma_1, \ldots, \sigma_n) : \sigma \in \Sigma & \text{and} \\ f(\sigma_1, \ldots, \sigma_n) : \sigma' \notin \Sigma & \text{for all } \sigma' \neq \sigma; \end{cases}$$

$$\frac{\mathcal{E} \vdash t_1 : \sigma_1 \quad \ldots \quad \mathcal{E} \vdash t_n : \sigma_n}{\mathcal{E} \vdash (f^\sigma\ t_1\ \ldots\ t_n) : \sigma}\ \textbf{(sfun)} \quad \text{if} \begin{cases} f(\sigma_1, \ldots, \sigma_n) : \sigma \in \Sigma & \text{and} \\ f(\sigma_1, \ldots, \sigma_n) : \sigma' \in \Sigma & \text{for some } \sigma' \neq \sigma; \end{cases}$$

$$\frac{\mathcal{E} \cup \{v_1 : \sigma_1,\ \ldots,\ v_n : \sigma_n\} \vdash t : \texttt{Bool}}{\mathcal{E} \vdash (Q\ v_1 : \sigma_1\ \ldots\ v_n : \sigma_n\ t) :\ \texttt{Bool}}\ \textbf{(Q)} \quad \text{if } Q \in \{\exists, \forall\} \text{ and } n > 0;$$

$$\frac{\mathcal{E} \vdash t_1 : \sigma_1 \quad \ldots \quad \mathcal{E} \vdash t_n : \sigma_n \quad \mathcal{E} \cup \{v_1 : \sigma_1,\ \ldots,\ v_n : \sigma_n\} \vdash t : \sigma}{\mathcal{E} \vdash (\textbf{let}\ v_1 = t_1\ \ldots\ v_n = t_n\ \textbf{in}\ t) : \sigma}\ \textbf{(let)} \quad \text{if } n > 0.$$

We say that a term t is *well-sorted* of sort σ if it is well-sorted of sort σ in $\{\}$, the empty environment. A well-sorted term is *quantifier free* if no quantifier appears in it – the (Q) rule is never used in its well-sortedness derivation. A well-sorted *formula* in $\mathcal{E}$ is a well-sorted term of sort `Bool` in $\mathcal{E}$.

Definition 5.1.5 An *atom* is a formula in which the logical connectives $\wedge$ (conjunction), $\vee$ (disjunction) and $\neg$ (negation) do not appear. A *literal* is a or $\neg(a)$ where a is an atom. A *cube* (resp. a *clause*) is a conjunction (resp. a disjunction) of literals. A formula is in Disjunctive Normal Form (DNF) if it is a disjunction of cubes. A formula is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses.

Additional notation. It is possible to define a concrete syntax using *Polish prefix notation*: an application of $f(\sigma_1, \ldots, \sigma_n) : \sigma$ is of the form $(f\ t_1\ \ldots\ t_n)$. While this notation is ideal for machines as it is easy to parse and print, it is usually painful to process for humans who are more accustomed to *infix notation* for applications of common function symbols, *e.g.* $(n + 1 > 3)\ \wedge b$ instead of $(\wedge\ (>\ \ (+\ n\ 1)\ 3)\ b)$. In this thesis we will manipulate terms extensively. We thus lighten their representation: application of $f(\sigma_1, \ldots, \sigma_n) : \sigma$ will usually be written $f(t_1,\ \ldots\ , t_n)$ instead of $f\ \ t_1\ \ \ldots\ \ t_n$ with the following alternatives: $(t_1\ f\ t_2)$ if $n = 2$, $(f\ t_1)$ if $n = 1$, or simply f if $n = 0$. Also, we allow ourselves to write

$$Q(v_1 : \sigma_1,\ \ldots,\ v_n : \sigma_n)\ t \quad \text{for} \quad Q\ v_1 : \sigma_1\ \ldots\ v_n : \sigma_n\ t$$

where $Q \in \{\forall, \exists\}$, and

$$\textbf{let}\ (v_1 = t_1, \ldots, v_n = t_n)\ \textbf{in}\ t \quad \text{for} \quad \textbf{let}\ v_1 = t_1\ \ldots\ v_n = t_n\ \textbf{in}\ t.$$

The following operator precedence rules apply, where Q is a quantifier: $= \ \triangleright \neg \triangleright \wedge \triangleright \vee \triangleright Q(\ldots)$. where $\triangleright$ means "has precedence over". Also, in a signature with function symbols $*, -, +, \geq, \leq, >, <$ the following precedence convention will be followed:

$$* \ \triangleright\ -\ \triangleright\ +\ \triangleright\ \{\geq,\ \leq,\ >,\ <\}\ \triangleright\ =$$

For example, $n + 1 > 3\ \wedge\ b$ is the same as $((n+1) > 3)\ \wedge\ b$ and as $\wedge(> (+(n, 1), 3), b)$. Last, we will often use the shorthand $\Rightarrow$ defined as $\phi \Rightarrow \psi \stackrel{\Delta}{=} \neg\phi \vee \psi$ where ϕ and ψ are formulas. Note that "$\stackrel{\Delta}{=}$" represents *meta-equality* and should be read as "is defined as". It differs from "$=$" which is just a function symbol in the signature.

We write **if** t_1 **then** t_2 **else** t_3 where $t_1, t_2, t_3 \in \texttt{Bool}$, for the traditional conditional construction $(t_1 \wedge t_2) \ \vee \ (\neg t_1 \wedge t_3)$. We allow ourselves to write

$$\bigwedge_{i \in [1,n]} t_i, \quad \text{resp.} \quad \bigvee_{i \in [1,n]} t_i$$

for $t_1 \wedge \dots \wedge t_n$, resp. $t_1 \vee \dots \vee t_n$. Let us now illustrate the notions introduced so far with this lighter syntax in the following examples.

Example 5.1.1 (Int signature) Int *is the many-sorted signature with*

- Sorts $\triangleq \{\texttt{Bool}, \texttt{Int}\}$;
- Fun *is composed of the symbols in* $\{0, 1, \dots\}$ *with rank* $\texttt{Int}$*, the function symbols from the* Core *signature,* $= (\texttt{Int}, \texttt{Int}) : \texttt{Bool}$*, and*

$+$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$	$-$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$	$-$	$(\texttt{Int}):$	$\texttt{Int}$
$*$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$	mod	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$	div	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$
abs	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Int}$	$<$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Bool}$	$>$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Bool}$
$\leq$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Bool}$				$\geq$	$(\texttt{Int}, \texttt{Int}):$	$\texttt{Bool}$

Definition 5.1.6 (Signature expansion) Let $\Sigma \triangleq \langle \mathsf{Sorts}, \mathsf{Var}, \mathsf{Fun}, \mathit{Rank} \rangle$ be a many-sorted signature and S a set of pairs $(f, \sigma_1 \ \dots \ \sigma_n \ \sigma)$ with sorts $\sigma_1, \dots, \sigma_n, \sigma$ in Sorts and function symbol f not necessarily in Fun. If F denotes the set of function symbols in S, we write $\Sigma(S)$ for $\langle \mathsf{Sorts}, \mathsf{Var} \setminus F, \mathsf{Fun} \cup F, \mathit{Rank} \cup S \rangle$. $\Sigma(S)$ is a signature. Elements of S can also be written $f(\sigma_1, \dots, \sigma_n) : \sigma$ or, if $n = 0$, $f : \sigma$.

Example 5.1.2 (Well-sorted terms) *In* $\mathsf{Int}(\{n_1 : \texttt{Int}, n_2 : \texttt{Int}, b_1 : \texttt{Bool}, b_2 : \texttt{Bool}\})$,

- *terms "*$5 - b_1$*" and "*$+(b_2, 11, n_1)$*" are not well-sorted in any environment;*
- *"*$v_1 - (5 + n_1)$*" is a well-sorted term of sort* $\texttt{Int}$ *in* $\{v_1 : \texttt{Int}\}$*:*

$$\dfrac{\dfrac{}{\{v_1 : \texttt{Int}\} \vdash v_1 : \texttt{Int}}\,(var) \qquad \dfrac{\dfrac{}{\{v_1 : \texttt{Int}\} \vdash 5 : \texttt{Int}}\,(fun) \quad \dfrac{}{\{v_1 : \texttt{Int}\} \vdash n_1 : \texttt{Int}}\,(fun)}{\{v_1 : \texttt{Int}\} \vdash 5 + n_1 : \texttt{Int}}\,(fun)}{\{v_1 : \texttt{Int}\} \vdash v_1 - (5 + n_1) : \texttt{Int}}\,(fun)$$

- *"*n_2*", "*$n_1 + 42$*" and "*$n_1 \geq 3$*" are well-sorted terms of sort* $\texttt{Int}$*;*
- *"*$b_1 \wedge n_1 \geq 3$*", "*$\textbf{let } v_1 = 1 + 2 \textbf{ in } (n > v_1)$*" and "*$\forall v_1 : \texttt{Int},\ (v_1 \geq n_2 \ \vee \ \exists v_2 : \texttt{Bool},\ v_2 \wedge \neg v_2)$*" are well-sorted formulas. The latter for example is derived as follows:*

$$
\dfrac{
  \dfrac{
    \dfrac{}{\{v_1 : \texttt{Int}\} \vdash v_1 : \texttt{Int}}\,(var)
    \quad
    \dfrac{}{\{v_1 : \texttt{Int}\} \vdash n_2 : \texttt{Int}}\,(fun)
  }{\{v_1 : \texttt{Int}\} \vdash (v_1 \geq n_2) :\ \texttt{Bool}}\,(fun)
  \quad
  \dfrac{
    \dfrac{
      \dfrac{}{\{v_1 : \texttt{Int}, v_2 : \texttt{Bool}\} \vdash v_2 :\ \texttt{Bool}}\,(var)
    }{\{v_1 : \texttt{Int}, v_2 : \texttt{Bool}\} \vdash \neg(v_2) :\ \texttt{Bool}}\,(fun)
  }{\{v_1 : \texttt{Int}\} \vdash \exists v_2 : \texttt{Bool},\ \neg(v_2) :\ \texttt{Bool}}\,(Q)
}{
  \dfrac{\{v_1 : \texttt{Int}\} \vdash ((v_1 \geq n_2)\ \vee\ \exists v_2 : \texttt{Bool},\ \neg(v_2)) :\ \texttt{Bool}}{\{\} \vdash \forall v_1 : \texttt{Int},\ (v_1 \geq n_2\ \vee\ \exists v_2 : \texttt{Bool},\ \neg(v_2)) :\ \texttt{Bool}}\,(Q)
}\,(fun)
$$

- *"b_1", "$\textbf{let } v_1 = 1 + 2 \textbf{ in } (n > v_1)$" and "$n_1 \leq 42$" are atoms. They are also literals, and so are "$\neg b_1$", "$\neg(\textbf{let } v_1 = 1 + 2 \textbf{ in } (n > v_1))$" and "$\neg(n_1 \leq 42)$"*

- *"$v_2 \wedge \neg b_2$" is well-sorted in $\{v_2 : \texttt{Bool}\}$ but is not well-sorted in $\{\}$.*

Definition 5.1.7 The set of *free variables* of a term t, or $FV(t)$, is defined recursively as follows:

- $FV(v) \triangleq \{v\}$ if $v \in \mathsf{Var}$; $\quad FV(c) \triangleq \{\}$ if $c \in \mathsf{Const}$;

- $FV(f(t_1, \ldots, t_n)) \triangleq FV(f^\sigma(t_1, \ldots, t_n)) \triangleq FV(t_1) \cup \cdots \cup FV(t_n)$ for $f \in \mathsf{Fun} \setminus \mathsf{Const}$,

- $FV(Q(v_1 : \sigma_1, \ldots, v_n : \sigma_n)\ t_0) \triangleq FV(t_0) \setminus \{v_1, \ldots, v_n\}$ for $Q \in \{\exists, \forall\}$, and

- $FV(\textbf{let } (v_1 = t_1\ \ldots\ v_n = t_n) \textbf{ in } t_0)$ is

$$(FV(t_0) \setminus \{v_1, \ldots, v_n\})\ \cup\ \{v \mid v_i \in FV(t_0) \text{ and } v \in FV(t_i) \text{ for } 1 \leq i \leq n\}.$$

Property 5.1.1 Given a signature Σ and a term t: if t is well-sorted then it has no free variables.

PROOF (SUGGESTED) Direct consequence of the following lemma:
If t is a well-sorted term in an environment $\{v_1 : \sigma_1, \ldots, v_n : \sigma_n\}$, then $FV(t) \subseteq \{v_1, \ldots, v_n\}$.■

Remark. Following the SMT-LIB 2 standard[1], we will only consider well-sorted terms (in an empty environment). A perhaps more traditional approach is to assign a sort to each variable and allow well-sorted terms to have free variables. However, once semantics are given in the next section, we will see that this definition of well-sortedness is without loss of generality.

[1] Discussed in Section 5.3.

5.1.2 Semantics

Model theory is the study of the semantics of a signature, and of the formulas built on it. Well-sorted formulas and terms alone do not make sense, they are simply trees of symbols. The symbols and the sorts need to be interpreted in some way. As we shall see in the next definition, the symbols appearing in signature Core – *i.e.* the mandatory symbols of a signature – have a unique interpretation corresponding to Boolean logic. Symbols such as "+" or "`Int`" in Int however have no restriction on their interpretation besides their ranks.

A structure gives semantics to the symbols of a signature. Sort symbols are interpreted as sets of actual values, and function symbols are interpreted as functions over these sets with respect to their rank.

Definition 5.1.8 (Structure) Given a many-sorted signature $\Sigma \triangleq \langle\ \mathsf{Sorts}, \mathsf{Var}, \mathsf{Fun}, \mathit{Rank}\ \rangle$, a *structure* M for Σ is a pair $\langle \mathcal{C}, \mathcal{I} \rangle$ such that

- the *carrier function* $\mathcal{C}$ assigns a non-empty domain $\mathcal{C}(\sigma)$ to every sort σ in Sorts, called the *carrier* of σ, with $\mathcal{C}(\texttt{Bool}) \triangleq \{\mathbf{T}, \mathbf{F}\}$, where $\mathbf{T}$ and $\mathbf{F}$ are the two Boolean truth values for *true* and *false* respectively; we note $\mathcal{C}(\mathsf{Sorts})$ the union of all the carriers;
- the *interpretation function* $\mathcal{I}$ is defined on Fun as follows:
  - each constant $c : \sigma$ from Fun is interpreted as $\mathcal{I}(c) \in \mathcal{C}(\sigma)$, with $\mathcal{I}(\top) \triangleq \mathbf{T}$ and $\mathcal{I}(\bot) \triangleq \mathbf{F}$, and
  - each symbol $f(\sigma_1, \ldots, \sigma_n) : \sigma$ with $n \neq 0$ from Fun is interpreted as a function

$$\mathcal{I}(f) : \mathcal{C}(\sigma_1) \times \cdots \times \mathcal{C}(\sigma_n) \to \mathcal{C}(\sigma)$$

    with symbols $\neg$, $\vee$, $\wedge$ and $=$ interpreted follows:
    * $\mathcal{I}(\neg)(p)$ is $\mathbf{T}$ if and only if p is $\mathbf{F}$;
    * $\mathcal{I}(\vee)(p, q)$ is $\mathbf{T}$ if and only if p or q is $\mathbf{T}$;
    * $\mathcal{I}(\wedge)(p, q)$ is $\mathbf{T}$ if and only if p and q is $\mathbf{T}$;
    * $\mathcal{I}(=)(l, r)$ is $\mathbf{T}$ if and only if l is the same as r.

A structure $\langle \mathcal{C}, \mathcal{I} \rangle$ is *partial* if $\mathcal{I}$ is a structure defined on a subset of Fun.

Definition 5.1.9 (Assignment) Given a structure $\langle \mathcal{C}, \mathcal{I} \rangle$ on Σ, an assignment A on an environment $\mathcal{E} \triangleq \{v_1 : \sigma_1, \ldots, v_n : \sigma_n\}$ is a function from $\{v_1, \ldots, v_n\}$ to $\mathcal{C}(\mathsf{Sorts})$. It maps $v_1, \ldots, v_n$ to $val_1 \in \mathcal{C}(\sigma_1), \ldots, val_n \in \mathcal{C}(\sigma_n)$ respectively.
Also if A is an assignment then $\mathsf{A}' \triangleq \mathsf{A}[v_0 \mapsto val_0]$ is defined by $\mathsf{A}'(v) \triangleq val_0$ if $v = v_0$ and $\mathsf{A}'(v) \triangleq \mathsf{A}(v)$ otherwise.

Definition 5.1.10 (Interpretation of terms) Given a many-sorted signature Σ, a structure $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$ and a well-sorted term t in $\mathcal{E}$, the interpretation of t by M and an assignment A on $\mathcal{E}$, written $[\![t]\!]^{\mathcal{I}}_{\mathsf{A}}$, is defined inductively on the structure of t as follows:

- $[\![v]\!]^{\mathcal{I}}_{\mathsf{A}} \triangleq \mathsf{A}(v)$ for $v \in \mathsf{Var}$;
- $\mathcal{I}(c)$ for $c \in \mathsf{Const}$;
- $\mathcal{I}(f)([\![t_1]\!]^{\mathcal{I}}_{\mathsf{A}}, \ldots, [\![t_n]\!]^{\mathcal{I}}_{\mathsf{A}})$ for $f \in \mathsf{Fun} \setminus \mathsf{Const}$;
- $[\![\exists(v_1 : \sigma_1,\ \ldots,\ v_n : \sigma_n)\ \phi]\!]^{\mathcal{I}}_{\mathsf{A}}$ is $\mathbf{T}$ if and only if for some $val_1 \in \mathcal{C}(\sigma_1), \ldots, val_n \in \mathcal{C}(\sigma_n)$,
$$[\![\phi]\!]^{\mathcal{I}}_{\mathsf{A}[v_1 \mapsto val_1]\ldots[v_n \mapsto val_n]} \text{ is } \mathbf{T}$$
- $[\![\forall(v_1 : \sigma_1,\ \ldots,\ v_n : \sigma_n)\ \phi]\!]^{\mathcal{I}}_{\mathsf{A}}$ is $\mathbf{T}$ if and only if for all $val_1 \in \mathcal{C}(\sigma_1), \ldots, val_n \in \mathcal{C}(\sigma_n)$,
$$[\![\phi]\!]^{\mathcal{I}}_{\mathsf{A}[v_1 \mapsto val_1]\ldots[v_n \mapsto val_n]} \text{ is } \mathbf{T}$$
- $[\![\mathbf{let}\ (v_1 = t_1, \ldots, v_n = t_n)\ \mathbf{in}\ t_0]\!]^{\mathcal{I}}_{\mathsf{A}} \triangleq [\![t_0]\!]^{\mathcal{I}}_{\mathsf{A}[v_1 \mapsto [\![t_1]\!]^{\mathcal{I}}_{\mathsf{A}}]\ldots[v_n \mapsto [\![t_n]\!]^{\mathcal{I}}_{\mathsf{A}}]}$.

Property 5.1.2 Given a many-sorted signature Σ, a structure $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$, and a well-sorted term t (in an empty environment), the interpretation of t depends solely on M. To lighten notation we write $\mathcal{I}(t)$ for $[\![t]\!]^{\mathcal{I}}_{\{\}}$ if t is a well-sorted term.

Definition 5.1.11 (Models) Given a many-sorted signature Σ and a well-sorted formula ϕ, a potentially partial structure $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$ is a *model* of ϕ if $\mathcal{I}(\phi)$ is $\mathbf{T}$, written $M \models \phi$. Note that M is not a model of ϕ if and only if $\mathcal{I}(\phi)$ is $\mathbf{F}$, which is the same as $M \models \neg\phi$. We will oftentimes write directly $M \models \neg\phi$ for "M is not a model of ϕ".

Definition 5.1.12 (Satisfiability) Given a many-sorted signature Σ, a formula ϕ is *satisfiable*, or *sat*, if there exists a structure M such that $M \models \phi$. Formula ϕ is *unsatisfiable*, or *unsat*, if it is not satisfiable.

Definition 5.1.13 (Validity) Given a many-sorted signature Σ, a formula ϕ is *valid*, or is a *tautology*, if any structure M is a model of ϕ, written $\models \phi$.

Property 5.1.3 Given a many-sorted signature Σ, a well-sorted formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable.

PROOF ($\Leftrightarrow$) Assume ϕ is valid. Then by definition for all structures $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$, $\mathcal{I}(\phi)$ is $\mathbf{T}$. Equivalently, for all structures $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$, $\mathcal{I}(\neg\phi)$ is $\mathbf{F}$, *i.e.* $\neg\phi$ is unsatisfiable. ∎

Definition 5.1.14 (Equisatisfiability, consequence and equivalence) Let Σ be a many-sorted signature and ϕ and ψ be two well-sorted formulas. We say that

- ϕ and ψ are *equisatisfiable* if ϕ is satisfiable if and only if ψ is satisfiable;
- ψ is a *(semantic) consequence* of ϕ, written $\phi \models \psi$, if any model of ϕ is a model of ψ;
- ψ is *(semantically) equivalent* to ϕ, written $\phi \equiv \psi$, if $\phi \models \psi$ and $\psi \models \phi$.

Extra constants. It is often useful to introduce constant symbols which we are sure are not used anywhere else. Given a signature Σ, we say that we introduce *extra constants* $x_1 : \sigma_1$, ..., $x_n : \sigma_n$ when we consider $\Sigma' \triangleq \Sigma(\{x_1 : \sigma_1, \dots, x_n : \sigma_n\})$ instead of Σ, where symbols $x_1, \dots, x_n$ do not appear in Σ and $\sigma_1, \dots, \sigma_n$ are sorts of Σ. Note that any formula ϕ in Σ is still a formula in Σ', and any model of ϕ in Σ is also a model of ϕ in Σ'. Conversely, from any model of ϕ in Σ' a model of ϕ in Σ can be extracted by removing the interpretations of $x_1, \dots, x_n$, if any.

Satisfiability, and therefore validity, are defined on well-sorted formulas which forbid free variables. This is without loss of generality. Indeed, given a many-sorted signature Σ, consider a formula ϕ well-formed in an environment $\{v_1 : \sigma_1, \dots, v_n : \sigma_n\}$ ($FV(\phi) \subseteq \{v_1, \dots, v_n\}$). While we could define a notion of *free-satisfiability* as the existence of a structure $M \triangleq \langle \mathcal{C}, \mathcal{I} \rangle$ and an assignment A such that $[\![\phi]\!]^{\mathcal{I}}_{\mathsf{A}}$ is $\mathbf{T}$, we instead introduce extra constants $x_1 : \sigma_1$, ..., $x_n : \sigma_n$ and consider the formula

$$\psi \triangleq \mathbf{let}\ (v_1 = x_1, \dots, v_n = x_n)\ \mathbf{in}\ \phi$$

which is satisfiable if and only if ϕ is free-satisfiable in Σ. An alternative approach is to consider $\psi' \triangleq \exists (v_1 : \sigma_1, \dots, v_n : \sigma_n)\ \phi$ which is equisatisfiable with ϕ in Σ. However, the free variables of ϕ are completely hidden in the sense that a model of ψ' would not give access to values of v_1, ..., v_n directly. By replacing free variables with extra constants however, it is easy to retrieve their values as in the case of free-satisfiability.

Remark. When defining an actual interpretation function $\mathcal{I}$, we will usually not recall the interpretation of $\top$, $\bot$, $\neg$, $\vee$, $\wedge$ and $=$ as they are the same in all structures. The same goes for the carrier of `Bool`. This is the case in the following examples.

Example 5.1.3 (Satisfiability in Core) *In* $\mathsf{Core}(\{a : \mathtt{Bool}, b : \mathtt{Bool}, c : \mathtt{Bool}\})$*, formula* $\phi_1 \triangleq a \vee (b \wedge \neg a)$ *is satisfiable since*

$$\langle \{\}, \{a \mapsto \mathbf{F},\ b \mapsto \mathbf{T},\ c \mapsto \mathbf{F}\} \rangle \models \phi_1.$$

Additionally,

$$\langle \{\}, \{a \mapsto \mathbf{F},\ b \mapsto \mathbf{T},\ c \mapsto \mathbf{T}\}\rangle \models \phi_1.$$

$\phi_2 \triangleq a \wedge \neg a \wedge c$ *on the other hand is unsatisfiable: no interpretation (of* a*) can build a model of* ϕ_2*. Hence* $\neg\phi_2 = \neg(a \wedge \neg a \wedge c) \equiv \neg a \vee a \vee \neg c$ *is valid.*

It is important at this point to note that the notion of structure is not restricted in any way – other than rank consistency – when interpreting non-logical symbols on signatures such as Int.

Example 5.1.4 (Structures on Int) *In* $\mathsf{Int}(\{n : \mathtt{Int}, m : \mathtt{Int}\})$*, let* M_1 *be* $\langle \mathcal{C}_1, \mathcal{I}_1\rangle$ *where* $\mathcal{C}_1$ *is* $\{\mathtt{Int} \mapsto \mathbb{Z}\}$ *with* $\mathbb{Z}$ *the set of integers* $\{\cdots - 2, -1, 0, 1, 2, \dots\}$*, and* $\mathcal{I}_1$ *is*

- $\mathcal{I}_1(s)$ *with* $s \in \{0, 1, \dots\}$ *is the integer* s *represents;* $\mathcal{I}_1(n)$ *is* 0 *and* $\mathcal{I}_1(m)$ *is* 2*;*
- $\mathcal{I}_1(+)$*,* $\mathcal{I}_1(-)$*,* $\mathcal{I}_1(*)$*,* $\mathcal{I}_1(mod)$ *and* $\mathcal{I}_1(div)$ *are respectively the addition, difference, multiplication, modulo and integer division over integers;*
- $\mathcal{I}_1(<)$*,* $\mathcal{I}_1(>)$*,* $\mathcal{I}_1(\leq)$ *and* $\mathcal{I}_1(\geq)$ *are the usual order relations over integers* – e.g. $\mathcal{I}_1(>)(3, 7)$ *is* $\mathbf{T}$*,* $\mathcal{I}_1(\leq)(0, -1)$ *is* $\mathbf{F}$*;*

is a structure.

Example 5.1.5 (Satisfiability in Int) *In* $\mathsf{Int} \cup \{n : \mathtt{Int}, m : \mathtt{Int}\}$*, we saw that* M_1 *from Example 5.1.4 is a structure structure. Formula*

$$\phi_1 \triangleq \neg(7 + m = 9) \wedge (n + 2 \geq 1)$$

is sat: $M_1 \models \phi_1$*. Note that, if* ψ *is a formula, formula* $\psi \wedge \neg\psi$ *is unsatisfiable, whatever* ψ *is. This follows from the interpretation of* $\wedge$*,* $\neg$*, and the fact that a well-sorted formula* ψ *will always end up being evaluated either to* $\mathbf{T}$ *or* $\mathbf{F}$*. Hence either* $\mathcal{I}(\psi)$ *or* $\mathcal{I}(\neg\psi)$ *is false.*

5.1.3 Logics

Definition 1 (Decision procedures) A *procedure* is an algorithm that does not require any human interaction. A *decision procedure* on a class of polar ("yes/no") questions is a procedure able to answer – *decide* – any of the questions in said class correctly in finite time. In particular, a decision procedure always terminates. □

The problem of automated theorem proving in a signature Σ would be solved[2] by exhibiting a decision procedure for satisfiability, or dually validity, of Σ-formulas.

Unfortunately, for any many-sorted signature with at least one sort symbol σ different from `Bool` and a function symbol ϕ such that $\phi(\sigma_1, \sigma_2) : \sigma_0$ with σ_1 and σ_2 different from `Bool`, no such decision procedure exists. Satisfiability (and validity) of formulas in such signatures is *undecidable*. This is a consequence of a result due to Trakhtenbrot [76]. As a consequence, satisfiability of formulas in the Int signature is undecidable. It is thus necessary to constrain the signatures and the notion of satisfiability to obtain *decidable fragments* where decision procedures exist.

Fortunately, it is often the case in static analysis of programs that the challenges encountered are shallow in the sense that they do not need the full power of expression provided by many-sorted signatures. They can be expressed in decidable fragments. Common restrictions include banning unwanted interpretations, forbidding quantifiers, imposing linearity on arithmetic terms, *etc.*

Definition 5.1.15 (Theory) Given a many-sorted signature Σ, a Σ-*theory* $\mathcal{T}$ is a class of Σ-structures.

Definition 5.1.16 (Logic) A many-sorted first-order *logic* $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ is a signature Σ, a Σ-theory $\mathcal{T}$ and a set of Σ-formulas $\mathcal{F}$.

Definition 5.1.17 (Satisfiability modulo theory) Given a logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$, a Σ-formula ϕ in $\mathcal{F}$ is *satisfiable modulo* $\mathcal{T}$ or $\mathcal{T}$*-consistent* in $\mathcal{L}$ if there exists a structure M in $\mathcal{T}$ such that $M \models \phi$. It is *valid modulo* $\mathcal{T}$ if all the structures in $\mathcal{S}$ are models for ϕ, written $\models_{\mathcal{T}} \phi$.

We adapt the notions of equisatisfiability, consequence and equivalence ($\equiv_{\mathcal{T}}$) from Definition 5.1.14 to modulo theory in the obvious way. Note that the dual relation between validity and satisfiability from Property 5.1.3 still holds. A logic $\mathcal{L}$ is decidable if there exists a decision procedure for the questions “Is ϕ satisfiable modulo $\mathcal{T}$?”, or, dually, “Is ϕ valid modulo $\mathcal{T}$?”, for ϕ in $\mathcal{F}$.

Definition 5.1.18 (Decidability) A logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ is *decidable* if there exists a decision procedure for the questions “Is ϕ satisfiable modulo $\mathcal{T}$?” where ϕ is in $\mathcal{F}$. A logic that is not decidable is called *undecidable*.

At this point we shortly discuss let-bindings. A formula containing let-bindings can always be transformed in a let-binding-free formula by replacing all occurrences of the bound

[2] Although it could require an arbitrary large amount of time.

variables by the terms they are bound to, in a bottom-up manner. So without loss of generality, we will tend to omit handling of let-bindings in the rest of this thesis. This is the case in the following example as well as in the next section about SAT and SMT solving.

Example 5.1.6 (Propositional logic) *The logic $\mathcal{L}_{Prop}(S)$, where S is set of symbols called* propositions, *is composed of*

- *the signature* Core$(\{b : Bool,\ b \in S\})$*;*
- *the theory $\mathcal{T}_{Prop}$ allowing all structures on* Core$(\{b : Bool,\ b \in S\})$*, and*
- *set of formulas $\mathcal{F}$ consisting of all quantifier-free formulas.*

We now give a (naïve) decision procedure based on truth tables, illustrated on an example below. Given a formula ϕ, let $S_\phi \subseteq S$ be the propositions appearing in ϕ. Consider all 2^n possible combinations of truth values for the elements of S_ϕ, and compute the value of all sub-formulas of ϕ in a bottom up manner using the following truth tables on the function symbols for all combinations.

p	q	$\neg p$	$p \vee q$	$p \wedge q$	$p = q$
F	**F**	**T**	**F**	**F**	**T**
F	**T**	**T**	**T**	**F**	**F**
T	**F**	**F**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**

Ultimately, the truth value of ϕ is known for all combinations. Each combination corresponds to a family of structures where the interpretation of S_ϕ is constrained while all the other propositions can be assigned to any value. Together, those 2^n families represent all possible structures. If ϕ evaluates to $\mathbf{T}$ for at least one of the combinations, then ϕ is satisfiable. Dually, if ϕ evaluates to $\mathbf{T}$ for all of them, then ϕ is valid. It is easy to see that this is a decision procedure: it computes all possible truth values a formula can be evaluated to by a structure and hence can decide the satisfiability (or validity) of ϕ.

For example, if $b_1, b_2 \in S$, let $\psi \triangleq b_1 \Rightarrow b_1 \wedge (b_2 \vee \neg b_2)$, which is really $\neg b_1 \vee (b_1 \wedge (b_2 \vee \neg b_2))$. Then $S_\psi = \{b_1, b_2\}$. There are $2^2 = 4$ combinations of values for the pair (b_1, b_2):

(b_1, b_2)	$(\mathbf{F}, \mathbf{F})$	$(\mathbf{F}, \mathbf{T})$	$(\mathbf{T}, \mathbf{F})$	$(\mathbf{T}, \mathbf{T})$
$\neg b_1$	**T**	**T**	**F**	**F**
$b_2 \vee \neg b_2$	**T**	**T**	**T**	**T**
$b_1 \wedge (b_2 \vee \neg b_2)$	**F**	**F**	**T**	**T**
$\psi \triangleq \neg b_1 \vee (b_1 \wedge (b_2 \vee \neg b_2))$	**T**	**T**	**T**	**T**
$\neg\psi$	**F**	**F**	**F**	**F**

All combinations make ψ evaluate to $\mathbf{T}$, so ψ is valid. Dually, $\neg\psi$ is unsatisfiable.

5.1.4 Proof Systems

Unfortunately, it is not always possible to represent all structures and exhaustively enumerate the values a formula can be evaluated to. It can be easier to perform syntactic manipulations on formulas using an appropriate formalism in order to to reach a conclusion on the decision problem at hand.

Proof theory is the study of the syntactical structure of terms with respect to a *proof system* $\mathcal{P}$. Terms are viewed as trees of symbols and are manipulated independently from the semantics. There are different types of proof systems, the most common ones being Hilbert/Frege calculi, natural deduction calculi and sequent calculi – also known as Gentzen systems. A proof system $\mathcal{P}$ can be thought of as a collection of *rules of inference* associating some *premises* $P_1, P_2, \ldots, P_n$ ($n \geq 0$) with a *conclusion* C. They are usually represented as

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}\ (r)$$

which means that C follows from P_1 and P_2 and ... and P_n by rule (r). A rule of inference with no premises ($n = 0$) is called an *axiom*. A *proof* in $\mathcal{P}$ of a formula ϕ is a *derivation* of ϕ, *i.e.* a finite tree of inference rule applications, with axioms as leaves. For instance,

$$\cfrac{\cfrac{}{\phi_1}\ (r_2) \qquad \cfrac{\cfrac{\cfrac{}{\phi_5}\ (r_6)}{\phi_3}\ (r_4) \qquad \cfrac{}{\phi_4}\ (r_5)}{\phi_2}\ (r_3)}{\phi}\ (r_1)$$

where $(r_1), \ldots, (r_6)$ are rules of inference of $\mathcal{P}$. This is usually written $\vdash_{\mathcal{P}} \phi$, or simply $\vdash \phi$ when $\mathcal{P}$ is obvious from the context. In the following, we will write $\{\psi_1, \ldots, \psi_n\} \vdash_{\mathcal{P}} \phi$ for $\vdash_{\mathcal{P}} (\psi_1 \wedge \cdots \wedge \psi_n) \Rightarrow \phi$ which is really $\vdash_{\mathcal{P}} \neg(\psi_1 \wedge \cdots \wedge \psi_n) \vee \phi$. This means that assuming "ψ_1 and ... and ψ_n, ϕ is provable", or "ψ_1 and ... and ψ_n *entails* ϕ". When $n = 1$, we write $\psi_1 \vdash \phi$.

Ideally provability and validity should be correlated. The next two definitions state the most important properties a proof system can have with respect to semantics.

Definition 5.1.19 (Soundness) A proof system is *sound* with respect to the semantics of a logic $\mathcal{L}$ if every formula provable in the proof system is valid in $\mathcal{L}$.

Definition 5.1.20 (Completeness) A proof system is *complete* with respect to the semantics of a logic $\mathcal{L}$ if every valid formula in $\mathcal{L}$ is provable in the proof system.

Property 5.1.4 Given a logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ and a sound and complete proof system for $\mathcal{L}$, a decision procedure for the provability of formulas in $\mathcal{F}$ is also a decision procedure for the validity of formulas in $\mathcal{F}$.

The three most common proof systems (Hilbert/Frege calculi, natural deduction calculi and sequent calculi) are all sound and complete for many-sorted signatures and their associated semantics. Note that proofs can be checked *mechanically* by verifying that only legal inference rule applications are used.

5.2 SAT and SMT Solving

The Boolean SATisfiability problem (SAT) is the problem of deciding the satisfiability of a formula in $\mathcal{L}_{Prop}$ modulo $\mathcal{T}_{Prop}$. The Satisfiability Modulo Theory problems (SMT) is the class of problems of deciding the satisfiability of a formula in an arbitrary (first-order) logic modulo the underlying theory, assuming satisfiability of conjunction of atoms is decidable. This section details SAT and SMT solving, which has reached a very high degree of maturity during the last two decades [9] thanks to the pioneer work of [27, 28] and breakthroughs such as the Chaff SAT solver [59]. The SMT-LIB initiative [4] aims at providing rigorous descriptions of theories encountered in SMT solving, unifying SMT solvers interfaces, and building large libraries of benchmarks in different theories. logic standard only

We present SAT and SMT solving in the general context of a logic with underlying theory $\mathcal{T}$ in which the satisfiability of conjunctions of literals is decidable. In the restrictive case of SAT, $\mathcal{T} = \mathcal{T}_{Prop}$. Most modern SAT solvers are based on a technique called the Davis-Putnam-Logemann-Loveland (DPLL [27, 28]) procedure and do not follow the naïve approach detailed in Example 5.1.6. DPLL seeks a model for a propositional formula ϕ by completing a partial evaluation function M iteratively. In [60] a framework called *abstract DPLL* is introduced, giving means to express and reason on DPLL and its many variations and extensions, as well as its adaptations to solving SMT queries. We thus reuse the notation, definitions and theorems from [60] to give an overview of the techniques encountered in SAT and SMT solving. Logic, theory, entailment, *etc.* are understood as defined in Section 5.1.

Notation. Let $\mathcal{L}$ be a logic. We remind that a *clause* is a disjunction of literals, *i.e.* of the form $l_1 \vee \cdots \vee l_n$ where $l_1, \ldots, l_n$ are literals. A formula is in *Conjunctive Normal Form*, or CNF, if it is a conjunction of clauses, *i.e.* of the form $C_1 \wedge \cdots \wedge C_n$ where $C_1, \ldots, C_n$ are clauses. In this section we will see clauses a sets of literals, and CNF formulas as sets of clauses. We allow ourselves to write clause C as $C' \vee l$ when $l \in C$ and $C' = C \setminus \{l\}$. A *partial evaluation function* M is a set of literals: a literal l is true in M if l is an element of M, written $l \in M$. Literal l is false in M if $\neg l \in M$. A clause C is true in M ($M \models C$) if $C \cap M \neq \emptyset$, where $\emptyset$ is the empty set. If all the literals of clause C are false in M, then C is false in M, or is a *conflict clause* in M ($M \models \neg C$). If neither $M \models C$ nor $M \models \neg C$ then C is undefined in M. A CNF formula ϕ is true in M, or $M \models \phi$, if all its clauses are true in M; if $M \models \phi$, and M is

$\mathcal{T}$-consistent, then M is a model of ϕ.

In this section we will only consider formulas in CNF. This is without loss of generality since any formula can be converted in a CNF formula in linear time by introducing extra constant symbols [77]. In abstract DPLL, a *DPLL procedure* is viewed as a procedure returning a sequence of states corresponding to a proof of $\mathcal{T}$-consistency, or inconsistency, of a CNF formula. A state is either S_{fail} or a pair (M, ϕ), written $M \parallel \phi$, where ϕ is a CNF formula and partial evaluation function M is a sequence of potentially annotated literals l^d. Annotated literals are called *decision literals*. The empty sequence is denoted $\emptyset$. The concatenation of sequences of literals M and M' is written MM', and appending a literal l to a sequence M is noted Ml.

Definition 5.2.1 (Abstract DPLL) Given a formula ϕ, a DPLL procedure is a state machine where

- $\emptyset \parallel \phi$ is the only initial state,
- S_{fail} and states of the form $M \parallel \phi'$ where $M \models \phi$ and M is $\mathcal{T}$ consistent are the final states, and
- transitions are specified by transition rules.

We write $s \implies s'$ for "a transition between states s and s' exists". The reflexive transitive closure of $\implies$ is noted $\stackrel{*}{\implies}$, and we write $s \stackrel{!}{\implies} s'$ for $s \stackrel{*}{\implies} s'$ and s' is final state.

5.2.1 SAT solving

This section focuses on SAT solving, algorithms addressing the Boolean satisfiability problem, or SAT, in order to introduce the basic ideas behind DPLL. The logic used in the following is propositional logic $\mathcal{L}_{Prop}(S)$ from Example 5.1.6. The underlying theory here is then $\mathcal{T}_{Prop}$; since it does not restrict the legal structures in any way, $\mathcal{T}_{Prop}$-consistency is the same as satisfiability of quantifier-free formulas. Note that in this context literals can only be constant function symbols of sort `Bool` or its negation.

Definition 5.2.2 (Basic DPLL) The *Basic DPLL system* consists of the following transition rules:

UnitPropagate:

$$M \parallel \phi, C \vee l \implies Ml \parallel \phi, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M. \end{cases}$$

Decide:

$$M \parallel \phi \implies Ml^d \parallel \phi \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } \phi \\ l \text{ is undefined in } M. \end{cases}$$

Backjump:

$$Ml^dM' \parallel \phi \implies Ml' \parallel \phi \quad \textbf{if} \begin{cases} \text{there is some clause } C \vee l' \text{ such that:} \\ \bullet\ \phi \models_{\mathcal{T}} C \vee l' \text{ and } M \models \neg C, \\ \bullet\ l' \text{ is undefined in } M \text{ and} \\ \bullet\ l' \text{ or } \neg l' \text{ occurs in a clause of } \phi. \end{cases}$$

Fail:

$$M \parallel \phi, C \implies S_{fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals.} \end{cases}$$

Informally, the UnitPropagate rule corresponds to an automatic consequence of the current partial evaluation function: indeed, when $M \models \neg C$ and l is undefined in M, the only way to complete partial evaluation function M in a model $M' \supset M$ for formula $\phi, C \vee l$ is that $M' \models C \vee l$ and therefore $M' \models l$. The Decide rule can be thought of as heuristically choosing an unassigned literal l, and heuristically adding l or $\neg l$ to M. It is also known as *branching* or *case splitting*. In a state $M \parallel \phi$, a clause C of ϕ is a conflict clause if $M \models \neg C$. In such states, it is shown in [60] that either the Fail rule or the Backjump rule applies. Backjump undoes a decision l^d made by Decide if Ml^dM' cannot be completed in a model for ϕ because of l^d.

Example 5.2.1 (DPLL) *Consider the CNF formula* $\phi \triangleq \{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$ *where* a*,* b*,* c *and* d *are propositions. An example of a legal DPLL run on* ϕ *follows:*

	$\emptyset$				$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	*apply Decide*
$\implies$	a^d				$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	*apply UnitPropagate*
$\implies$	a^d	b			$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	*apply Decide*
$\implies$	a^d	b	c^d		$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	*apply UnitPropagate*
$\implies$	a^d	b	c^d	$\neg d$	$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	*apply Backjump* ($C \vee l' = \neg a \vee \neg c$)
$\implies$	a^d	b	$\neg c$		$\parallel$	$\{\neg a \vee b,\ \neg c \vee \neg d,\ \neg b \vee \neg c \vee d\}$	

Definition 5.2.3 (Basic DPLL procedure) A *Basic DPLL procedure* is any procedure taking an input CNF formula ϕ and computing a sequence $\emptyset \parallel \phi \overset{!}{\implies} s$.

Property 5.2.1 Basic DPLL procedures are decision procedures for the satisfiability of CNF propositional formulas. That is, given a CNF formula ϕ, there exists no infinite sequence of states starting from $\emptyset \parallel \phi$, and

- $\emptyset \parallel \phi \overset{!}{\Longrightarrow} S_{fail}$ if and only if ϕ is unsatisfiable.
- $\emptyset \parallel \phi \overset{!}{\Longrightarrow} M \parallel \phi$ if and only if ϕ is satisfiable. Also, in this case, M is a model of ϕ.

This is a strong result, since it means that no matter the strategy chosen for applying the rules, it results in a decision procedure for SAT. It is now possible to add new rules or to alter existing ones in order to express the different approaches to modern efficient SAT solving, while making sure the properties of the original system are preserved. Actually, the Backjump rule is already an evolution of the original (chronological) Backtrack rule, which switches the last decision made:

Backtrack:

$$Ml^dM' \parallel \phi \quad \Longrightarrow \quad M\neg l \parallel \phi \quad \textbf{if} \quad \begin{cases} Ml^dM' \models \neg\phi \\ M' \text{ does not contain any decision literal.} \end{cases}$$

While chronological backtracking might require many transitions to realize a decision it made a while ago should be switched, the Backjump rule gives room for more efficient backtracking strategies able to cancel many decisions in one step.

In the definition of backjumping, the clause $C_{cdcl} = C \vee l'$ such that $\phi \models_{\mathcal{T}} C \vee l'$ and $M \models \neg C$ is interesting. Assume we are backjumping from a state with a conflict clause. If we had considered $\phi \cup \{C \vee l'\}$ instead of ϕ then potentially many bad decisions could have been avoided. Moreover, it might be the case that *learning* this clause could avoid future bad decisions. This observation leads to one of the most fundamental techniques in DPLL-based SAT solving: *(conflict-driven) clause learning* [70, 69]. Since $\phi \models C_{cdcl}$, ϕ is equivalent to $\phi \cup \{C_{cdcl}\}$. Although equivalent, these two formulas can cause a DPLL procedure to behave very differently. The Learn rule *learns* such clauses without sacrificing correctness or termination. A learnt clause is called a *lemma*. It is complemented by a Forget rule to avoid an explosion of the size of the formula by *forgetting* lemmas. For instance , lemmas which have not been involved in conflicts or used as unit clauses in some time.

Definition 5.2.4 (Learning DPLL) *Learning DPLL* is the system which transition relation is denoted $\Longrightarrow_L$, obtained by adding the rules Learn and Forget given below to Basic DPLL.

Learn:

$$M \parallel \phi \quad \Longrightarrow \quad M \parallel \phi, C \quad \textbf{if} \quad \begin{cases} \text{all atoms of } C \text{ occur in } \phi \\ \phi \models_{\mathcal{T}} C. \end{cases}$$

Forget:

$$M \parallel \phi, C \quad \Longrightarrow \quad M \parallel \phi \quad \textbf{if} \quad \begin{cases} \phi \models_{\mathcal{T}} C \end{cases}$$

Note that in *Learning DPLL*, *i.e.* Basic DPLL with Learn and Forget, there is no risk to learn infinitely many different clauses: as there is a finite number of literals appearing in ϕ, there is only a finite number of clauses that can be constructed using them. However, there is a danger to learn the same clause forever. This can be avoided by forcing one of the rules of Basic DPLL between two applications of Learn. Since Learn and Forget replace the formula of the current state by an equivalent one, and there is no infinite sequence of states in Basic DPLL, then there is no infinite sequence of states in Basic DPLL with Learn and Forget.

Property 5.2.2 (Termination of Learning DPLL) There exist no infinite sequences of states starting from $\emptyset \parallel \phi$ if no clause C is learnt infinitely many times along a sequence. Also, Learning DPLL procedures are decision procedures for the satisfiability of CNF propositional formulas.

Clause learning is a crucial component of modern DPLL solvers. Usually a *dependency graph* tracking the reason why a literal was assigned a value is maintained and used to learn concise lemmas as soon as a conflict is detected. Such techniques, such as the *first UIP* technique [69], achieve a very high level of efficiency but escape the scope of this thesis. Another interesting widespread technique is that of *reset*. SAT solver developers noticed that *bad decisions* made at early stages of the procedure can cause a huge amount of time consuming iterations before backjumping them. In practice, it can be more efficient to cancel the run and start a new one using the clauses learnt so far to avoid such early bad decisions [42].

5.2.2 SMT solvers

Let $\mathcal{L}$ be a logic with a non-empty theory $\mathcal{T}$. SMT solvers address the SMT problem by combining SAT solving with a *theory specific solver* or *$\mathcal{T}$-solver*. $\mathcal{T}$-solvers are tools checking the satisfiability of a conjunction of literals – viewed as a set of literals – in the logic associated to the theory they work on – *e.g.* the theory of arrays, linear integer arithmetic...

A first simple approach to SMT solving is to ask the SAT solver for a propositional model of the CNF formula ϕ where atoms are all viewed as propositional symbols. If none can be found, *i.e.* ϕ considered as a propositional formula is unsat, then *a fortiori* ϕ is $\mathcal{T}$-inconsistent. If a model M is returned, M is either $\mathcal{T}$-consistent and ϕ is satisfiable, or it is $\mathcal{T}$-inconsistent and the SAT solver is relaunched on $\phi \wedge \psi$ where ψ is such that $\psi \models \neg M$. The following rule enforces this mechanism.

Definition 5.2.5 (Very Lazy Theory DPLL) *Very Lazy Theory DPLL* is the system which transition relation is denoted $\Longrightarrow_{VLT}$, obtained by adding the rule Very Lazy Theory Learning, given below, to the Learning DPLL system.

Very Lazy Theory Learning:

$$M l M' \parallel \phi \quad \Longrightarrow \quad \emptyset \parallel \phi, \neg l_1 \vee \cdots \vee \neg l_n \vee \neg l \quad \textbf{if} \quad \begin{cases} M l M' \models \phi \\ \{l_1, \ldots, l_n\} \subseteq M \\ l_1 \wedge \cdots \wedge l_n \models_{\mathcal{T}} \neg l \end{cases}$$

This approach is called *lazy*: no information from the theory is used besides propositional model validation or invalidation. As was the case with Learn and Forget, this new rule jeopardizes termination of the procedure: it must not be possible to learn the same clause infinitely many times. Note that by forgetting a lemma learnt by applying Very Lazy Theory Learning, DPLL will potentially have to learn it again in order to avoid finding the same propositional partial evaluation function again.

Example 5.2.2 (Very Lazy Theory DPLL on QF_LIA) *Quantifier-Free Linear Integer Arithmetic (*QF_LIA*) is a logic defined in the SMT-LIB 2 standard as having sorts* `Bool` *and* `Int`*, usual arithmetic function symbols ($+$, $-$, $*$, $>$, $\leq$, $\ldots$) which are interpreted as expected and only accepting quantifier-free formulas. Its underlying theory is noted $\mathcal{T}_{LinInts}$. In* QF_LIA$(\{n : Int\})$*, consider the CNF formula*

$$\phi \triangleq \{\underbrace{n = 2}_{l_1} \vee \underbrace{n \geq 5}_{l_2}, \quad \underbrace{n + 2 = 4}_{l_3} \vee \underbrace{n < 0}_{l_4}, \quad \underbrace{n < 2}_{l_5} \vee \underbrace{n \leq 10}_{l_6}\}.$$

From a propositional point of view, it is the same as $\phi \triangleq \{l_1 \vee l_2, \quad l_3 \vee l_4, \quad l_5 \vee l_6\}$. It is not hard to find a (propositional) model for ϕ, say $l_1^d\ l_3^d\ l_5^d$. At this point, if $l_1^d\ l_3^d\ l_5^d$ is $\mathcal{T}_{LinInts}$-consistent then so is ϕ. But $l_1 \wedge l_3 \models_{\mathcal{T}_{LinInts}} \neg l_5$, or more explicitly $n = 2 \wedge n + 2 = 4 \models_{\mathcal{T}_{LinInts}} \neg(n < 2)$.
So applying rule Very Lazy Theory Learning *with $M = l_1^d\ l_3^d$ and $l = l_5^d$ and $M' = \emptyset$, DPLL learns the clause $\neg l_1 \vee \neg l_3 \vee \neg l_5$. It is easy to see that in order to establish $\mathcal{T}_{LinInts}$-consistency of ϕ, potentially many lemmas will be learnt depending on the decision heuristics at the propositional level, implying many calls to the $\mathcal{T}_{LinInts}$-solver.*

The idea to improve on this first approach is to better integrate $\mathcal{T}$-inconsistency in the search for a model, while avoiding to restart with an empty partial evaluation function.

Definition 5.2.6 (Lazy Theory DPLL) *Lazy Theory DPLL* is the system which transition relation is denoted $\Longrightarrow_{VL}$, obtained by adding the rule Lazy Theory Learning, given below, to the Learning DPLL system.
Lazy Theory Learning:

$$M l M' \parallel \phi \quad \Longrightarrow \quad M l M' \parallel \phi, \neg l_1 \vee \cdots \vee \neg l_n \vee \neg l \quad \textbf{if} \quad \begin{cases} \{l_1, \ldots, l_n\} \subseteq M \\ l_1 \wedge \cdots \wedge l_n \models_{\mathcal{T}} \neg l \\ \neg l_1 \vee \cdots \vee \neg l_n \vee \neg l \notin \phi \end{cases}$$

Now, if $\{l_1, \ldots, l_n\} \subseteq M$ then $M\,l \models \neg(\neg l_1 \vee \cdots \vee \neg l_n \vee \neg l)$. The Lazy Theory Learning rule therefore always leads to a conflict, so either Fail or Backjump should be applied directly afterwards to handle the $\mathcal{T}$-inconsistency.

Property 5.2.3 The (*Very*) *Lazy Theory DPLL* procedures are decision procedures for the $\mathcal{T}$-consistency of CNF formulas, as long as no clause C is learnt by Learn or (Very) Lazy Theory Learning infinitely many times along a sequence.

While some solvers rely on this kind of lazy approach, such as CVC4 [3], it is possible to integrate DPLL and the theory solver in a tighter way. This family of approaches, known as DPLL($\mathcal{T}$), consists in consulting the $\mathcal{T}$-solver during the propagation phase whenever possible – *a.k.a. eager theory propagation*.

Definition 5.2.7 (DPLL($\mathcal{T}$)) DPLL($\mathcal{T}$) with eager theory propagation is the system which transition relation is written $\Longrightarrow_{Edpll(\mathcal{T})}$, obtained by adding the rule Theory Propagate to the DPLL with learning system:

Theory Propagate:

$$M \parallel \phi \quad \Longrightarrow \quad M\,l \parallel \phi \quad \textbf{if} \quad \begin{cases} M \models_{\mathcal{T}} l \\ l \text{ or } \neg l \text{ occurs in a clause of } \phi \\ l \text{ is undefined in } M. \end{cases}$$

Also, Theory Propagate has priority over all the other rules.

Property 5.2.4 DPLL($\mathcal{T}$) with eager theory propagation procedures are decision procedures for the $\mathcal{T}$-consistency of CNF formulas, as long as no clause C is learnt by Learn infinitely many times along a sequence.

Since it has property over all other rules, Theory Propagate forces DPLL($\mathcal{T}$) to take into account the consequences of the current state of the partial evaluation function. This is illustrated on the following example.

Example 5.2.3 (DPLL($\mathcal{T}_{LinInts}$) with eager theory propagation) *In* QF_LIA($\{n : Int\}$)*, consider again the CNF formula*

$$\phi \triangleq \{\underbrace{n = 2}_{l_1} \vee \underbrace{n \geq 5}_{l_2}, \quad \underbrace{n + 2 = 4}_{l_3} \vee \underbrace{n < 0}_{l_4}, \quad \underbrace{n < 2}_{l_5} \vee \underbrace{n \leq 10}_{l_6}\}.$$

Assume that the DPLL procedure starts with a decision, say l_1^d. Since Theory Propagate is applicable on l_3 ($(n = 2)^d \models_{\mathcal{T}_{LinInts}} n + 2 = 4$) and l_6 ($(n = 2)^d \models_{\mathcal{T}_{LinInts}} n \leq 10$) and has priority over the other rules, a $\mathcal{T}_{LinInts}$-consistent model – either $l_1^d\ l_3\ l_6$ or $l_1^d\ l_6\ l_3$ – is found in two transitions.

Eager theory propagation can become very expensive as it requires exhaustive examination of the unassigned literals in the $\mathcal{T}$-solver to see which are $\mathcal{T}$-consequences of the current partial evaluation function. In practice, most DPLL($\mathcal{T}$) implementations apply non-exhaustive theory propagation, along with lazy theory learning to ensure that $\mathcal{T}$-conflicts are detected and handled.

The logics used in the rest of this thesis are taken from the SMT-LIB 2 standard: *(i)* Quantifier-Free Linear Integer Arithmetic (QF_LIA), *(ii)* Quantifier-Free Linear Real Arithmetic (QF_LRA), and *(iii)* Array, Uninterpreted Functions and Linear Integer and Real Arithmetic (AUFLIRA). On several occasions we will refer to Quantifier-Free Linear Real Arithmetic (QF_LIRA) which does not exist in the SMT-LIB standard. We use this name to characterize precisely the logic we are talking about and avoid confusion with arrays and uninterpreted functions found in the closest SMT-LIB logic, AUFLIRA. In practice however, SMT queries dealing with formulas in QF_LIRA really take place in AUFLIRA. AUFLIRA is based on the `Reals_Ints` signature which features the `Int` sort, the `Real` sort and most usual arithmetic operators, as well as functions such to_int, $mod \ldots$ which we will not use in this thesis. Theory $\mathcal{T}_{\texttt{Reals_Ints}}$ restricts legal structures so that operators such as $+, * \ldots$ behave as expected in integer and real arithmetic. AUFLIRA restricts formulas such that only linear arithmetic terms are allowed. In AUFLIRA, arithmetic constants are of the form $0, 1, 42, \ldots$ (numerals) and $0.0, 54.54327, \ldots$ (decimals).

5.3 SMT: Standard, Implementations and Additional Features

Most modern SMT solvers are compliant with the SMT-LIB 2 standard [4], which provides a unified interface for users and benchmarks thanks to SMT-LIB *commands*. Today's prominent SMT-LIB 2 compliant solvers are

- Z3 [29], developed at Microsoft Research,
- MathSAT 5 [20] from the University of Trento, and
- CVC4 [3] developed jointly at the University of Iowa and New York University.

Let us briefly present the SMT-LIB 2 standard specification for interactions with SMT solvers. We omit the details regarding setting the logic and declaring function symbols, and instead focus on the *assert* and *check-sat* commands. Most modern SAT and SMT solvers maintain an internal state, or context. The assert command takes a formula as argument and adds it to the context of the solver. The check-sat command takes no argument and checks for the satisfiability of the conjunction of all the formulas in the context, returning

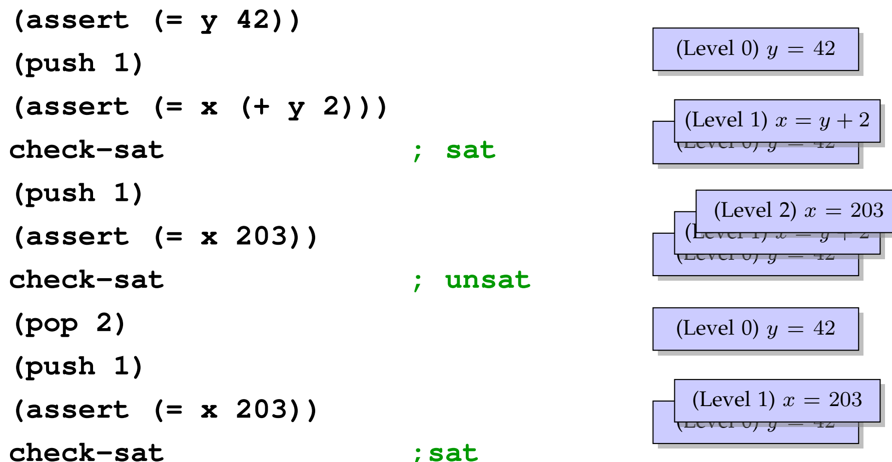


Figure 5.1: Push and pop example, assertion stack on the right.

"sat", "unsat" or "unknown". After a check-sat, it is possible to assert additional formulas and perform check-sats at any point. For instance:

```
; Logic is linear integer arithmetic,
; x and y are constants of sort Int.
(assert (and (>= x 0) (= y 42)))
(assert (= x (+ y 2)))
check-sat
; (x >= 0) and (y = 42) and (x = y + 2) is sat.
(assert (= x 203))
check-sat
; unsat.
```

As discussed above, one of the key features of the DPLL($\mathcal{T}$) approach is conflict driven clause learning. Information learnt during a check-sat of $\phi \triangleq \phi_1 \wedge \cdots \wedge \phi_n$ is stored in the context for the next analysis. Indeed, it is implied by ϕ: when additional formulas are asserted, this information is still valid: the solver does not have to start from scratch and potentially re-learn everything it learnt the first time.

Solvers can go further and maintain an *assertion stack*. Formulas can be asserted at an assertion level n, and check-sats now check the satisfiability of all the formulas asserted at each level of the stack. The *push* and *pop* commands lets the user navigate in the stack, as illustrated on Figure 5.1.

Last, modern SAT solvers are usually also able to produce an *unsat core* when a CNF formula ϕ is unsatisfiable. An unsat core $\mathcal{UC}$ of ϕ is a set of mutually unsatisfiable clauses of

ϕ. It is a *minimal* explanation of why ϕ is unsatisfiable. In general *minimal* means minimal for inclusion, *i.e.* any strict subset of $\mathcal{UC}$ is satisfiable. It would be better for $\mathcal{UC}$ to be minimal in the sense of cardinality on the set of clauses of ϕ, *i.e.* such that there is no unsatisfiable subset of clauses of ϕ with less elements than $\mathcal{UC}$. Computing such an unsat core is very expensive in practice, while a minimal unsat core with respect to inclusion can be easily extracted from a DPLL run with appropriate bookkeeping.

In practice, the granularity of the unsat core is that of the assert commands: when asserting a formula it is possible to give it a name. Upon an unsat result, the *get-unsat-core* command returns the name of formulas which were found to be in conflict. For example:

```
...
(assert (! (>= x 0) :named xGE0))
(assert (! (= y 42) :named yEQ42))
(assert (! (= x (+ y 2)) :named xEQyPL2))
(assert (! (= x 203) :named xEQ203))
(assert (! (= x 44) :named xEQ44))
check-sat
; unsat.
get-unsat-core
; (yEQ42 xEQyPL2 xEQ203)
```

Notice that the unsat core returned is not minimal with respect to cardinality, as `xEQ44` and `xEQ203` is also an unsat core. Which unsat core is returned depends on the order in which the solver considers the formulas asserted. When given the commands above, Z3 produces `(yEQ42 xEQyPL2 xEQ203)`. But by changing the order of the asserts to

```
...
(assert (! (>= x 0) :named xGE0))
(assert (! (= x 44) :named xEQ44)) ; <- has moved.
(assert (! (= y 42) :named yEQ42))
(assert (! (= x (+ y 2)) :named xEQyPL2))
(assert (! (= x 203) :named xEQ203))
...
```

then Z3 yields `(xEQ44 xEQ203)` as the unsat core.

Incrementality is extensively used in implementations of verification techniques as it can make a difference of several orders of magnitude. Unsat core on the other hand provide more information on an unsat result for virtually no cost, as it is extracted from the proof of unsatisfiability. Often, an unsat core query can replace several satisfiability checks, as we will see in the next chapter.

CHAPTER 6

Transition Systems, State Invariant Verification

SAT and SMT solvers are very powerful tools able to solve large scale satisfiability problems expressed in various decidable fragments of many-sorted first-order logic (MSFOL). A model on the other hand is written in a synchronous language such as Lustre: transposing a verification problem of a model to MSFOL formulas is far from straightforward.

Synchronous dataflow models can be compiled to a first intermediary formalism, *transition systems* from which MSFOL formulas can be extracted to conduct an analysis. Transition systems can be represented as *transition graphs*: directed graphs where node represent states and edges legal transitions between two states. While we will use it to illustrate examples of transition systems, this explicit representation is inconvenient to represent systems having a large or infinite number of states. In Section 6.1 we present a way to represent such structures intentionally – *i.e.* without enumerating all the states and transitions explicitly – using formulas in a many-sorted logic. Informally, a *transition system* is thought of in terms of execution as starting from a set of *initial states* characterized by an MSFOL predicate on the state variables of the system. All legal *transitions* between states are also expressed as an MSFOL predicate. We introduce this notion formally in Section 6.1. *Reachability verification* consists in attempting to prove or disprove that a transition system can never reach a violation of a given *proof objective* (PO), given as a formula over the state variables. Two conclusions can be reached: *(i)* a proof is found that the proof objective holds, or *(ii)* a succession of states is found, starting from an initial state s_0 and reaching a state where the proof objective is violated, showing that the proof objective does not hold. Additionally, the verification can fail to reach any conclusion, in which case the status of the proof objective is unknown. We review the different approaches to transition system reachability verification in Section 6.2 and Section 6.3, before presenting the outline of this thesis.

6.1 Transition Systems

We now assume a logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ in MSFOL where all the Σ-structures appearing in $\mathcal{T}$ use exactly the same carrier function $\mathcal{C}$. That is to say any sort $\sigma \in \Sigma$ can only have one carrier: $\mathcal{C}(\sigma)$. A "concrete value of sort $\sigma \in \Sigma$" is an element of $\mathcal{C}(\sigma)$. Also, we assume that all models can be represented as concrete values of coherent sorts. This is the case for instance for quantifier-free linear integer arithmetic, quantifier-free linear integer real arithmetic (the logic we will heavily rely on in Part II), but not quantifier-free non-linear real arithmetic.

A convenient way to represent a transition system in MSFOL is to encode the initial states and the transition relation as formulas on the state variables of the system. Initial states are the models of the *initial formula*, a formula on the state variables of the system. The transition relation is encoded as a formula on the state variables – viewed as the *current state* – and primed versions of the state variables – thought of as the *next state*. The models of the *transition formula* characterize pairs of states between which a transition exists.

Definition 6.1.1 (Transition system) Given a logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$, a *transition system S* is defined as a tuple $\langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ where

- Var is the vector of symbols called the *state variables* $\langle v^1, v^2, \ldots, v^m \rangle$ of S and $\mathcal{D}$ is a function mapping each element of Var to a sort from signature Σ;

- I is a well-sorted Σ-formula in the environment $\{v^1 : \mathcal{D}(v^1), \ldots, v_m : \mathcal{D}(v^m)\}$ called the *initial formula* of the system;

- T is a well-sorted Σ-formula in the environment $\{v^1 : \mathcal{D}(v^1)\ ,\ldots, v^m : \mathcal{D}(v^m),\ \ v'^1 : \mathcal{D}(v^1), \ldots, v'^m : \mathcal{D}(v^m)\}$ called the *transition formula* of the transition system.

Definition 6.1.2 Given a logic $\mathcal{L} \triangleq \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ and a transition system $S \triangleq \langle \{v^1, \ldots, v^m\}, \mathcal{D}, I, T \rangle$, $\mathcal{L}^S$ is the logic

$$\mathcal{L}(\{v^1_{\mathbf{0}} : \mathcal{D}(v^1), \ldots, v^m_{\mathbf{0}} : \mathcal{D}(v^m)\} \cup \{v^1_{\mathbf{1}} : \mathcal{D}(v^1), \ldots, v^m_{\mathbf{1}} : \mathcal{D}(v^m)\} \cup \ldots).$$

$\mathcal{L}^S$ contains additional constant function symbols allowing to represent – and distinguish – state variables at each execution step. We call them *unrolled state variables*.

Notation. So I and T are both well-sorted formulas in their respective environment in $\mathcal{L}^S$, and the annotated free constants $s^i_{\mathbf{j}}$ give us means to *unroll* the transition formula. For

instance, a formula encoding a sequence of four states can be easily constructed as

$$\begin{aligned}&\Big(\textbf{let}\ (v^1 = v^1_{\mathbf{0}},\ \ldots,\ v^m = v^m_{\mathbf{0}},\ v'^1 = v^1_{\mathbf{1}},\ \ldots,\ v'^m = v^m_{\mathbf{1}})\ \textbf{in}\ T\Big)\\ \wedge\ &\Big(\textbf{let}\ (v^1 = v^1_{\mathbf{1}},\ \ldots,\ v^m = v^m_{\mathbf{1}},\ v'^1 = v^1_{\mathbf{2}},\ \ldots,\ v'^m = v^m_{\mathbf{2}})\ \textbf{in}\ T\Big)\\ \wedge\ &\Big(\textbf{let}\ (v^1 = v^1_{\mathbf{2}},\ \ldots,\ v^m = v^m_{\mathbf{2}},\ v'^1 = v^1_{\mathbf{3}},\ \ldots,\ v'^m = v^m_{\mathbf{3}})\ \textbf{in}\ T\Big).\end{aligned}$$

More concisely, by writing $s \triangleq \langle v^1, \ldots, v^m\rangle$, $s' \triangleq \langle v'^1, \ldots, v'^m\rangle$, $s_{\mathbf{j}} \triangleq \langle v^1_{\mathbf{j}}, \ldots, v^m_{\mathbf{j}}\rangle$ and abusing let-binding notation:

$$(\textbf{let}\ (s = s_{\mathbf{0}},\ s' = s_{\mathbf{1}})\ \textbf{in}\ T)\ \wedge\ (\textbf{let}\ (v = s_{\mathbf{1}},\ s' = s_{\mathbf{2}})\ \textbf{in}\ T)\ \wedge\ (\textbf{let}\ (v = s_{\mathbf{2}},\ s' = s_{\mathbf{3}})\ \textbf{in}\ T).$$

Even more concisely, by writing $T(s_{\mathbf{i}}, s_{\mathbf{j}})$ for $\textbf{let}\ (s = s_{\mathbf{i}},\ s' = s_{\mathbf{j}})\ \textbf{in}\ T$:

$$T(s_{\mathbf{0}}, s_{\mathbf{1}})\ \wedge\ T(s_{\mathbf{1}}, s_{\mathbf{2}})\ \wedge\ T(s_{\mathbf{2}}, s_{\mathbf{3}})$$

By adopting similar notation for I, it is possible to require that the sequence starts from an initial state:

$$I(s_{\mathbf{0}})\ \wedge\ T(s_{\mathbf{0}}, s_{\mathbf{1}})\ \wedge\ T(s_{\mathbf{1}}, s_{\mathbf{2}})\ \wedge\ T(s_{\mathbf{2}}, s_{\mathbf{3}})$$

Assuming it is satisfiable, a model of this formula gives concrete values to the four unrolled states $s_{\mathbf{0}}$, $s_{\mathbf{1}}$, $s_{\mathbf{2}}$ and $s_{\mathbf{3}}$. We will use this notation throughout the rest of this thesis, and in particular will specify initial and transition formulas in a functional form, *e.g.*

$$I(\langle v^1, v^2\rangle) \triangleq v^1 \geq 0\ \wedge\ v^2 = 3.$$

So, $s = \langle v^1, v^2\rangle$ is the vector of state variables: v^1 and v^2 are variables of $\mathcal{L}^S$. Vector $\langle 2, 3\rangle$ is a *concrete* (initial) state: 2 and 3 are integers. We abuse notation further and identify constants the interpretation of which is imposed with their actual value: for instance, symbol "1" with the integer 1. This allows us to write $I(\langle 2, 3\rangle) \triangleq 2 \geq 0\ \wedge\ 3 = 3$ – which is a valid formula modulo $\mathcal{T}_{\texttt{Reals_Ints}}$.

Example 6.1.1 (Counter) *Let us represent a counter as a transition system in quantifier-free linear integer arithmetic with a Boolean input a. The integer counter c is incremented if a is true, keeps its current value otherwise, and is initially 0. This can be represented as the transition system $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T\rangle$ where*

- Var *is $\langle a, c\rangle$ and $\mathcal{D}$ is $\{a \mapsto \texttt{Bool}, c \mapsto \texttt{Int}\}$, and states are written $s \triangleq \langle a, c\rangle$;*
- *the initial state predicate is $I(s) \triangleq (c = 0)$, and*
- *the transition relation is $T(s, s') \triangleq \big(a' \wedge (c' = c + 1)\big)\ \vee\ \big(\neg a' \wedge (c' = c)\big)$.*

Consider the formula in $\mathcal{L}^S$ corresponding to a trace of three states starting from an initial state:

$$I(s_0) \ \wedge \ T(s_0, s_1) \ \wedge \ T(s_1, s_2) \ \wedge \ T(s_2, s_3).$$

Models of this formula yield values for the unrolled states, such as $s_0 \triangleq \langle \mathbf{T}, 0 \rangle$, $s_1 \triangleq \langle \mathbf{T}, 1 \rangle$, $s_2 \triangleq \langle \mathbf{F}, 1 \rangle$ and $s_3 \triangleq \langle \mathbf{T}, 2 \rangle$. On the other hand the following trace of concrete states, understood as a structure, does not constitute a model: $s_0 \triangleq \langle \mathbf{T}, 0 \rangle$, $s_1 \triangleq \langle \mathbf{F}, 0 \rangle$, $s_2 \triangleq \langle \mathbf{T}, 0 \rangle$ and $s_3 \triangleq \langle \mathbf{T}, 2 \rangle$. Indeed, the transition between s_1 and s_2 is not legal ($T(s_1, s_2)$ evaluates to $\mathbf{F}$), and neither is the one between s_2 and s_3 for that matter.

Definition 6.1.3 (Trace, reachable state space) Given a transition system $\langle$ Var, $\mathcal{D}$, I, T $\rangle$, a *trace* is a vector of concrete states $\langle s_0, \ldots, s_n \rangle$ such that $\forall i \in [0, n-1]$, $T(s_i, s_{i+1})$ evaluates to $\mathbf{T}$. The trace is *initialized* if $I(s_0)$ evaluates to $\mathbf{T}$. A state is *reachable* if there exists an initialized trace leading to it. The *reachable state space* of a transition system is the set of all its reachable states. The *state space* on the other hand is the Cartesian product of the carriers of the sorts of all the state variables.

Definition 6.1.4 (Invariants) Given

- a transition system $S \triangleq \langle \langle v^1, \ldots, v^m \rangle, \mathcal{D}, I, T \rangle$ in a logic $\mathcal{L}$ built on a signature Σ and
- a well-sorted Σ-formula $\mathcal{I}$ in the environment $\{v^1 : \mathcal{D}(v^1), \ldots, v^m : \mathcal{D}(v^m)\}$,

we say that $\mathcal{I}$ is an *invariant* for S, or *holds* on S if and only if for all reachable state s, $\neg\mathcal{I}(s)$ is unsatisfiable modulo theory in $\mathcal{L}^S$. Otherwise $\mathcal{I}$ is *falsifiable* in S: a reachable state *falsifying* $\mathcal{I}$ exists.

Definition 6.1.5 (Reachability analysis) Given

- a transition system $S \triangleq \langle \langle v^1, \ldots, v^m \rangle, \mathcal{D}, I, T \rangle$ in a logic $\mathcal{L}$ built on a signature Σ and
- a well-sorted Σ-formula $\mathcal{R}$ in the environment $\{v^1 : \mathcal{D}(v^1), \ldots, v^m : \mathcal{D}(v^m)\}$.

A *reachability analysis* of $\mathcal{R}$ on S is an analysis answering "true" if a concrete state s such that $\mathcal{R}(s)$ exists and "false" otherwise.
Alternatively, let PO be a well-sorted Σ-formula in the environment $\{v^1 : \mathcal{D}(v^1), \ldots, v^m : \mathcal{D}(v^m)\}$. If a reachability analysis of $\neg$PO yields the answer "false", then PO is an invariant on S.

Definition 6.1.6 (Image, pre-image) Given a transition system $\langle$Var, $\mathcal{D}, I, T\rangle$, let ϕ be a well-sorted Σ-formula in the environment $\{v^1 : \mathcal{D}(v^1), \ldots, v^m : \mathcal{D}(v^m)\}$. The *image* of ϕ by the

transition relation is the set of all the concrete values for s_1 that can be extracted from models of the formula

$$\textbf{let } v' = s_1 \textbf{ in } \exists v,\ \phi(v) \wedge T(v, v').$$

That is, all the states that can be reached in one transition from a state such that $\phi(v)$. The *pre-image* of ϕ are all the concrete values for s_0 that can be extracted from models of the formula

$$\textbf{let } v = s_0 \textbf{ in } \exists v'\, T(v, v') \wedge \phi(v').$$

That is, all the states from which a state such that $\phi(v')$ can be reached in one transition.

Example 6.1.2 (Counter) *Recall that in AUFLIRA, sort* `Bool` *is necessarily interpreted as* $\{\mathbf{T}, \mathbf{F}\}$ *and* `Int` *is necessarily interpreted as the set of integers. So, in the transition system described in Example 6.1.1, the state space is* $\{\langle s^a, s^c\rangle \mid s^a \in \{\mathbf{T}, \mathbf{F}\}$ *and* s^c *is an integer*$\}$.
It is easy to see that the reachable state space is $\{\langle s^a, s^c\rangle \mid s^a \in \{\mathbf{T}, \mathbf{F}\}$ *and* $s^c \geq 0\}$. *Therefore,* $c \geq 0$ *is an invariant for this system. On the other hand, PO* $\triangleq c \leq 1$ *is falsifiable. State* $\langle true, 2\rangle$ *for instance violates PO, and is reachable – in particular* via *the initialized trace* $\langle true, 0\rangle$, $\langle true, 1\rangle$, $\langle true, 2\rangle$.

Consider now formula $\phi \triangleq c = -2$. *The image of* ϕ *are all the states reachable in one transition from states satisfying* ϕ*; a possible characterization of those states is* $(c = -2) \vee (c = -1)$. *The pre-image of* ϕ *are all the states that can lead to states verifying* ϕ *in one transition, for instance* $(c = -3) \vee (c = -2)$, *or* $(-3 \leq c) \wedge (c \leq -2)$. *Note that the counter system has no reachable state verifying* ϕ *but that its image and pre-image are defined nonetheless.*

6.2 Bounded Model Checking and k-induction

Like proof systems, verification techniques can be sound or complete with respect to a logic. Generally, formal methods are sound, *i.e.* are correct when they conclude that a proof objective holds for a transition system. Fewer are complete, in that it can be the case that they cannot find the answer to the problem at hand. Sound and complete techniques are decision procedures for verification problems. Unfortunately, there is no decision procedure for this problem in general.

There are however sound and complete decision procedure for the problem of verification on transition systems having a finite reachable state space, or *finite* transition systems. A naïve approach is to represent all reachable states and check that none falsify the proof objective. Since there is a finite number of reachable states, the approach eventually terminates either when a reachable state falsifying the PO is found, or when they have all been verified.

Doing so is very costly and calls for efficient, often symbolic or abstract representations of the state space *via* clever data structures, typically *Binary Decision Diagrams* (BDD [54]). This approach enjoyed many a success in particular in the field of hardware verification until the late nineties. Still, the increasingly complex systems developed today have such huge state spaces that, despite years of research and expert engineering, these techniques fail to scale on many complex systems. For this reason the verification community turned to SAT, and later to SMT, for a more tractable symbolic representation of sets of states.

The techniques presented in this thesis are based on k-induction, a powerful verification technique built on SMT solvers to prove or falsify safety properties on transition systems. We first present Bounded Model Checking (BMC) in Section 6.2.1. BMC relies on SMT solving to answer the series of questions "is a state falsifying the PO reachable in k transitions from the initial states?" starting from $k = 0$. We then discuss k-induction in Section 6.2.2: an induction proof consists in verifying that the answer to the two following questions is "no": *(i)* "do the initial states falsify PO?", and *(ii)* "from a state verifying PO, can a state falsifying it be reached in one transition?". K-induction is the generalization of induction to a trace of k states, as we shall see.

6.2.1 Bounded Model Checking

Bounded Model Checking (BMC) consists in using the power of SMT solvers to explore the reachable state space up to a given depth. Given a transition system $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ and a satisfiable proof objective PO, the formula

$$I(s_0) \ \wedge \bigwedge_{i \in [0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1}) \ \wedge \bigvee_{i \in [0,k]} \neg \mathrm{PO}(s_\mathbf{i}) \tag{6.1}$$

is unsat if and only if there is no initialized trace of k states in which PO is falsified.

Incremental BMC consists in iterating on the value of k, starting from 0. It can be interesting to rewrite (6.1) by removing parts redundant to previous iterations:

$$BMC(k) \equiv I(s_0) \ \wedge \bigwedge_{i \in [0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1}) \ \wedge \ \neg \mathrm{PO}(s_\mathbf{k}). \tag{6.2}$$

First, $BMC(0)$ checks whether the initial states can falsify the proof objective. Then $BMC(1)$ checks if any state one transition away from the initial states falsifies PO, and so on. If $BMC(k)$ is sat, then a model can be extracted and constitutes a counterexample, *i.e.* an initialized trace reaching a violation of the proof objective.

Unsat core analysis. Unfortunately, the scheme presented above does not terminate on a valid proof objective, even on finite state systems. Indeed, there are two explanations to an

unsatisfiability result of (6.2): either there is no violation of PO k transitions away from the initial states, or the system has no state at all k transitions away from the initial states. More formally, the second explanation corresponds to

$$I(s_0) \ \wedge \bigwedge_{i\in[0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1}) \tag{6.3}$$

being unsat on its own. Instead of checking the satisfiability of (6.3) at each iteration, we ask the SMT solver for an unsat core (*cf.* Section 5.3) whenever (6.2) is unsat. Conjuncts of (6.2) should be asserted separately and named: $I(s_0)$ is named `init`, $\bigwedge_{i\in[0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1})$ is named `transitions` and $\neg\mathrm{PO}(s_\mathbf{k})$ is named `notPo`. Now, assume (6.2) is unsat, and that the solver returns $\mathcal{UC} \triangleq \{$`init`, `transitions`, `notPo`$\}$ when asked for an unsat core. Any strict subset of $\mathcal{UC}$ is sat, so in particular

$$\overbrace{\{I(s_0)\}}^{\texttt{init}} \wedge \overbrace{\bigwedge_{i\in[0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1})}^{\texttt{transitions}}$$

is sat: there exists an initialized trace of $k+1$ states in this system. The fact that (6.2) is unsat hence comes from the constraint $\neg\mathrm{PO}(s_\mathbf{k})$ on the final state.
Now, assume (6.2) is still unsat, but that the solver returns the unsat core $\mathcal{UC} \triangleq \{$ `init`, `transitions` $\}$. Since by definition $\bigwedge \mathcal{UC}$ is unsat, no initialized trace of length $k+1$ exists for this system.

Incremental BMC with unsat core analysis can detect that the analysis has gone beyond the maximal length of the initialized traces of the system, which is enough to conclude that the proof objective holds. This is not enough for the technique to terminate on any finite state system, as illustrated on the following example.

Example 6.2.1 (Double Counter) *The* double counter *is the transition system parameterized by two integer values* $\mathbf{n_x}$ *and* $\mathbf{n_y}$ *such that* $0 \leq \mathbf{n_y} \leq \mathbf{n_x}$ *holds. It is defined as*

$$S_{DC}(\mathbf{n_x}, \mathbf{n_y}) \triangleq \langle\langle a, b, c, x, y\rangle,\ \texttt{Bool} \times \texttt{Bool} \times \texttt{Bool} \times \texttt{Int} \times \texttt{Int},\ I,\ T\rangle$$

where a, b and c are inputs, and

$$I(s) \triangleq (x = 0) \wedge (y = 0)$$

$$T(s, s') \triangleq \left(x' = \begin{cases} \textbf{\textit{if}}\ (b' \vee c') & 0 \\ \textbf{\textit{else if}}\ (a' \ \wedge\ x < \mathbf{n_x}) & x+1 \\ \textbf{\textit{else}} & x \end{cases}\right) \wedge \left(y' = \begin{cases} \textbf{\textit{if}}\ (c') & 0 \\ \textbf{\textit{else if}}\ (a' \ \wedge\ y < \mathbf{n_y}) & y+1 \\ \textbf{\textit{else}} & y \end{cases}\right).$$

Figure 6.1 shows the reachable state space of $S_{DC}(4, 2)$. Variables x and y are counters which are increased at the same time, when a is true, and saturate at $\mathbf{n_x}$ and $\mathbf{n_y}$ respectively. Notice that if

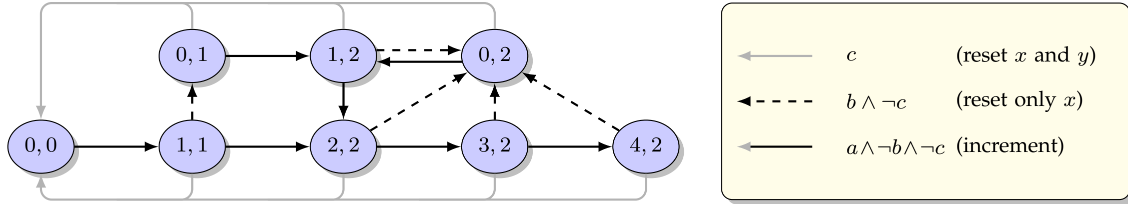


Figure 6.1: Reachable state space of $S_{DC}(4, 2)$, nodes show the values of x and y.

y is reset, then so is x, but the converse is not true. Transitions preserving the current state are not represented, and neither are those resetting x but incrementing y, i.e. *when a and b are true and c is false.*

An incremental BMC run on the double counter system would never end, since it is possible to construct initialized traces of arbitrary length, despite the fact that its reachable state space is finite. Indeed, the system can stay in the state $(0, 0)$ for example as long as it wants. It can also change states while looping forever, for instance on the states $(0, 0)$, increment to $(1, 1)$, reset back to $(0, 0)$ *etc.*

Loop-free path constraint. Incremental BMC can be modified to avoid this problem by enforcing the *loop-free path constraint*, *i.e.* requesting that all states of the trace should be different:

$$BMC(k) \equiv I(s_{\mathbf{0}}) \;\wedge \bigwedge_{i\in[0,k-1]} T(s_{\mathbf{i}}, s_{\mathbf{i+1}}) \;\wedge\; \neg\mathrm{PO}(s_{\mathbf{k}}) \;\wedge \bigwedge_{0\leq i<j\leq k} s_{\mathbf{i}} \neq s_{\mathbf{j}}.$$

Definition 6.2.1 (Reocurrence diameter) The *reocurrence diameter* of a system is the (possibly infinite) length of its longest loop-free initialized trace. Notice that all *finite transition systems*, *i.e.* systems having a finite reachable state space, have a finite reocurrence diameter.

Property 6.2.1 Incremental BMC with the loop-free path constraint is a decision procedure for reachability analysis of finite transition systems.

PROOF Any state of a transition system is accessible *via* a loop-free initialized trace. So, either

- the proof objective is false in some state n transitions away from the initial states, the technique will find it during iteration n and conclude correctly; or
- the proof objective holds and incremental BMC will conclude correctly once it will go above the finite reocurrence diameter, which it will notice thanks to the unsat core analysis, without finding any counterexample.

It is thus a decision procedure for this problem. ■

Notice that incremental BMC is also *model complete* on **any** transition system: if the proof objective does not hold in some state s reachable in n transitions from the initial state, then the technique will necessarily find it on iteration n and conclude correctly.

Incremental BMC can be further improved, notably by using incremental SMT solving to keep knowledge acquired during previous iterations. Still, it is very sensitive to the explosion of the reachable state space and does not scale up on real world problems. The following technique is based on induction and is a more realistic approach to formal verification.

6.2.2 K-induction

Induction. Informally, a *induction* proof for a proof objective PO on a transition system consists in showing that PO holds for the initial states, and is preserved by the transition relation of the system: from a state in which PO holds, taking a transition can only lead to states in which PO also holds. It follows directly that PO holds for all reachable states of the system. More formally, consider a transition system $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ and a predicate $\text{PO}(s)$ on its state variables. An induction proof consists in showing that

$$Base_V \triangleq I(s_0) \Rightarrow \text{PO}(s_0) \quad \text{and} \quad Step_V \triangleq (\text{PO}(s_0) \wedge T(s_0, s_1)) \;\Rightarrow\; \text{PO}(s_1)$$

are valid in $\mathcal{L}^S$. But $Base_V$ and $Step_V$ are valid if and only if

$$\begin{array}{llllll} (\textit{base instance}) & Base & \triangleq & \neg(Base_V) & \triangleq & I(s_0) \wedge \neg\text{PO}(s_0), \text{ and} \\ (\textit{step instance}) & Step & \triangleq & \neg(Step_V) & \triangleq & \text{PO}(s_0) \wedge T(s_0, s_1) \wedge \neg\text{PO}(s_1) \end{array}$$

are unsatisfiable in $\mathcal{L}^S$. Intuitively, this means that "there is no state that is initial and falsifies PO" ($Base$) and "there is no state that verifies PO and leads in one transition to a state falsifying PO" ($Step$). As one would expect, induction relies on SMT solvers for the unsatisfiability checks.

If $Base$ turns out to be satisfiable, then PO does not hold: there is an initial state which falsifies it, except in trivial cases where I is unsatisfiable. On the other hand, if $Step$ is satisfiable, nothing can be concluded – except that PO is not inductive. Indeed, the counterexample state s_0 (from which a falsification of PO follows) might not be a reachable state, in which case it is called a *spurious* counterexample. As a consequence, induction is a sound proof technique for the reachability analysis of transition systems, but it is not complete.

It is rather easy to see on the double counter system from Example 6.2.1 that variables x and y take their values in the intervals $[0, n_x]$ and $[0, n_y]$ respectively in any states the double counter can reach. More precisely,

$$Bounds(s) \triangleq (0 \leq x \leq \mathbf{n_x}) \wedge (0 \leq y \leq \mathbf{n_y})$$

is an invariant for S_{DC}. It is also inductive: as in the initial state x and y are both 0, $Bounds$ cannot be falsified. The base instance is unsat. Also, it is not possible in one transition to escape the intervals, so the step instance is unsat too.

Now, consider the proof objective $\text{PO}(s) \triangleq (x = \mathbf{n_x}) \implies (y = \mathbf{n_y})$: "if x is saturated, then so is y". Not only is it harder to see that PO is an invariant, it is also not inductive. The base instance is unsat since $x \neq n_x$: the initial state does not falsify PO. The step instance on the other hand is satisfiable. From a *root state* $s_\mathbf{0}$ such as $x = n_x - 1$ and $y = n_y - 2$ ($\text{PO}(s_\mathbf{0})$ holds) it is indeed possible to reach a state $s_\mathbf{1}$ such that $x = n_x$ and $y = n_y - 1$ in one transition by incrementing x and y; $\text{PO}(s_\mathbf{1})$ is false and $\langle s_\mathbf{0}, s_\mathbf{1} \rangle$ a model for the step instance, a counterexample. But it is a spurious counterexample as no state such that $x = n_x - 1$ and $y \triangleq n_y - 2$ is reachable.

Strengthening. It is possible to rule out this spurious counterexample by *strengthening* PO to

$$((x = \mathbf{n_x}) \implies (y = \mathbf{n_y})) \wedge \neg(x = \mathbf{n_x} - 1 \ \wedge \ y = \mathbf{n_y} - 2).$$

This strengthened PO blocks the root state of the counterexample: it is not eligible as a model for the step instance anymore. The step instance is still be sat though: $s_\mathbf{0}$ such that $x = \mathbf{n_x} - 1$ and $y = \mathbf{n_y} - 3$ leads to a violation of the strengthened proof objective, more precisely of $\neg(x = \mathbf{n_x} - 1 \ \wedge \ y = \mathbf{n_y} - 2)$. This second counter example can be ruled out in the same way, but this is a never-ending task. We know that $Bounds$ is an invariant for the double counter system. So, we can strengthen PO by changing it to

$$\Big((x = \mathbf{n_x}) \implies (y = \mathbf{n_y})\Big) \wedge (0 \leq x \leq \mathbf{n_x}) \wedge (0 \leq y \leq \mathbf{n_y})$$

With this strengthened proof objective, there is only a finite number of spurious counterexamples to block for given values of $\mathbf{n_x}$ and $\mathbf{n_y}$; *i.e.* all the states such that $x = \mathbf{n_x} - 1$ and $y = \mathbf{n_y} - 2$, or $y = \mathbf{n_y} - 3 \ldots$ or $y = \mathbf{n_y} - \mathbf{n_y} = 0$. Indeed, a state $s_\mathbf{0}$ such that $y = -1$ is blocked by the strengthened proof objective, since $y < 0$ and thus is not a model for

$$Step \triangleq \text{PO}(s_\mathbf{0}) \wedge T(s_\mathbf{0}, s_\mathbf{1}) \wedge \neg\text{PO}(s_\mathbf{1})$$

because $\text{PO}(s_\mathbf{0})$ is false. As a result, once PO is strengthened to

$$\begin{array}{ll} & ((x = \mathbf{n_x}) \implies (y = \mathbf{n_y})) \ \wedge \ (0 \leq x \leq \mathbf{n_x}) \ \wedge \ (0 \leq y \leq \mathbf{n_y}) \\ \wedge & \neg(x = \mathbf{n_x} - 1 \ \wedge \ y = \mathbf{n_y} - 2) \ \wedge \ \ldots \ \wedge \ \neg(x = \mathbf{n_x} - 1 \ \wedge \ y = \mathbf{n_y} - \mathbf{n_y}) \end{array}$$

it is an inductive invariant for the double counter system. This enumerative strategy of blocking counterexamples one by one is not realistic as in general there can be an too many spurious counterexamples to block, or even an infinity of them on infinite transition systems.

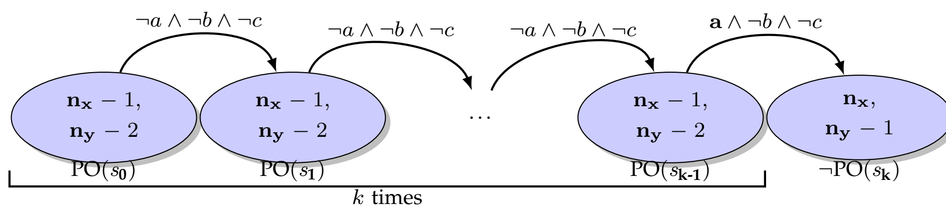
Definition 6.2.2 (K-induction) *K-induction* is the generalization of induction to k transitions. Given a transition system $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ and a predicate $\mathrm{PO}(s)$ over the state variables of S, a k-induction proof consists in showing that two formulas are unsatisfiable:

- $Base_k$, which is unsatisfiable if and only if there is no state falsifying PO reachable from the initial states in $k-1$ transitions or less, and
- $Step_k$ is unsatisfiable if and only if any trace of $k-1$ states verifying PO cannot lead to a state falsifying it.

Note that 1-induction is the same as induction, and that the base instance is the same as BMC. $Base_k$ and $Step_k$ are formally defined as

$$\begin{array}{lcl} Base_k(I, T, \mathrm{PO}) & \equiv & \overbrace{I(s_{\mathbf{0}})}^{\text{Initial state}} \wedge \overbrace{\bigwedge_{i \in [0,k-2]} T(s_{\mathbf{i}}, s_{\mathbf{i+1}})}^{\text{trace of k-1 transitions}} \wedge \overbrace{\bigvee_{i \in [0,k-1]} \neg\mathrm{PO}(s_{\mathbf{i}})}^{\text{PO falsified on some state}} \\ Step_k(T, \mathrm{PO}) & \equiv & \underbrace{\bigwedge_{i \in [0,k-1]} T(s_{\mathbf{i}}, s_{\mathbf{i+1}})}_{\text{trace of } k \text{ transitions}} \wedge \underbrace{\bigwedge_{i \in [0,k-1]} \mathrm{PO}(s_{\mathbf{i}})}_{\text{PO satisfied on first } k \text{ states}} \wedge \underbrace{\neg\mathrm{PO}(s_{\mathbf{k}})}_{\text{PO falsified by last state}} . \end{array}$$

The idea behind k-induction is to try to block spurious counterexamples: incrementing the k further is a form of property strengthening. The step instance requires a trace of k states verifying the proof objective before a transition leading to its falsification is taken. Sadly, this technique by itself does not block, say, states such that $x = \mathbf{n_x} - 1$ and $y = \mathbf{n_y} - 2$. Indeed, the system can always maintain the current state if the inputs of the system are all false. Thus, the trace

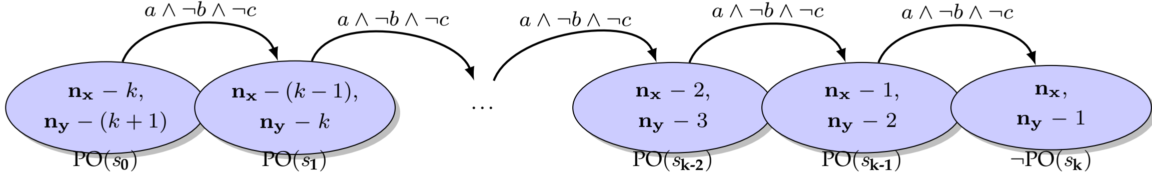


is a spurious counterexample for k-induction on the double counter system, for any k.

Loop-free path constraint. In practice most k-induction implementations enforce the loop-free path constraint introduced above in the context of BMC. It consists in forcing the states to be pairwise different by adding to the base and step instance the constraints $s_\mathbf{i} \neq s_\mathbf{j}$ for all i and j such that $0 \leq i < j \leq k$. With the loop-free path constraint, the strengthened version of the proof objective described above, *i.e.*

$$((x = \mathbf{n_x}) \implies (y = \mathbf{n_y})) \wedge (0 \leq x \leq \mathbf{n_x}) \wedge (0 \leq y \leq \mathbf{n_y}),$$

is $\mathbf{n_y}$ inductive. For any $k < \mathbf{n_y}$ though, the following trace is a spurious counterexample



But for $\mathbf{n_y}$-induction, $s_\mathbf{0}$ would be such that $x = \mathbf{n_x} - k$ and $y = \mathbf{n_y} - (k+1) = -1$. Since y is negative, PO($s_\mathbf{0}$) is false and the trace is not a model for the step instance.

Incremental k-induction. K-induction-based verification tools all follow the same approach. When given a transition system and a proof objective PO, they try to prove it by (1-)induction. If both the base and step instances are unsat, PO is an (inductive) invariant. If the base instance is satisfiable, then PO does not hold in all states of the system since an initial state falsifies it.

If the step instance is satisfiable however, nothing can be concluded. The tool then moves on to 2-induction and can either prove that PO is 2-inductive, find a counterexample state one transition away from an initial state, or fail to conclude (step is satisfiable). The tools then tries a 3-induction proof and so on. Note that as was the case for incremental BMC, the base instance can be simplified to

$$Base_k(I, T, \text{PO}) \equiv \overbrace{I(s_\mathbf{0})}^{\text{Initial state}} \wedge \overbrace{\bigwedge_{i \in [0,k-2]} T(s_\mathbf{i}, s_\mathbf{i+1})}^{\text{trace of k-1 transitions}} \wedge \overbrace{\neg\text{PO}(s_\mathbf{k\text{-}1})}^{\text{PO falsified in the last state}} .$$

Also similarly to BMC, an unsat core analysis of the base instance when it is unsat detects that the reocurrence diameter of the system has been reached, and thus that all states have been examined.

Property 6.2.2 Incremental K-induction with the loop-free path constraint and unsat core base instance analysis is a decision procedure for reachability analysis of **finite** transition systems.

PROOF The base instance of incremental k-induction is the same as incremental BMC. The step instance adds a check for k-inductiveness, allowing the technique to conclude earlier when it finds a k-induction proof. Since k-induction is sound, *i.e.* only proves actual invariants, and BMC is a decision procedure for finite transition system (Proposition 6.2.1) incremental k-induction is also a decision procedure for finite transition systems. ∎

Multi-PO incremental k-induction. We generalize incremental k-induction to the analysis of a set of proof objectives $S_{\text{PO}} \triangleq \{\text{PO}_1, \ldots, \text{PO}_n\}$ computing three maximal subsets of S_{PO}: V, F and U. The step and base instances are thus

$$\begin{aligned}
Base_k(I, T, \text{PO}) &\equiv \overbrace{I(s_0)}^{\text{Initial state}} \wedge \overbrace{\bigwedge_{i\in[0,k-2]} T(s_\mathbf{i}, s_\mathbf{i+1})}^{\text{trace of k-1 transitions}} \wedge \overbrace{\bigvee_{j\in[1,n]} \neg\text{PO}_j(s_\mathbf{k})}^{\text{one of the } \text{PO}_j \text{ falsified in the last state}} \\
Step_k(T, \text{PO}) &\equiv \underbrace{\bigwedge_{i\in[0,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1})}_{\text{trace of } k \text{ transitions}} \wedge \underbrace{\bigwedge_{i\in[0,k-1]} \bigwedge_{j\in[1,n]} \text{PO}_j(s_\mathbf{i})}_{\text{all } \text{PO}_j \text{ satisfied on first } k \text{ states}} \wedge \underbrace{\bigvee_{j\in[1,n]} \neg\text{PO}_j(s_\mathbf{k})}_{\text{one of the } \text{PO}_j \text{ falsified in the last state}} .
\end{aligned}$$

If the base instance is satisfiable a concrete counterexample trace exists for some of the proof objectives. It is possible to know which by evaluating $\neg\text{PO}_j(s_\mathbf{k})$ for $j \in [1, n]$ in the model returned by the SMT solver. We remove them from S_{PO} and put them in a set F (falsified) before attempting a new k-induction proof. Assume now that the step instance is satisfiable. The same scheme lets us identify the PO_j for which the step instance does not hold. We put those in a set U (unknown). If both base and step are unsat, then all the PO_i are **mutually** inductive, and constitute the set V of valid proof objective. Notice that F, U and V are maximal disjoint subsets of S_{PO}.

6.3 Invariant Discovery and Other Techniques

In practice though, transition systems stemming from realistic designs have a huge reachable state space. Incremental k-induction does not scale on such problems because of the combinatorial explosion: in the worst case it needs to *unroll* the transition relation as far as the reocurrence diameter of the system, which might be infinite. Despite their high efficiency, SMT solvers cannot keep up and the analysis exhausts resources or time. Still, for most of the real world verification problems it is possible to strengthen the proof objective to make it k-inductive for a relatively small k by discovering extra invariants of the system, which better characterize the reachable state space. For the double counter system for instance, PO can be strengthened to

$$((x = \mathbf{n_x}) \implies (y = \mathbf{n_y})) \wedge (0 \leq x \leq \mathbf{n_x}) \wedge (0 \leq y \leq \mathbf{n_y}) \wedge (x \leq y + \mathbf{n_x} - \mathbf{n_y})$$

which is an inductive invariant implying the original proof objective. Note that $(x \leq y + \mathbf{n_x} - \mathbf{n_y})$ is a *relational* invariant: it crystallizes a relation between two state variables. Relational invariants are very valuable but often tedious to discover by hand, and even harder to discover automatically. In the following we discuss techniques for *invariant discovery*, generally by abstracting away details about the transition relation to avoid the combinatorial explosion of the reachable state space and discover invariants.

Abstract Interpretation (AI [23]) is a technique working on an abstraction of the semantics of the system it analyzes. While there are many formalisms to express the concrete semantics of a system, it is in general not computable. AI works on abstract (computable) semantics, in a sound way: if an invariant for the abstract semantics is found, then it is also an invariant for the concrete semantics. The converse is not true though, so it is not a complete technique. AI uses *abstract domains* to construct over approximations of the reachable state space. Abstract domains are in fact *complete lattices*, a special kind of *posets*.

Definition 6.3.1 (Poset) A *poset*, or *partially ordered set*, is a pair $\langle S, \preceq \rangle$ where S is a set and $\preceq$ is an ordering relation, *i.e.* it is

- reflexive: for all $x \in S$, $x \preceq x$;
- anti-symmetric: for all $x, y \in S$, if $x \preceq y$ and $y \preceq x$ then $x = y$;
- transitive: for all $x, y, z \in S$, if $x \preceq y$ and $y \preceq z$ then $x \preceq z$.

Note that $\preceq$ is not necessarily total: there is no requirement that for all $x, y \in S$, either $x \preceq y$ or $y \preceq x$.

Definition 6.3.2 (Lattice) A poset $\mathcal{P} \triangleq \langle S, \preceq \rangle$ is a *lattice* if every two elements e_1 and e_2 of S have a *least upper bound* (or *join*) and a *greatest lower bound* (or *meet*).
A *complete lattice* is a lattice $\mathcal{P} \triangleq \langle S, \preceq \rangle$ in which any subset of S has a least upper bound and a greatest lower bound.

Elements of complete lattices are used to represent sets of states in the concrete semantics of transition systems, the idea being to over approximate the reachable state space of a given system.

Definition 6.3.3 (Concrete domain) Let $\mathcal{U}$ be the state space (universe) of a transition system S. We define the *concrete domain* of S as the lattice $\langle \mathcal{P}(\mathcal{U}), \subseteq \rangle$, where $\mathcal{P}(\mathcal{U})$ is the *power set* of $\mathcal{U}$ and $\subseteq$ is the usual set inclusion relation.

Definition 6.3.4 (Concretization function) Given the lattice of concrete values of a program $\mathcal{D}$ and an abstract domain $\mathcal{D}^\sharp$ with partial order $\sqsubseteq$, a *concretization function* is a function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ that is monotone, *i.e.*

$$\text{for all } x^\sharp, y^\sharp \in \mathcal{D}^\sharp,\ x^\sharp \sqsubseteq y^\sharp \Rightarrow \gamma(x^\sharp) \subseteq \gamma(y^\sharp).$$

Informally, if $x^\sharp$ is a smaller abstract value than $y^\sharp$, then the concretization of $x^\sharp$ characterizes fewer states than that of $y^\sharp$.

AI does not consider the semantics of the language used to express the system but instead defines abstract semantics consistent with the abstract domain chosen. The concrete semantics of a function application $\phi(t_1, \ldots, t_n)$ of the language is a function $[\![\phi(t_1, \ldots, t_n)]\!] : \mathcal{D} \to \mathcal{D}$. AI defines abstract semantics, *i.e.* a function $[\![\phi(t_1, \ldots, t_n)]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$. The abstract semantics must be sound with respect to $\mathcal{D}$: for all function symbol ϕ and $x^\sharp \in \mathcal{D}^\sharp$,

$$[\![\phi(t_1, \ldots, t_n)]\!](\gamma(x^\sharp)) \subseteq \gamma([\![\phi(t_1, \ldots, t_n)]\!]^\sharp(x^\sharp)).$$

Informally, this means that no states are lost by the abstract semantics. It is then possible to define the *abstract transition function* $[\![T]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ from the (concrete) transition relation of the system T. $[\![T]\!]^\sharp$ is such that for all $x^\sharp \in \mathcal{D}^\sharp$:

$$\gamma([\![T]\!]^\sharp(x^\sharp)) \ \subseteq\ \gamma(x^\sharp) \cup \{s',\ (\exists s \in \gamma(x^\sharp),\ T(s, s'))\} \tag{6.4}$$

This implies that $[\![T]\!]^\sharp$ is order-preserving, *i.e.* for all $x^\sharp, y^\sharp \in \mathcal{D}^\sharp$

$$x^\sharp \sqsubseteq y^\sharp \ \Rightarrow\ [\![T]\!]^\sharp(x^\sharp) \sqsubseteq [\![T]\!]^\sharp(y^\sharp)$$

In this context, the Knaster-Tarski theorem [51, 74] guarantees the existence of a least fixed point *lfp* for $T^\sharp$ in $\mathcal{D}^\sharp$. Concretely, *lfp* is the smallest – with respect to $\sqsubseteq$ – $x^\sharp \in \mathcal{D}^\sharp$ such that $T^\sharp(x^\sharp) = x^\sharp$. Additionally, *lfp* is computable by an iterative process called fixed point iteration. A predicate $\mathcal{I}(s)$ on the state variables of the system is extracted from *lfp*, characterizing exactly $\gamma(\mathit{lfp})$. $\mathcal{I}$ is such that

$$\mathcal{I}(\mathbf{s_0}) \wedge T(\mathbf{s_0}, \mathbf{s_1}) \Rightarrow \mathcal{I}(\mathbf{s_1}),$$

that is to say the step instance of an induction proof of $\mathcal{I}$ is unsat. Indeed, $\mathcal{I}$ is the concretization of a fixed point by the abstract transition function, with over approximates the concrete transition relation (*cf.* Equation 6.4). Since the concretization function is monotone, $\mathcal{I}$ is preserved by the transition relation.

Note that we did not take into account the initial states, and thus it is not certain at this point that the $\mathcal{I}$ is indeed an invariant for the system at hand. In practice this is taken care

of at the beginning of the fixed point iteration.

Unfortunately, AI has so many parameters to control the transition relation abstraction, the abstract domain and the fixed point iteration that it requires expert tuning to work on complex systems, often accompanied by a deep understanding of said systems. AI implementations include *(i)* Astrée [10] which analyzes C code, and *(ii)* NBac [43] which analyzes properties of Lustre models by using a combination of forward and backward fixed point computation. We now present the two predominant automatic SMT-based techniques for SMT-based invariant discovery.

Interpolation-based Model Checking (IMC) as described by McMillan in [55] is an SMT-based technique. The idea is to discover an inductive invariant $\mathcal{I}$ entailing that a violation of the proof objective PO cannot be reached in k transition. Hence, $\mathcal{I} \land \text{PO}$ is k-inductive. This approach uses *interpolation* extensively.

Definition 6.3.5 (Interpolation) Given two formulas A and B such that $A \land B$ is unsat, *interpolation* consists in producing an *interpolant* for A and B, *i.e.* a formula P such that

- $A \models P$ (any model of A is a model of P),
- $P \land B$ is unsat,
- the free function symbols appearing in P are among the ones appearing in both A and B.

Informally, an interpolant of A and B, with $A \land B$ unsatisfiable, is a consequence of A which is also unsat with B and only characterizes what is common between A and B. Also, an interpolant can always be derived from an unsatisfiability proof [24].

Let $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ be a transition system. Assume that I is satisfiable and let the predicate R_0 be I. Consider the following formulas:

$$A_i \equiv R_i(s_\mathbf{0}) \;\land\; T(s_\mathbf{0}, s_\mathbf{1}) \quad \text{and} \quad B \equiv \overbrace{\bigwedge_{i \in [1,k-1]} T(s_\mathbf{i}, s_\mathbf{i+1}) \;\land\; \bigvee_{i \in [1,k]} \neg\text{PO}(s_\mathbf{i})}^{\text{trace of } k \text{ states some of which falsify PO}}.$$

Note that the BMC equation (6.1) is the same as $A_0 \land B \land \neg\text{PO}(s_\mathbf{0})$, and that A_i only mentions $\langle s_\mathbf{0}, s_\mathbf{1} \rangle$ while B only uses states $\langle s_\mathbf{1}, \ldots, s_\mathbf{k} \rangle$. The idea of IMC is to construct the series of predicate R_i: R_{i+1} is an over approximation of the states reachable in one transition from R_i.

First, if $I(s_\mathbf{0}) \land \neg\text{PO}(s_\mathbf{0})$ is satisfiable, then at least one of the initial states is a counterexample, so PO does not hold. Otherwise, the iteration on the series R_i begins. If $A_0 \land B$ is

satisfiable then an actual counterexample has been found, since $R_0 \equiv I$. If it is unsat on the other hand, let us compute the interpolant R_1 of A_0 and B. Now R only mentions the variables A_0 and B have in common: s_1. As $A_0 \models R_1$, the interpolant evaluates to true on all states reachable in one transition from the initial states, it is an over approximation of the image of the initial states. Last, $R_1 \wedge B$ is unsat, so R_1 characterizes states from which no violation of the proof objective can be reached in $k - 1$ transitions or less. A new iteration begins. If $A_1 \wedge B$ is satisfiable, nothing can be concluded as R_1 is an over approximation and potentially characterizes unreachable states; the technique fails. If it unsat, the R_2 is the interpolant of A_1 and B.

Every time an interpolant R_n is computed, it can be the case that $R_n \Rightarrow \bigvee_{i \in [0,n-1]}$: the new interpolant does not characterize states different from the ones seen before. A fixed point of the transition relation is reached and proves that $\bigvee_{i \in [0,n-1]}$ is an inductive invariant. It characterizes states from which ¬PO cannot been reached in $k - 1$ transitions or less, and thus proves that PO is also an invariant.

The series of R_i computed at each step of the incremental scheme is ever growing and can become impossible for SMT solvers to handle in practice because of its size. Also, incremental IMC suffers from the same drawback as incremental k-induction in that it needs to unroll the transition relation potentially many times. The abstraction mechanism speeds up the process, but unrolling the transition relation even only a few times might not be possible on real systems.

Property Directed Reachability (PDR, *a.k.a.* IC3 [12, 34]) does away with multiple unrolling. All the jobs sent to the SMT-solvers have at most one unrolling of the transition relation: the difficulty of the analysis is split into many "simpler" problems. Let $S \triangleq \langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ be a transition system. PDR maintains a sequence of *frames* F_i: F_0 is I, and for $i > 0$ F_i is an initially empty set of cubes such that for $j > 1$, $F_{j+1} \subseteq F_j$. We write $\mathcal{F}_i$ for the formula $\bigwedge_{c \in F_i} \neg c$ called the *formula of frame* i. The ith frame F_i contains cubes known to be unreachable in i transitions from the initial states. The formula $\mathcal{F}_i$ characterizes the states that are not in these cubes. It thus represents an over approximation of the states reachable in up to i transitions from the initial states. Since for $j > 1$, $F_{j+1} \subseteq F_j$, then $\mathcal{F}_j \Rightarrow \mathcal{F}_{j+1}$ holds. Following [34], the formal properties of the frame sequence of size $k + 1$ are:

(1) $\mathcal{F}_0 \equiv I$;

(2) F_i is a set of cubes initially empty for $i > 0$;

(3a) $\mathcal{F}_i \Rightarrow \mathcal{F}_{i+1}$ for $i \geq 0$;

(3b) $F_{i+1} \subseteq F_i$ for $i > 0$;

(4a) $\mathcal{F}_{i+1}$ is an over approximation of the image of $\mathcal{F}_i$, *i.e.* $(\mathcal{F}_i(s_0) \wedge T(s_0, s_1)) \Rightarrow \mathcal{F}_{i+1}(s_1)$;

(4b) $\mathcal{F}_i$ is an over approximation of the states reachable in i transitions from the initial states;

(5) $\mathcal{F}_i \Rightarrow$ PO except for the last frame $\mathcal{F}_k$.

Now, the algorithm constructs the frame sequence iteratively, starting from the sequence $[F_0]$. At iteration k the satisfiability of

$$\mathcal{F}_k(s_0) \wedge \neg\text{PO}(s_0) \tag{6.5}$$

is checked, "does frame k imply PO?".

$\mathcal{F}_0 \equiv I$ $\mathcal{F}_1 \Rightarrow$ PO $\cdots$ $\mathcal{F}_{k-1} \Rightarrow$ PO $\mathcal{F}_k \wedge \neg$PO?

$[\, F_0 \,,\, F_1 \,,\, \cdots \,,\, F_{k-1} \,,\, F_k \,]$

If it is satisfiable and if $k = 0$, then an initial state falsifying PO has been found. If $k > 0$, the counterexample might be spurious as $\mathcal{F}_k$ is an over approximation (4b). Let IPO$_k$ (intermediary proof objective) be the cube deriving from this counterexample on frame k. Now, is IPO$_k$ reachable from $\mathcal{F}_{k-1}$? If

$$\mathcal{F}_{k-1}(s_0) \wedge T(s_0, s_1) \wedge \text{IPO}_k(s_1) \tag{6.6}$$

is unsat, the answer is "no" because $\mathcal{F}_{k-1}$ blocks IPO$_k$ for frame k. We can add IPO$_k$ to frame F_i for $i \in [1, k]$ because of (3a) and the technique goes back to checking if frame k implies PO.

If $\mathcal{F}_{k-1}$ does not block IPO$_k$ – (6.6) is sat – then something new must be learnt at frame $k - 1$. Let IPO$_{k-1}$ be the cube derived from the state leading to IPO$_k$ found in (6.6). IPO$_k$ cannot be blocked for now and is memorized for when IPO$_{k-1}$ will be blocked.

$\mathcal{F}_0 \equiv I$ $\mathcal{F}_1 \Rightarrow$ PO $\cdots$ $\mathcal{F}_{k-1} \Rightarrow$ PO $\mathcal{F}_k \wedge \neg$PO?

$[\, F_0 \,,\, F_1 \,,\, \cdots \,,\, F_{k-1} \,,\, F_k \,]$

To block: IPO$_{k-1}$ ← To block: IPO$_k$

The blocking / learning process then proceeds recursively. As soon as IPO$_i$ is blocked for some i, the technique checks if IPO$_{i-1}$ is now blocked and so on. If at some point frame 0 is not able to block an intermediary proof objective, then PO does not hold and the series of IPO memorized is an initialized counterexample trace.

If (6.6) is unsatisfiable, frame k implies PO. Before PDR iterates by creating frame $k+1$ it launches a *propagation phase*. For i from 1 to k, this phase tries to propagate each cube c of frame i to frame $i+1$ by checking the satisfiability of

$$\mathcal{F}_i(s_0) \wedge T(s_0, s_1) \wedge c(s_1).$$

If it is sat then c is not blocked at frame $i+1$ and cannot be propagated, but if it is unsat then F_{i+1} is augmented with c. Assume now that $F_j = F_{j+1}$ for some j. This means that $\mathcal{F}_j$ is stable by the transition relation by (4a), and holds for the initial states because of (3a). Hence, $\mathcal{F}_j$ is an inductive invariant; but $\mathcal{F}_j \Rightarrow$ PO by (5), so PO $\wedge \mathcal{F}_j$ is also an inductive invariant. PO is successfully strengthened and the analysis is complete.

PDR as described above is highly inefficient: it blocks counterexample states one by one which is not realistic for complex Boolean systems or integer or rational-valued variables. Blocking generalizations of counterexamples results in a huge performance gain. Consider the satisfiability query corresponding to IPO$_{i+1}$:

$$\mathcal{F}_i(s_0) \wedge T(s_0, s_1) \wedge \text{IPO}_{i+1}(s_1).$$

Assume it is satisfiable. In [34], which discusses PDR in the purely propositional case, the author proposes to use ternary simulation to identify parts of the models relevant to its satisfiability to produce a generalized IPO$_i$. The cube encompasses more states, and quickens convergence. If the query is unsatisfiable on the other hand, it is interesting to ask for an unsat core: some parts of IPO$_{i+1}$ might not be relevant for the unsat result and a more general version of IPO$_{i+1}$ is added to F_{i+1}, blocking more bad states. Also, an IPO is a cube from which a violation of the proof objective can be reached; so, whenever one must is blocked at frame i we might as well forward it to frame $i+1$. We refer the reader interested in more optimizations and soundness arguments to [34].

PDR was first introduced as a purely propositional technique, but generalizations to SMT can be found in [40, 19] resulting in tools yielding very promising results. We present our own in Section 8.3.

6.4 Motivation and Outline of the Thesis

Discovering invariants on realistic systems is hard, in particular relational ones. Discovering property-directed invariants able to strengthen a proof objective is even harder, but is often mandatory for the verification of functional proof objectives on realistic systems. With the notable exception of Abstract Interpretation which is not property-directed, the fundamental reason why the methods presented above do not scale up to systems in our field of

verification of avionics functional chains is that they are unable to discover strengthening relational invariants. We claim that it is sometimes better to try to *guess* them, *i.e.* heuristically propose potential invariants and let a multi-PO k-induction engine such as the one presented in Subsection 6.2.2 set apart those it can prove.

The Kind model checker [47] follows this approach: templates from database are instantiated on state variables and constants appearing in the transition system [46]. A k-induction engine then tries to prove the proof objective(s) on the system thanks to the confirmed invariants. Unfortunately the potential invariants generated are not property directed: they have no guarantee to block states involved in the satisfiability of the step instance of the inductive reasoning. Also, the instantiation process on systems with many state variables and constants tends to flood the k-induction engine with potential invariants. Instantiation further diverges as the database of templates grows bigger.

Our contribution is twofold. First, we define a parallel architecture based on a k-induction engine allowing the cooperation of invariant discovery methods with k-induction. Second, we propose a property-directed potential invariant generation heuristic. We demonstrate its ability to discover relational invariants strengthening functional proof objectives on common design patterns in the field of avionics critical embedded systems. We now present the plan we will follow to discuss our work.

Part II describes our methodology. We begin by improving the quantifier elimination algorithm due to Monniaux in Chapter 7, both in terms of performance and in the expression power of the formulas it handles. We then introduce our collaborative proof engine architecture based on k-induction in Chapter 8 as well as two invariant discovery techniques taking advantage of the improved quantifier elimination algorithm. We then present HullQe (Chapter 9), a heuristic generating property-directed potential invariants combining quantifier elimination and convex hull computation. Then, in Part III, we expose the most interesting aspects of our implementation, called Stuff, of the architecture presented in Part II. Chapter 10 discusses parallelism and extensibility in Stuff, and shows that the semantics of transition systems defined in Section 6.1 are precisely respected in the framework. We then introduce Assumptio, our SMT solver wrapper used by all the techniques featured in Stuff. Last, we conclude on this study and discuss perspectives in Part IV.

PART II

Invariant Discovery Powered By Quantifier Elimination

This part of the thesis presents our main contribution: a parallel architecture for a k-induction based proof engine with cooperating invariant discovery methods. The methods we will propose rely heavily on Quantifier Elimination (QE). QE is a fundamental topic in mathematical logic, and has been the subject of ongoing research for several decades. We begin by reminding the major QE breakthroughs and present the most frequent and successful uses of QE for safety critical systems design and verification. We then present our main contributions. First, we propose extensions and improvements to the QE algorithm from [56] in Chapter 7. Then, we present our approach to the problem of verifying safety properties on transition systems in Chapter 8, and describe two invariant discovery methods implemented in our formal framework Stuff: BrutalIQe and a generalization of PDR to SMT using (partial) quantifier elimination. Last, we introduce HullQe, a heuristic for the generation of potential invariants and report on our experiments with the Rockwell Collins triplex voter system, and a reconfiguration logic system in Chapter 9.

Definition 6.4.1 (Quantifier elimination) Given a theory $\mathcal{T}$, a quantifier free $\mathcal{T}$-formula ϕ and a vector of variables V, Quantifier Elimination consists in producing a quantifier free $\mathcal{T}$-formula $\mathcal{O}$ such that $\mathcal{O}$ is equivalent in $\mathcal{T}$ to $\exists V \ \phi$. More$\mathcal{T}$- precisely, $\mathcal{O}$ has the same models as $\exists V \ \phi$ in $\mathcal{T}$, or equivalently

- any partial model of $\mathcal{O}$ in $\mathcal{T}$ can be extended to a model of ϕ in $\mathcal{T}$, and
- any model of ϕ in $\mathcal{T}$ is a model of $\mathcal{O}$ in $\mathcal{T}$.

We write this operation as $QE(V, \phi)$.

In 1948, Tarski proved the decidability of real closed fields by exhibiting a general quantifier elimination algorithm (published in 1951 [73]). By iteratively eliminating all variables of a system of polynomial constraints over real closed fields, a ground expression is eventually obtained, which can easily be evaluated to either true or false. However Tarski's algorithm

has a prohibitive computational cost, and remains unusable for practical purposes, even with today's digital computers. In 1975, Collins published a new general quantifier elimination method, *Cylindrical Algebraic Decomposition* (CAD) [21], with a drastically lower, yet still very high, computational complexity (2^{2^n} with n the number of variables to eliminate). The first CAD implementations appeared in the early eighties and have since then continuously been refined and enhanced [41] to reach a level sufficient to allow real world system verification applications to be considered. Another breakthrough was achieved in 1988 by Weispfenning and Loos, with the method of quantifier elimination by *Virtual Substitution* (VS), first for linear systems [52], and then for at most quadratic systems over reals [79, 80]. Even more recently, Gröbner bases have attracted attention [61, 81]. Today, implementations of these general non-linear QE algorithms are available in various software packages (qepcad, redlog, matlab, maple, *etc.*).

Concurrently to these theoretical developments, applications of QE for the analysis and verification of safety critical systems appeared as well. If we consider transition systems $\langle \mathsf{Var}, \mathcal{D}, I, T \rangle$ stemming from anything from hybrid automata to software controllers, a well studied approach is that of reachability analysis by pre-image computation of sets of states through the transition relation. In that case QE is used to iteratively construct the series G_i such that $G_i(s) \triangleq QE(s',\ F(s) \wedge T(s, s') \wedge G_{i-1}(s'))$, starting from a $G_0 \equiv \neg$PO corresponding to a formula representing unsafe states violating a proof objective PO, and where F is a formula "guiding" the backwards traversal in some way (usually environment constraints on the system inputs). Here, QE is used to eliminate next state variables s' from a formula representing a transition leading to the states G_{i-1}. So G_i is the pre-image of G_{i-1} through the transition relation. Each pre-image is checked against initial states to detect counterexammples, and a fixed point detection mechanism in the pre-image computation terminates the analysis if all preceeding states have been found [1].

Another well studied approach is that of template-based invariant discovery. A template $\phi(u, x)$ representing a potential invariant with parameters u is instantiated on state variables x and next state variables x'. The formula $QE(u, I(x) \implies \phi(u, x) \wedge x, x', \phi(u, x) \wedge T(x, x') \implies \phi(u, x'))$ states that the template is an invariant of the transition system, *i.e.* that some values exist for parameters u such that initial states all satisfy the template, and such that the template is preserved by the transition relation. QE is used to eliminate state variables x and x', and derive conditions on u making $\phi(u, x)$ inductive [50]. Template-based approaches are also used for controller synthesis [72]. Conditions over template parameters ensuring safe mode changes are determined using QE on formulas describing safety conditions and modes transitions guarded by template formulas.

In this kind of applications, even though the theoretical worst case complexity of VS is better than that of CAD, it tends to produce formulas that blow up in size and can be difficult to handle, and so CAD continues to be the method of choice in several applications. Some works such as [72] promote the combination of CAD and VS method, together with intermediate formula simplification, to address complex QE problems of safety critical control systems.

Overall, the general non-linear QE methods over reals discussed above remain very costly, which limits their applicability on real world systems. For this reason, a lot of effort have also been invested into finding efficient QE algorithms for linear fragments of both real and integer arithmetic, as well as mixed integer-real linear arithmetic. Indeed, many interesting systems and associated verification problems can be formalized using linear arithmetic: linear hybrid automata, timed automata, counter-manipulating software programs, etc. Having scalable linear QE methods can be extremely effective for such applications. Besides the well known Fourier-Motzkin quantifier elimination method, an efficient QE method is the aforementioned Virtual Substitution technique [52]. As in the non-linear case, applications of linear QE are mainly (this list is not exhaustive) pre-image computation [26], template-based invariant discovery [50] and template-based automatic program abstraction [57].

The most recent advance in real linear QE is that of lazy model enumeration based on SMT solving and polyhedral projection [56], further studied in [62, 58, 44]. This method benefits from the latest advances in SMT solving and brought a significant performance increase over existing methods.

Vocabulary and Notation. This part of the manuscript only deals with terms in quantifier-free logics. In addition, we will not use let bindings explicitly, only through unrolling of the transition relation of a transition system as defined in Section 6.1. Thus we lighten notations and will refer to “constant function symbols whose interpretation is not constrained by the underlying theory” as “variables”. In particular, unrolled state variables will be refered to as “variables”. On the other hand, constant function symbols such as “1”, “$\top$”, “5.1024”, ... which have a constrained interpretation will be called “constants”.

CHAPTER 7

SMT-Based Quantifier Elimination

The work presented in this chapter has been published at Modélisation des Systèmes Réactifs (MSR) 2013 [15]. We first present the QE algorithm introduced by Monniaux in [56] in Section 7.1. We then propose to extend it to queries mixing Booleans, integer octagons and linear real arithmetic in Section 7.2. We also propose alternative model extrapolation algorithms in Section 7.3 and compare their performances against Monniaux's SMT-test in Section 7.4.

7.1 Monniaux's QE Algorithm

The *Quantifier Elimination by Lazy Model Enumeration* algorithm from [56] is given in Algorithm 5. It takes as input a formula ϕ and a collection of quantified variables V. The formula ϕ is assumed to be in the QF_LRA fragment and V only contains real valued variables. In this algorithm, each iteration consists of three phases: first *model extraction*, then *model extrapolation*, and last *extrapolant projection*. During these iterations, models of $\phi \wedge \neg\mathcal{O}$ are obtained using satisfiability modulo theory tests. $\mathcal{O}$ is a quantifier free formula in DNF in which the variables V do not appear, generated such that at any point, any partial model of $\mathcal{O}$ can be extended to a model of ϕ. It is initially $\perp$ and is constructed incrementally. The iterations stop when the model extraction step cannot find any new models: $\phi \wedge \neg\mathcal{O}$ is unsat, *i.e.* $\phi \models \mathcal{O}$.

The goal of the extrapolation phase is to derive, from a given model of a formula, a new formula which is more general than the model (*i.e.* it characterizes more models than the one it is derived from), and still implies the formula. A formal and more general definition of the notion of extrapolant [62] follows.

Definition 7.1.1 (Model extrapolation) Given a satisfiable formula A and M a model of A, a formula E is an extrapolant of M for A if: $M \models E$, and $E \models A$.
For the sake of the algorithm, we also impose that the extrapolant is a cube (a conjunction of literals). Notice that $\bigwedge_{v \in Vars(A)}(v = M(v))$ is always an extrapolant of M for A.

In [56], the extrapolation of M is performed by a technique called *SMT-test*, which works

Algorithm 5 QE by Lazy Model Enumeration.

Require: ϕ: quantifier-free linear arithmetic formula
Require: V: collection of variables to eliminate from ϕ.
Ensure: $\mathcal{O}$: formula in disjunctive normal form such that $\mathcal{O} \equiv \exists V\ \phi$

```
function QE(V, φ)
    O ← ⊥
    while isSatisfiable(φ ∧ ¬O) do
        M ← getModel(φ ∧ ¬O)                    ▷ model extraction
        E ← extrapolate(φ, M)                   ▷ model extrapolation
        P ← project(E, V)                       ▷ extrapolant projection
        O ← O ∨ P                               ▷ incremental construction of the result
    end while
    return O                                    ▷ return the DNF formula
end function
```

as follows. First, consider the set of all the atoms of ϕ – *i.e.* linear equalities or inequalities over real terms – noted $atoms(\phi)$, and build the set of literals implied by the model:

$$\mathcal{L} \equiv \{a \mid M \models a,\ a \in atoms(\phi)\} \cup \{\neg a \mid M \models \neg a,\ a \in atoms(\phi)\} \tag{7.1}$$

At this point, $(\bigwedge_{l \in \mathcal{L}} l) \implies \phi$, *i.e.* $(\bigwedge_{l \in \mathcal{L}} l) \wedge \neg\phi$ is unsatisfiable and is an extrapolant of M for ϕ. However, it is likely that not all literals are needed. So then, build the set $\mathcal{L}' \subseteq \mathcal{L}$ by successively removing from $\mathcal{L}$ each literal l' such that $(\bigwedge_{l \in \mathcal{L},\ l \neq l'} l) \wedge \neg\phi$ is still unsatisfiable. Finally, return $(\bigwedge_{l \in \mathcal{L}'} l)$ as an extrapolant of M.

Then, once the extrapolant has been calculated, the elimination of quantified real variables is achieved by polyhedral projection, using for instance the Parma Polyhedra Library [2]. Each projected extrapolant feeds the construction of the DNF formula $\mathcal{O}$, the quantifier free result of the QE procedure.

7.2 Extension to Booleans and Integer Octagons

The original QE algorithm by lazy model enumeration described above only accepts formulas with atoms in the real linear fragment. Let us generalize it to the Quantifier-Free Linear Integer and Real Arithmetic (QF_LIRA) fragment. So, given a formula ϕ in QF_LIRA, we now have to eliminate variables V containing Boolean, integer and real variables. Provided we use a QF_LIRA capable SMT solver, the model enumeration and extrapolant generation subroutines of the original QE algorithm remain unchanged, since most of the theory specific reasoning is delegated to the SMT solver. However, the extrapolant projection phase of

the algorithm needs to be adapted. An extrapolant in this context is not a real polyhedron anymore, it is a cube c that can contain Boolean literals as well as integer and real literals. It can hence be split in three sub-cubes c_{bool}, c_{int} and c_{real} such that

- c_{bool} contains only Boolean literals,
- c_{int} (*resp.* c_{real}) contains only integer (*resp.* real) atoms, and
- $c \equiv c_{bool} \wedge c_{int} \wedge c_{real}$.

In our case, the separation of real and integer arithmetic is guaranteed by the type system of the language used to write the systems the QE challenges stem from. The set of quantified variables V can be partitioned in three subsets V_{bool}, V_{int} and V_{real} such that

- V_{bool} contains only Boolean variables,
- V_{int} (*resp.* V_{real}) contains only integer (*resp.* real) variables, and
- $V = V_{bool} \cup V_{int} \cup V_{real}$.

Property 7.2.1 (Quantifier separation) Given two quantifier-free formulas ϕ_1 and ϕ_2, and two sets of variables V_1 and V_2 such that $V_1 \cap FV(\phi_2) = \emptyset$, $V_2 \cap FV(\phi_1) = \emptyset$, and $FV(\phi_1) \cap FV(\phi_2) = \emptyset$ – where $FV(\phi)$ is the set of free variables of ϕ – then

$$\exists V_1 \cup V_2\ \phi_1 \wedge \phi_2 \equiv (\exists V_1\ \phi_1) \wedge (\exists V_2\ \phi_2).$$

Property 7.2.1 lets us separate the projection problem into three independent sub-problems which can be handled in parallel: projection of the Boolean literals, of the integer atoms and of the real atoms.

Quantifier Elimination on Linear Integer Arithmetic (LIA). Presburger proved that first-order LIA is a decidable theory by constructing a quantifier elimination procedure [63]. But to develop a QE procedure for LIA it is necessary to support linear congruence, *i.e.* terms of the form $a * x = b(\text{mod } c)$ where a, b and c are integer values and x is an integer valued variable. Indeed, eliminating y from $x = 3 * y$ for instance should yield $x = 0(\text{mod } 3)$. As we do not support the modulo operator, we restrict the integer arithmetic fragment we consider to *octagons*, *i.e.* the only legal integer literals are those of the form $v_1 + \cdots + v_i\ R\ c + v_{i+1} + \cdots + v_n$ where $v_1, \ldots, v_n$ are state variables, $R \in \{<, \leq, =, \geq, >\}$ and c is an integer constant. This is not a very strong restriction for our use cases: the integer valued variables used in critical embedded systems are mostly timers, and constraints over timers are typically linear.

So, as a result, our QE algorithm is able to handle formulas mixing Booleans literals, integer octagons and linear real literals. This extension can address typical command and

control designs which usually use Booleans to implement control logic, integers to implement various timers, watchdogs, *etc.* and reals to implement the data flow. We can thus deal with integers and Booleans natively, instead of cumbersomely relaxing them to reals to stay in QF_LRA.

In the rest of this section, we consider the language of QF_LIRA with the restriction that integer atoms are octagons:

$$\begin{aligned} B &::= \top \mid \bot \mid A \mid \neg B \mid B \wedge B \mid B \vee B \\ A &::= \langle boolId \rangle \mid R = R \mid R < R \mid I = I \mid I < I \\ I &::= \langle intId \rangle \mid \langle intCst \rangle \mid I + I \\ R &::= \langle realId \rangle \mid \langle realCSt \rangle \mid \langle realCst \rangle * R \mid R - R \mid R + R \end{aligned} \tag{7.2}$$

$\langle boolId \rangle$, $\langle intId \rangle$, $\langle realId \rangle$ represent Boolean, integer and real variable identifiers, $\langle intCst \rangle$, $\langle realCst \rangle$ represent integer and real constants, B represents formulas, A atoms, I-terms are integer-valued, R-terms are real-valued.

7.3 Improving Extrapolation

The purpose of extrapolation is to derive a formula E from a given model $M \models \phi$ which is more general than M itself, yet still entails ϕ. If we consider all possible extrapolants that can be built out of the atoms of ϕ, a *best extrapolant* for a given model M would be one with a *minimal literal count*. Such an extrapolant can be obtained by taking the conjunction of the subset of literals $\mathcal{U} \subseteq \mathcal{L}$ involved in any minimal unsatisfiable core of the following formula (where $\mathcal{L}$ is the set of literals of ϕ satisfied by the model M, *cf.* Equation (7.1)):

$$(\bigwedge_{l \in \mathcal{L}} l) \wedge \neg\phi \tag{7.3}$$

Minimal extrapolants with respect to inclusion are quite similar to *prime implicants* [78]. A prime implicant of a formula ϕ is a conjunction of literals that logically implies ϕ but ceases to when deprived of any of its literals. A *minimal prime implicant* is a prime implicant such that there exists no other prime implicant with fewer literals. Efficient algorithms exist to identify assignments of variables relevant for the satisfiability of a formula inside SAT and SMT solvers [31, 33]. In our case however, we are interested in generalizing a model with respect to $\mathcal{L}$, the set of literals appearing in the formula rather than pruning the satisfying assignment. To the best of our knowledge, little research has been done in that topic in the case of SMT.

In the QE by lazy enumeration algorithm, computing good extrapolants is the key to limiting the total number of iterations of the main model enumeration loop. However, using exact minimal unsat core computation on formula (7.3) to identify $\mathcal{U}$ for extrapolation turns out to be overly expensive in practice. It can be more beneficial for the overall performance of QE to generate relatively good approximations of the best extrapolant in shorter time.

The original extrapolation algorithm of [56], SMT-test, is an example of such a trade off between extrapolant size and computation time. With *SMT-test*, the obtained extrapolant is minimal with respect to inclusion, but not with respect to cardinality. This extrapolation approach has good generalization capabilities, but remains costly because a satisfiability check is needed to decide whether to keep or discard each possible literal.

In Subsection 7.3.1 we improve the original SMT-based extrapolation by using unsat cores to better approximate the ideal extrapolant more quickly. We then present an alternative extrapolation mechanism avoiding SMT queries by analyzing the Boolean structure of the input formula of the QE procedure in Subsection 7.3.2. Last, we discuss a third extrapolation approach in Subsection 7.3.3 combining all previously introduced optimizations.

7.3.1 Incrementality and Unsatisfiable Cores

The number of queries to the SMT solver can be reduced by taking advantage of the information on the problem structure that may have been discovered by the SMT solver as a by-product of each satisfiability check. When considering an $l' \in \mathcal{L}$ such that $G \equiv (\bigwedge_{l \in \mathcal{L},\ l \neq l'} l) \wedge \neg\phi$ is unsatisfiable, the SMT solver can be further queried to determine which literals of $\{l, l \in \mathcal{L}, l \neq l'\}$ were used in the unsatisfiability proof of G, and discard all the other literals from $\mathcal{L}$.

These literals are part of an unsatisfiable core of G, but this core is not necessarily minimal. Even though the outcome of this approach depends on how literals are considered by the decision heuristics of the solver and constraint propagation mechanisms, this provides a virtually no-cost way of potentially removing more than one literal at a time, thus reducing the number of satisfiability checks needed for *SMT-test* extrapolation.

Also, if the underlying solver allows it, we can take advantage of its assertion stack *via* the *push* / *pop* mechanism to avoid expensive restarts. The idea is to assert $\neg\phi$ at level 0 of the assertion stack, assert literals which were found necessary at level 1, and assert the literals to check at level 2. At this point, the asserted literals to check can be discarded by a simple *pop*, thus going back to level 1 of the assertion stack, preserving most of the solver state. The combination of unsat cores and incremental solving is presented on Algorithm 6.

Algorithm 6 Optimized SMT-test extrapolation.

```
Require: φ a formula, M a model of φ, useCores a Boolean.
Ensure: An extrapolant for M with respect to φ as a set of literals.
  function smtTestOpt(φ, M, useCores)
    toCheck ← ℒ                                   ▷ implied literals, cf. equation 7.1
    minimizeLits(φ, toCheck, useCores)
  end function

  function minimizeLits(φ, toCheck, useCores)
    toKeep ← ∅
    solver.assert(¬φ)                             ▷ assert ¬φ at level 0
    solver.push(1)                                ▷ now at level 1 (literals to keep)
    while toCheck ≠ ∅ do
      solver.push(1)                              ▷ now at level 2 (literals to check)
      literal ← toCheck.first                     ▷ get a literal to check from the toCheck set
      toCheck ← (toCheck \ {literal})             ▷ and remove it from the set
      for lit ∈ toCheck do
        solver.assert(lit)                        ▷ assert the literal to check at level 2
      end for
      if solver.checksat() = sat then
        toKeep ← (toKeep ∪ {literal})
        solver.pop(1)                             ▷ back to level 1 (literals to keep)
        solver.assert(literal)                    ▷ assert the necessary literal
      else
        if useCores then                          ▷ discard additional literals with unsat cores
          toCheck ← (solver.getUnsatCore() ∩ toCheck)
        end if
        solver.pop(1)                             ▷ back to level 1
      end if
    end while
    return toKeep
  end function
```

7.3.2 Extrapolation through structural analysis

Reducing the number of SMT queries during *SMT-test* extrapolation is beneficial, so it would be even better to avoid performing any satisfiability check at all. In this section, we discuss a way to identify relevant literals by analyzing the propositional structure of ϕ with respect to a given model M to generate extrapolants. A structural approach also seems to be used in [44] in the case of QE on a fragment of the theory of bit-vectors, but the authors provide very little algorithmic details on their method.

The method we propose here draws inspiration from the analysis described in [11], which processes the models returned by a model checker for Simulink data flow models to enhance their presentation to the user. The portions of the data flow network involved in the falsification of the proof objective are identified very efficiently using structural rules. From [11], we mainly reuse the notions of *sub-formula activity* and *cause*. The key idea is to keep viewing the formula as a tree, not only through its literals as in *SMT-test*, and to de-

fine rules for identifying relevant sub-trees in the context of a model thanks to the notion of active sub-formula.

Definition 7.3.1 (Active subformula) Let f be a formula and g be one of its immediate sub-formulas. We say that g is *active* with respect to f in the context of M if and only if the value of g in M *influences* the value of f in M. A sub-formula influences a formula if and only if either changing its value while preserving the values of the other sub-formulas of f would change the value of f, or its value alone fully determines the value of f in M (see also MCDC [18].

The activation conditions can be expressed as logical conditions over the values taken by the formulas in the model. To illustrate the notion, let us consider some formula ϕ and a model M such that $M \models \phi$, and assume that $a \wedge b$, a sub-formula of ϕ is active. Depending on the model M, one of the following situations occurs:

- if $M(\neg a)$ and $M(b)$, the value of a in M suffices to determine that $a \wedge b$ is false in M, but not the value of b. So a is active and b is not;
- if $M(a)$ and $M(\neg b)$, the value of b in M suffices to determine that $a \wedge b$ is false in M, but not the value of a. So b is active and a is not;
- if $M(\neg a)$ and $M(\neg b)$, then the value of either a or b in M independently determines that $a \wedge b$ is false in M. So both a and b are active, and this is a case of *independent cause combination*;
- if $M(a)$ and $M(b)$, the values of both a and b are needed to determine that $a \wedge b$ is true in M. So both a and b are active, and this is a case of *dependent cause combination*.

So for $a \wedge b$, the activation condition of a can be reduced to $M(\neg a \vee b)$, and the one for b to $M(\neg b \vee a)$. Similar conditions can be written for all logical operators of the language, reminding of Boolean Difference Calculus [64].

Definition 7.3.2 (Causes) Given a formula ϕ and a model M, a *cause* for a sub-formula g active with respect to ϕ, is a cube k over the free variables of ϕ, such that $M \models k$ and for any M' such that $M' \models k$, $M'(g) = M(g)$. There can be more than one cause for a given g with respect to a given model, the set of causes is denoted by $K_M(g)$, or $K(g)$ when M is obvious from context.

Moreover, the set of causes for a formula can be built inductively from the sets of causes of its sub-formulas. For the unary operator $\neg a$, causes are inherited from the sub-formula. For a binary operator $\langle op \rangle \in (\wedge, \vee)$, $K_M(a \langle op \rangle b)$ is equal to:

Term	Condition	Active sub-formulas	Returned causes/Combination
Boolean $\langle id \rangle$	$\mathbf{T}$	[]	$\langle id \rangle$ if $M(\langle id \rangle)$, $\neg\langle id \rangle$ otherwise
$\neg a$	$\mathbf{T}$	$[a]$	$K(a)$
$a \wedge b$	$M(\neg a)$	$[a]$	$K(a)$
	$M(a \wedge \neg b)$	$[b]$	$K(b)$
	$M(a \wedge b)$	$[a, b]$	$K(a) \wedge K(b)$
$a \vee b$	$M(a)$	$[a]$	$K(a)$
	$M(\neg a \wedge b)$	$[b]$	$K(b)$
	$M(\neg a \wedge \neg b)$	$[a, b]$	$K(a) \wedge K(b)$
$x = y$	$M(x = y)$	[]	$(x = y)$
	$M(x < y)$	[]	$(x < y)$
	$M(y < x)$	[]	$(y < x)$
$x < y$	$M(x < y)$	[]	$(x < y)$
	$M(\neg(x < y))$	[]	$\neg(x < y)$

Figure 7.1: Activation conditions and cause propagation rules.

- $K_M(a)$ if only a is active ($K_M(b)$ if only b is active);
- $K_M(a) \cup K_M(b)$ if both a and b are active in dependent combination;
- $\{C_a \wedge C_b,\ (C_a, C_b) \in K_M(a) \times K_M(b)\}$ if both a and b are active in independent combination.

These combination rules and the definition of a cause entail that if a (resp. b) was active with respect to $a \langle op \rangle b$ in M, a (resp. b) is still active in any model $M' \models k$ with $k \in K_M(a \langle op \rangle b)$, activation conditions being preserved in any $M' \models k$. As a consequence, we can see that if $M \models \phi$, any $k \in K_M(\phi)$ is in fact an extrapolant of M with respect to ϕ, because $M \models k$ and any $M' \models k$ is such that $M'(\phi) = M(\phi) = \top$, which essentially means that $k \implies \phi$.

In [11] the authors extract all possible causes for the root node of the formula by handling both dependent and independent cause combinations. However, our experiments in the context of QE have shown that the independent cause combination, which leads to the expansion of the Cartesian product of cause sets, leads to a detrimental blowup in the number of computed causes.

So in practice, we decide to generate a single extrapolant per model, and to do so we only handle *dependent cause combination*. In cases where *independent cause combination* occurs, we choose to simply compute and propagate causes for the leftmost active sub-formula. Algorithm 7 gives the recursive descent and bottom-up combination method we use to generate

Algorithm 7 Structural extrapolation.

Require: ϕ a formula, M a model of ϕ.
Ensure: An extrapolant for M with respect to ϕ as a set of literals.

function $struct(\phi, M)$
  $atoms(cause(\phi, M))$
end function

Require: f a formula, M a model.
Ensure: A cause for f in M is generated.

function $cause(f, M)$
  $activeSubFormulas \leftarrow active(f, M)$ ▷ identify active sub-formulas (*cf.* Fig 7.1)
  $subCauses \leftarrow activeSubFormulas$ **map** $\{x \rightarrow cause(x, M)\}$ ▷ recursive call
  return $combine(f, subCauses, M)$ ▷ combine sub-causes (*cf.* Fig 7.1)
end function

extrapolants.

All rules defining the functions $active$ and $combine$, are given in Figure 7.1. The *Term* column specifies the operator in question, the *Condition* column specifies the logical condition to be evaluated. The column *Active sub-formulas* gives the behavior of the $active$ function, which returns a list of active formulas. The column *Returned causes* gives the behavior of the $combine$ function, which returns a cause either freshly generated or obtained by combining sub-causes.

The first line of the table reads as: if the current formula is a Boolean identifier (*i.e.* a free Boolean variable), the recursion stops, and if its value is $\top$ in the model, the returned cause is the variable itself, otherwise it is the negation of this variable. When traversing Boolean operators ($\neg$, $\wedge$, $\vee$), the recursive descent continues on active sub-formulas, and causes of sub-formulas are combined to obtain a cause for the current formula. For instance, the third line of the table reads as: if the current formula is of the form $a \wedge b$, if $M(\neg a)$ is true, then sub-formula a only is active, when folding up the cause of a shall be returned for $a \wedge b$; otherwise if $M(a \wedge \neg b)$ is true, then b only is active, when folding up the cause of b shall be returned for $a \wedge b$; otherwise if $M(a \wedge b)$ is true, then both a and b are active, when folding up the conjunction of the cause of a and the cause of b shall be returned for $a \wedge b$. The recursive descent stops when it reaches arithmetic atoms ($=$,$<$), and an arithmetic literal is returned as cause. For instance, the fifth line of the table reads as: if the current formula is of the form $x = y$, then if $M(x)$ equals $M(y)$, the cause $x = y$ shall be returned, otherwise if $M(x) < M(y)$, then the cause $x < y$ shall be returned, last if $M(y) < M(x)$, the cause $y < x$ shall be returned[1].

[1]These literals are not necessarily found in the formula. Since $\neg(x = y)$ is essentially $(x < y) \vee (x > y)$, it is rewritten to either $(x < y)$ or $(x > y)$ depending on the model, to derive a single cause for that literal. This

7.3.3 Combining extrapolation techniques

The structural analysis of Section 7.3.2 can be used as a pre-processing step for the optimized SMT-test extrapolation procedure, providing a smaller set of literals to begin with. This hybrid extrapolation method is given in Algorithm 8. The next section explicits the benefits of this approach thanks to experimental results.

Algorithm 8 Hybrid extrapolation.

Require: ϕ a formula, M a model of ϕ, $useCores$ a Boolean.
Ensure: An extrapolant for M with respect to ϕ as a set of literals.
  function $hybrid(\phi, M, useCores)$
    $toCheck \leftarrow struct(\phi, M)$
    return $minimizeLits(\phi, toCheck, useCores)$
  end function

7.4 Benchmarking Results

We have implemented all the algorithms presented in this chapter[2], using the Microsoft Z3 [29] SMT solver and the Parma Polyhedra Library (PPL [2]) for the generalized projection discussed in Section 7.2. Integer octagons are relaxed to real octagons. Note that when PPL returns the result, we do not need to check that it is still legal when transposing it to integers: the literals given to PPL are satisfiable and therefore the cube contains at least one integer solution. In the implementation of Algorithm 5, model extraction takes advantage of the incremental nature of modern SMT solvers. During the first iteration, the algorithm asserts ϕ in the solver, and then asks for its satisfiability. If it is sat, then a model is extracted, extrapolated and then projected, resulting in a cube c. If the state of the solver is kept during these operations, then at the beginning of the second iteration the algorithm only needs to assert $\neg c$ in the SMT solver and does not need to re-learn information discovered during the check-sat of ϕ. The same goes for all following iterations.

The benchmarks used to evaluate the algorithms are QE jobs generated by our formal framework Stuff corresponding to successive symbolic pre-images computations of a negated safety proof objective (PO) on a transition system, both derived from a Lustre file. More precisely, if $T(s, s')$ is the transition relation of the system applied to state s and *next state* s', the iterative pre-image computation consists in

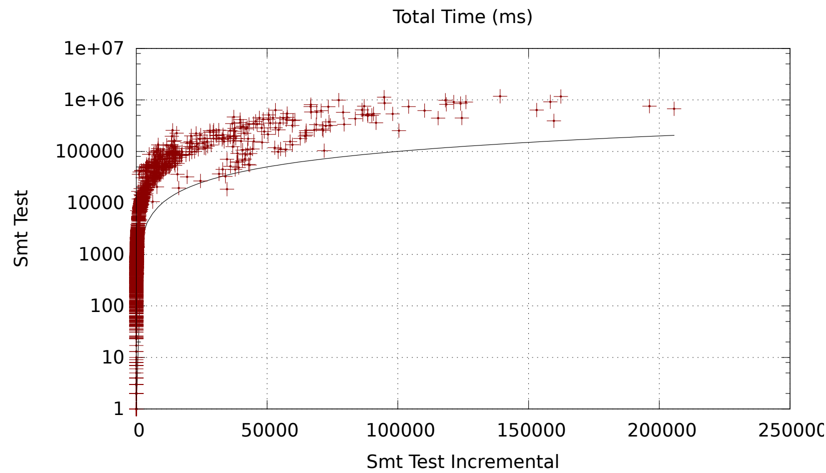
$$\mathcal{G}_1(s) \equiv QE(s', \mathrm{PO}(s) \wedge T(s, s') \wedge \neg\mathrm{PO}(s')) \tag{7.4}$$

$$\mathcal{G}_{i+1}(s) \equiv QE(s', \mathrm{PO}(s) \wedge T(s, s') \wedge \mathcal{G}_i(s')) \tag{7.5}$$

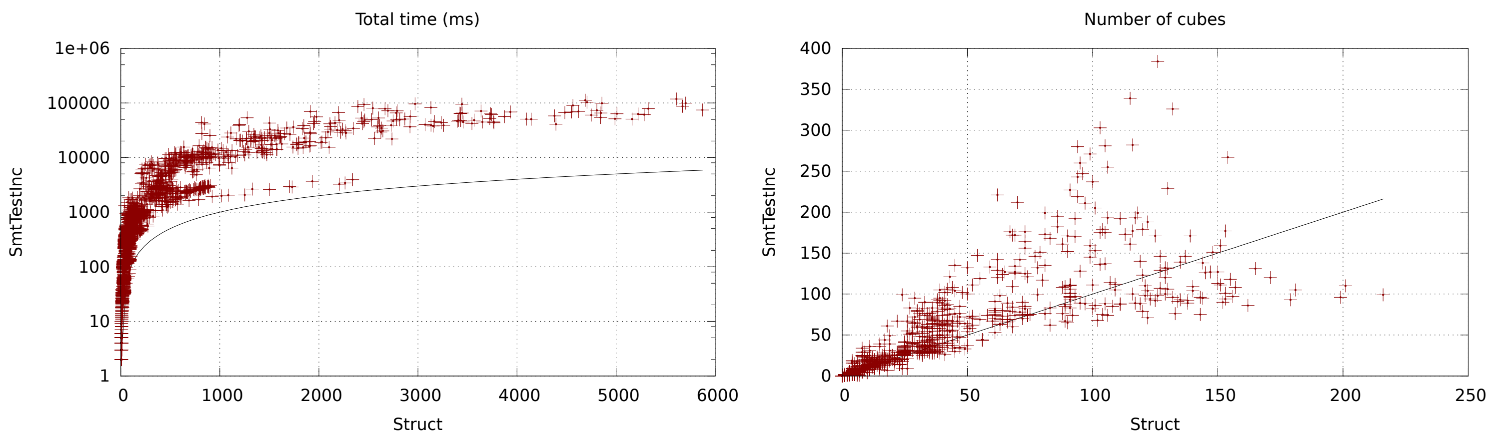
slightly escapes the scope of the definition of best extrapolant.

[2]Available at https://cavale.enseeiht.fr/redmine/projects/smt-qe/files

So $\forall i \geq 1$, the formula $\mathcal{G}_i$ represents states satisfying PO from which a trace of i transitions leading to ¬PO exists.



(a) Incremental SMT-test / SMT-test.



(b) Structural analysis / Incremental SMT-test.

Figure 7.2: Benchmark results part 1.

All experiments were conducted on a dual-core $i5$ laptop with 4Gb of RAM memory running Linux. We deactivated the unsat core optimization after observing that it significantly dragged down the performance of Z3 – in practice about 50% slower satisfiability checks when also using push and pops. Comparisons with techniques not using them would thus not be fair.

We compared the techniques presented in this chapter on Lustre files from the Kind benchmark database[3]. For each file, we generated QE challenges corresponding to the first

[3]Available at http://clc.cs.uiowa.edu/Kind/index.php?page=experimental-results.

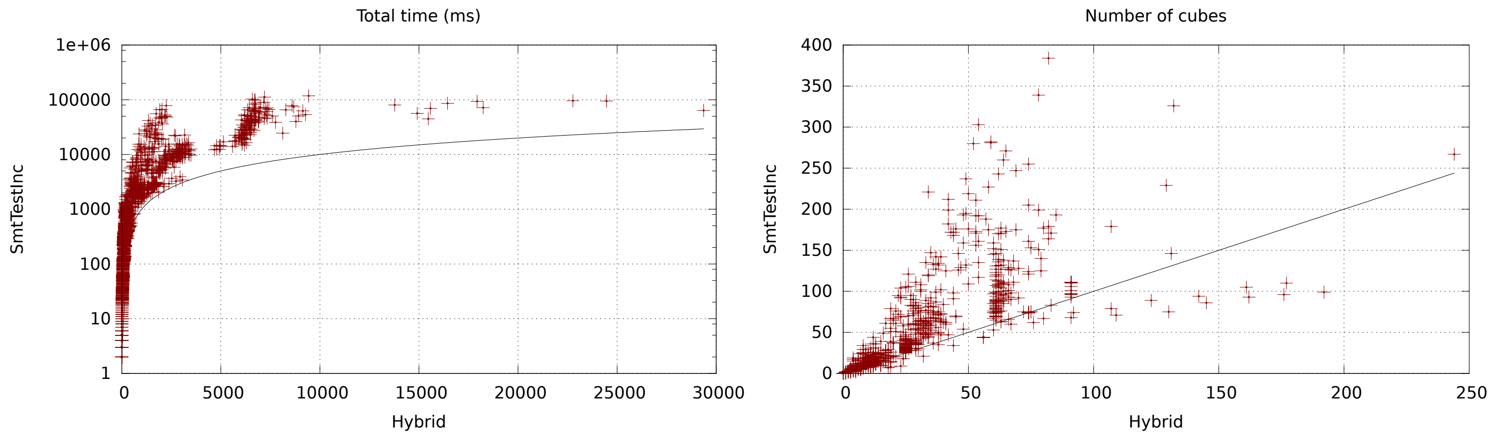
twenty pre-images of the proof objective, for a total of about 2800 QE challenges. The number of variables to eliminate ranges from 0 to 105, with an average of 24. The pairwise comparison plots for total time and number of cubes in the result formula are given in Figure 7.2 and Figure 7.3. They feature

- *SMT-test*, the original extrapolation algorithm proposed by Monniaux;
- *SMT-test Inc*, Algorithm 6 without unsat cores;
- *Struct*, the structural extrapolation from Algorithm 7;
- *Hybrid*, from Algorithm 8 without unsat cores.

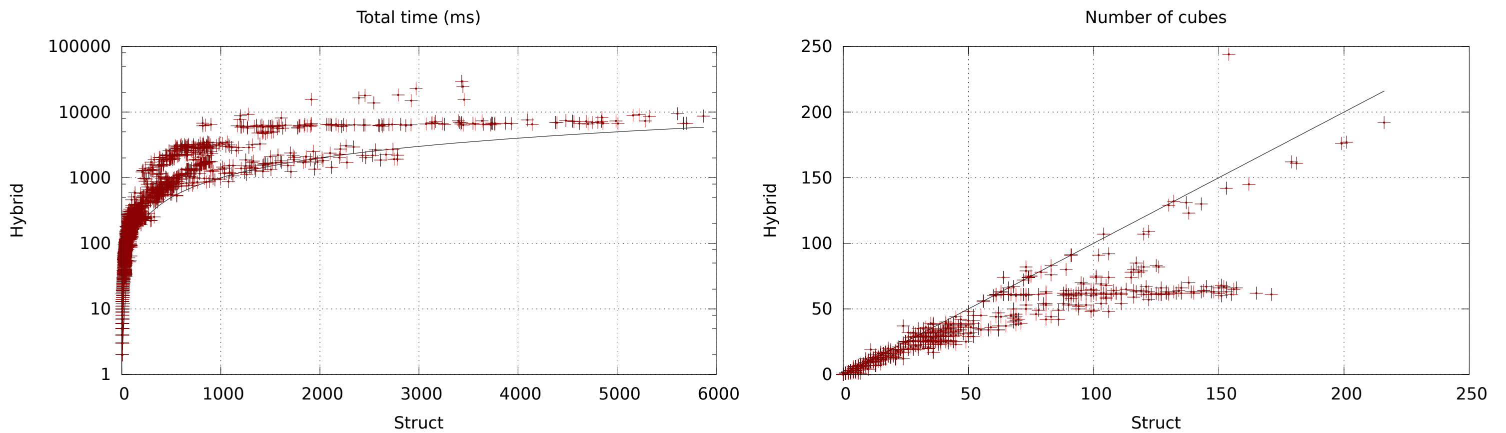
The interest of using incrementality in SMT-test is confirmed on Figure 7.2a –we thus dismiss SMT-test (without incrementality) from the rest of the benchmarks analysis. Figure 7.2b shows that it is indeed even more effective for runtime not to perform any satisfiability checks at all using only structural extrapolation, although the resulting DNF is not always as concise as it could be. Hybrid improves on that aspect while still being a lot faster than SMT-test Inc (Figure 7.3a). Last, Figure 7.3b shows that despite the fact that Hybrid performs less model enumeration iterations[4], it is still generally slower than Struct because of the costly satisfiability checks it performs after the structural extrapolation phase.

For a deeper analysis of the differences between SMT-test Inc, Struct and Hybrid consider now the data presented on Figure 7.4. It corresponds to benchmarks results on two industrial systems, a triplex voting logic provided by Rockwell Collins and a reconfiguration logic system. More precisely, the QE jobs on the reconfiguration logic system are the five first pre-images of the negation of "assuming at most two faulty channels out of three, it cannot be the case that no channel is in control for more than n cycles". Integer constant n depends on the bounds on the timers used to confirm failures. For the triplex voter, the first three pre-images of the negation of a Bounded Input implies Bounded Output (BIBO) proof objective are considered. These systems are discussed in more details in Chapter 9, and their code is available in Appendix A. We noticed that SMT-test Inc execution times and number of iterations vary widely from run to run on a same benchmark with variations of $+/-50\%$. This is due to its sensitivity to the order in which it checks each literal and –like Struct and Hybrid– to the order in which the models are found. The Struct technique on the other hand is more stable with variations of about $+/-5\%$. On Figure 7.4 we notice that Struct is by far faster, by orders of magnitude in some cases, on all benchmarks, than the reference SMT-test Inc and the Hybrid techniques.

[4] *i.e.* fewer cubes are computed.



(a) Hybrid / Incremental SMT-test.



(b) Structural analysis / Hybrid.

Figure 7.3: Benchmark results part 2.

The benefits of structural analysis are clear: identifying the active atoms by following active branches of the formula generalizes the model while avoiding SMT queries, thus resulting in a considerable speed-up. Even when producing more cubes in $\mathcal{O}$ than QE with *SMT-test Inc* (see Reconf 4 on Figure 7.4), QE with *Struct* is still orders of magnitude faster. This is not surprising since *SMT-test Inc* considers all n_a atoms of the formula before generalizing at the cost of n_a satisfiability checks. The downside is that by not considering the atoms semantics, only their role in the structure of the formula, unnecessary atoms can accumulate in the extrapolant and make it less general. The QE with *Hybrid* extrapolation technique illustrates this: thanks to additional SMT queries, it outputs a DNF up to three times smaller than QE with *Struct* alone, hence showing a better generalization power, albeit at the cost of speed.

		Reconf					Triplex		
		1	2	3	4	5	1	2	3
Total time ms	SMT-test Inc	53,172	114,371	99,953	131,855	318,257	13,855	30,163	119,771
	Struct	**800**	**883**	**2,063**	**5,587**	**10,078**	**1,627**	**4,785**	**23,413**
	Hybrid	25,078	27,837	31,024	50,160	51,913	7,005	16,978	42,063
Number of cubes in $\mathcal{O}$	SMT-test Inc	55	106	125	119	264	33	38	84
	Struct	37	**38**	84	153	243	26	31	51
	Hybrid	**37**	39	**45**	**70**	**74**	**23**	**28**	**48**
Time ms / step	SMT-test Inc	966	1,078	799	1,108	1,205	419	793	1,425
	Struct	**21**	**23**	**24**	**36**	**41**	**62**	**154**	459
	Hybrid	677	713	689	716	701	304	606	876
Extrapolation ms / step	SMT-test Inc	938	1,049	768	1,075	1,163	369	618	919
	Struct	**1.810**	**0.684**	**0.595**	**0.483**	**0.427**	**0.115**	**0.290**	**0.117**
	Hybrid	652	687	661	686	667	261	361	392

Figure 7.4: Experimental results.

CHAPTER 8

Invariant Discovery in a K-induction-based Framework

In this chapter we introduce our proof engine architecture. It is centered around k-induction, used to continuously confirm or invalidate potential invariants generated by the methods integrated to the framework. Result are communicated to the whole framework so that all methods benefit from the information discovered. We present the architecture in Section 8.1, and discuss two invariant generation methods featured in our implementation of the architecture, the Stuff formal framework. First, a simple template-based invariant discovery technique in Section 8.2 illustrates our interest in Quantifier Elimination. This approach is used in our formal framework to discover non-relational invariants, *i.e.* ranges over the state variables. We then present our generalization of PDR to SMT using (partial) QE in Section 8.3.

8.1 A Proof Engine Architecture Allowing Continuous Invariant Integration

We address the problem of the verification of reachability properties on Lustre programs with the following strategy, illustrated in Figure 8.1 and similar to [48]: a multi-proof objective incremental k-induction engine (*cf.* Section 6.2.2) runs in parallel with some potential invariant discovery techniques. As soon as new potential invariants are discovered, they are sent to the k-induction engine in conjunction with the main proof objective PO of the analysis. The k-induction is parameterized by a maximal induction length n to stop at. Valid (set V_n) proof objectives are separated from Falsified (set F_n) and Unknown (set U_n) ones. Elements of F_n are memorized to avoid running k-induction on them again. If PO $\in F_n$, then the main proof objective does not hold and the analysis is over. Elements of U_n are memorized for the next k-induction run. V_n contains invariants of the system. They are broadcast to the rest of the framework so that all methods can integrate them. If PO $\in V_n$, the main proof objective holds and the analysis is over.

Yet, in this case, it might happen that not all elements of the valid subset V_n are needed

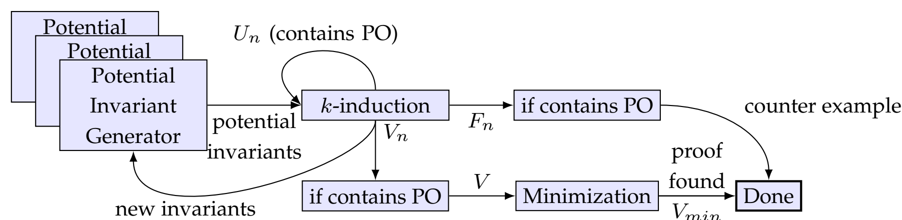


Figure 8.1: k-induction based collaborative framework.

to strengthen PO. A minimization pass inspects them one by one, discarding $l \neq$ PO from V_n if $\bigwedge(V_n \setminus l)$ remains n-inductive, to obtain a relatively small and readable set of invariants $V_{n,min}$. This set is hence minimal with respect to inclusion but not cardinality, as it depends on the order in which its elements are considered.

Two kinds of invariants have to be distinguished. *Non-relational* invariants only mention one state variable. They usually express bounds on a state variable, such as $(0 \leq x) \wedge (x \geq 42)$. Non-relational invariants are rarely enough to strengthen a proof objective but drastically reduce the search space for all techniques running in the framework. One can discover them rather easily using Abstract Interpretation with intervals or template-based methods as detailed below. *Relational* invariants on the other hand exhibit a link between two variables. For instance, $x - y \leq 4$, or $x + y + z \geq 0$. Relational invariants are difficult to discover: abstract interpretation requires experts tuning for the choice of the abstract domains and abstract semantics. For template-based techniques, the right template must exist in the template database. Even in that case, the many templates of the database need to be instantiated heuristically on the many state variables and constants of the system.

Some methods such as the ones presented in this chapter generate actual invariants by construction. In theory, they do not need to be confirmed by the k-induction engine. In qualification context however it is better to have k-induction confirm them anyway: qualification of a verification tool is a long and complex process. By validating invariants systematically by k-induction, it means that the only parts of the framework that need to be qualified are *(i)* the translation of the input program to the internal transition system representation, *(ii)* the k-induction engine, including the underlying SMT-solvers. Even if the rest of the framework is not correct, it does not matter as long as k-induction is the only one allowed to confirm invariants.

8.2 Invariant Discovery as a Quantifier Elimination Problem

We illustrate the power of quantifier elimination by exposing a first simple invariant generation technique implemented in Stuff and nicknamed *BrutalIQe*. We follow the ideas developed in [49] – briefly outlined in the introduction of Part II – but restrict ourselves to linear arithmetic fragments for which much more scalable quantifier elimination algorithms are available. This allows us to consider systems with integer valued variables as well as reals and Booleans, which is not possible in [49] since there is no QE algorithm in the non-linear integer fragment. This also means that we do not have the possibility to consider non-linear templates.

To give more details on the approach, consider a transition system with initial states predicate I, transition relation predicate T, and proof objective PO. Assume an oracle provides us with linear templates on the state variables of the system, *e.g.*

$$template(var_1, var_2)(min, max) \triangleq (min \leq var_1 + var_2) \wedge (var_1 + var_2 \leq max)$$

viewed as functions $template(vars)(params)$ returning a formula. Argument $vars$ represent state variables of the system, while $params$ represents the template parameters. Given a vector of state variables $sv \subseteq s$ of the actual transition system, the idea of the approach is to find a valuation p for the template parameters such that $template(sv)(p)$ is an invariant for the transition system. By substituting $template(sv, p)$ for P in base and step formulas of an induction proof attempt, the fact that the template is (1-)inductive is equivalent to

$$\overbrace{(I(s) \wedge \neg template(sv)(p))}^{\text{base instance}} \quad \vee \quad \overbrace{(template(sv)(p) \wedge T(s, s') \wedge \neg template(sv')(p))}^{\text{step instance}} \tag{8.1}$$

being unsatisfiable. Next, we build the following formula, whose models are exactly the parameters values for which the template is inductive:

$$\forall s,\ s',\ \neg\Big(\quad (I(s) \wedge \neg template(sv)(fresh)) \\ \vee \quad (template(sv)(fresh) \wedge T(s, s') \wedge \neg template(sv')(fresh))\Big) \tag{8.2}$$

where *fresh* is a vector of fresh variables which do not appear elsewhere in the system. This formula can be rewritten as:

$$\neg\Big(\exists\{s, s'\} \quad (I(s) \wedge \neg template(sv)(fresh)) \\ \vee \quad (template(sv)(fresh) \wedge T(s, s') \wedge \neg template(sv)(fresh))\Big). \tag{8.3}$$

At this point we can use QE to eliminate the system state variables, and obtain a quantifier-free characterization of these parameter values:

$$\mathcal{C} \triangleq \neg QE\Big(s \cup s', \quad (I(s) \wedge \neg \mathit{template}(sv)(\mathit{fresh})) \\ \vee \quad (\mathit{template}(sv)(\mathit{fresh}) \wedge T(s,s') \wedge \neg \mathit{template}(sv')(\mathit{fresh}))\Big). \tag{8.4}$$

Note that $\mathcal{C}$ only mentions variables from fresh. If $\mathcal{C}$ is unsatisfiable, then no values of fresh can make $\mathit{template}$ inductive for this system without further strengthening. If it is satisfiable, the formulas obtained by replacing fresh by their values in any model of (8.4) in the template are (1-)inductive by construction.

The template can be modified to make it *property-directed*, and obtain an explicit characterization of parameter values such that $\mathit{template}$ is a strengthening invariant for the PO:

$$\mathit{template}_{str} \triangleq \text{PO} \wedge \mathit{template} \tag{8.5}$$

Note also that this approach can easily be generalized to build k-inductive invariants, although it becomes considerably more expensive as k increases.

This technique is implemented in our framework under the nickname BrutalIQe. We use it to try and find inductive bounds on the system state variables by using templates of the form $(\mathit{min}\ R\ \mathit{mem})$, $(\mathit{mem}\ R\ \mathit{max})$ and their conjunction, where mem is a state variable and $R \in \{\leq, <\}$. This analysis is usually performed as a pre-processing step so that all the other techniques of the framework can benefit from it from the beginning. Once all the state variables have been tried, more complex templates drawn from a database can be considered, based on octagons for example, or arbitrary polyhedra.

Abstraction Interpretation using intervals, octagonal or polyhedral abstract domains could be used to obtain similar results. But we find this technique to be more accessible to anyone with a working knowledge of SMT-solvers and benefits from the rather steady advances made by the SMT community – at least when using the SMT-based QE algorithm from [56].

8.3 Property Directed Reachability Modulo Theory

As discussed in [19] page 6, most of the adaptations of PDR to SMT consists in replacing the underlying SAT solver by an SMT solver. Instead of learning cubes of Boolean literals, PDR learns cubes of Boolean or theory literals. In the following we discuss the two biggest problems of this adaptation: the generalization phase in Subsection 8.3.1 and frame

size management and implication checks in Subsection 8.3.2. We conclude this section by reporting on our experiments in Subsection 8.3.3.

8.3.1 Generalization Phase in SMT

Assume an intermediary proof objective is not blocked at frame k, *i.e.*

$$\mathcal{F}_{k-1}(s_0) \wedge T(s_0, s_1) \wedge \mathrm{IPO}_k(s_1)$$

is sat. Then, what new IPO should be created for frame $k-1$? Ideally, it should be the whole pre-image of $\mathrm{IPO}_k(s_1)$:

$$\mathrm{IPO}_{k-1} \triangleq QE(s_1, \mathcal{F}_{k-1}(s_0) \wedge T(s_0, s_1) \wedge \mathrm{IPO}_k(s_1)). \tag{8.6}$$

But remember that the strength of PDR lies in performing many relatively simple queries instead of a few complex ones. Quantifier Elimination is a complex and time consuming operation, so this approach does not scale up to real systems. It is more efficient to compute an under approximation of the pre-image, as also noticed by the author of [19].

One Step QE. Consider the QE algorithm discussed in Chapter 7. When eliminating variables $\langle v_1, \ldots, v_n \rangle$ on formula ϕ, at the end of the first iteration a cube c is produced such that $c \wedge \neg\phi$ is unsat, and $\langle v_1, \ldots, v_n \rangle$ do not appear in c. So, in the case of (8.6), c only mentions s_0 and

$$c \wedge \neg(\mathcal{F}_{k-1}(s_0) \wedge T(s_0, s_1) \wedge \mathrm{IPO}_k(s_1))$$

is unsat. That is to say c characterizes only states which can reach IPO_k in one transition from $\mathcal{F}_{k-1}$: it is an under approximation of the pre-image from (8.6).

So in practice, our generalization phase consists in one step of the QE algorithm described in Chapter 7.

8.3.2 Frame Size Management

During a run of PDR, most of the time is spent blocking cubes. The frames end up learning so many of them that the satisfiability checks become very expensive: the formulas rapidly become huge because of the sheer number of cubes representing the frames. A cheap way to reduce the size of the frames is to identify cubes implying each other. For instance, it can be the case that cubes $a \wedge b \wedge \neg c$ and $a \wedge b$ are both blocked at the same frame. The first cube can be safely removed as it is more restrictive than the second one; identifying such situations can be done syntactically in the propositional case. If the cubes are represented as

sets of literals, then if cube c_1 is a subset of cube c_2 then cube c_2 can be safely removed from the frame.

This does not work in the SMT case, *e.g.* for cubes $x = 2 \wedge y \geq 0$ and $x = 2 \wedge y \geq 10$. It is necessary to consider the semantics of arithmetic to realize that the first cube is more general, for instance by a satisfiability check. Consider now the cubes $x = 2 \wedge y \geq 10$ and $x = 2 \wedge y < 10$. Obviously, if both this cubes are blocked at a frame, it would be better to replace them by the cube $x = 2$ as it makes the formula of the frame more compact. We say that cube $x = 2$ is the *exact convex hull*[1], or ECH, of $x = 2 \wedge y \geq 10$ and $x = 2 \wedge y < 10$, *i.e.*

$$(x = 2) \Leftrightarrow ((x = 2 \wedge y \geq 10) \vee (x = 2 \wedge y < 10)).$$

We say that the cubes have been *merged exactly* in cube $x = 2$. Notice that exact convex hulls also cover the case when a cubes c_1 implies a second cube c_2: the ECH of $c_1 \vee c_2$ is simply c_1. Now, remember the propagation phase: if

$$\mathcal{F}_i(s_\mathbf{0}) \wedge T(s_\mathbf{0}, s_\mathbf{1}) \wedge c(s_\mathbf{1}).$$

is unsat for some cube c blocked by $\mathcal{F}_i$, then add c to $\mathcal{F}_{i+1}$. Propagating cubes can cause future cube blocking phases to finish earlier and is a crucial mechanism of PDR. Replacing cubes by their ECH when it exists can hinder this process: it could be that $x = 2 \wedge y \geq 10$ could be propagated to the next frame, but not $x = 2 \wedge y < 10$. Replacing these two cubes by $x = 2$ would prevent the propagation of $x = 2 \wedge y \geq 10$.

Exact convex hull representation. With these remarks in mind, we propose that every frame, except frame 0 which is the initial states, has a second set of cubes which are exact convex hulls of the original set of cubes and hence more compact. Whenever the formula of frame i is needed, *e.g.* to check if some cube is blocked at frame $i + 1$, then the disjunction of the exact convex hulls is used. The propagation phase on the other hand considers the original set of cubes to raise the chance that a cube is indeed propagated. Assume that a new cube $cube$ is blocked at frame $i > 0$. The update of the exact convex hulls $echs$ of the frame are computed by the function $mergeWithCube$ on Algorithm 9. It tries to exactly merge $cube$ with each element of $hulls$. If an exact merge with ech succeeds – the result is not **false** – and produces ECH $exactHull$, the function returns all the elements of $echs$, except ech, augmented with $exactHull$. If no ECH is found, the function returns $echs$ augmented with $cube$. Function $mergeWithCubes$ generalizes this idea to more than one new cube.

Note that it could be the case that when $mergeWithCube(s)$ returns, more exact convex hulls can be computed. It would be too expensive to try all combinations of the cubes of the

[1] Exact (and inexact) convex hulls will be discussed to much greater length in the next chapter.

Algorithm 9 Exact hull computation algorithms.

Require: $cube$ a cube, $echs$ a set of cubes.
  function $mergeWithCube(cube, echs)$
    for $(ech \in echs)$ **do**
      $exactHull = computeHull(cube, ech)$
      if $(exactHull \neq$ **false**$)$ **then**
        return $(echs \setminus ech) \cup \{hull\}$
      end if
    end for
    return $echs \cup \{cube\}$
  end function
Require: $cubes$ and $echs$ two sets of cubes.
Ensure: $\bigvee result$ is equivalent to $\bigvee(cubes \cup echs)$
  function $mergeWithCubes(cubes, echs)$
    $result = echs$
    for $(cube \in cubes)$ **do**
      $result = mergeWithCube(cube, result)$
    end for
    return $result$
  end function

output. Note also that the ECHs found depend on the order in which the merges are tried. In practice, updates of the ECH representation of a frame are performed asynchronously[2] to avoid slowing down the cube blocking phase. The principle is depicted on Figure 8.2. A frame is represented by

- $cubes$: the cubes blocked at this frame;
- $echs$: the ECHs computed so far;
- mem: a memory used to store new blocked cubes received while computing ECHs.

When new cubes blocked at frame i are received – **newCubes** – the computation of exact convex hulls is delegated to a separate process. The ECH representation of the frame – $echs$ – is augmented with the new blocked cubes. While the ECH process is working, new blocked cubes received by the process of the frame – **newCubes**′ – are stored separately in mem. The invariant of the internal state of the frame is that $echs$ contains all the ECHs found so far, as well as the cubes currently being merged with these ECHs by the separate process. Also,

[2] Using the actor paradigm discussed in Section 10.1.

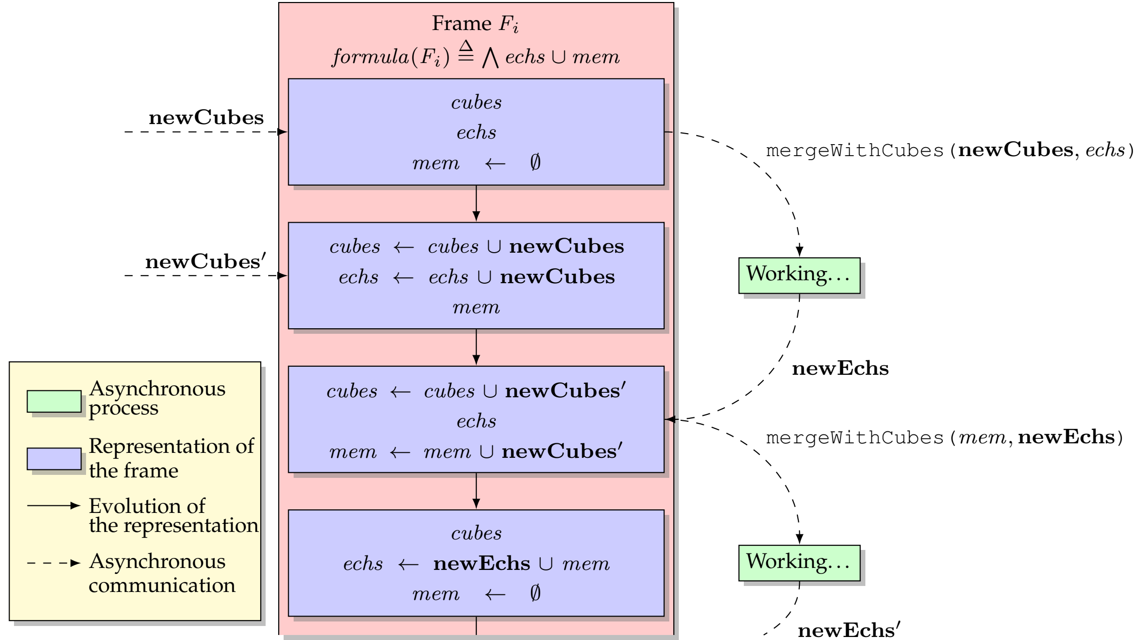


Figure 8.2: Asynchronous exact convex hull computation.

mem contains the blocked cubes received since the last ECH computation job was launched. At any time, the formula of the frame is given by

$$\bigwedge echs \cup mem.$$

Once the process which has been computing the ECHs communicates its results to the frame, the ECH representation of the frame is updated, a new ECH computation job is sent, and mem becomes the empty set. The formula of the frame is thus always as concise as it can be.

Consider now the propagation phase. Frame F_{i+1} receives some cubes $newCubes_i$ propagated from frame F_i, and now has to check if some new cubes should be propagated to frame F_{i+2}. Note that this will not change the state of F_{i+1} in any way. So, we launch asynchronously a different ECH computation algorithm `fixedPoint` given on Algorithm 10 on the ECH representation of F_{i+1} plus the propagated cubes. This function tries to merge all its input cubes two by two until it reaches a fixed point, *i.e.* no more ECH can be found exist. While the hulls found also depend on the order in which the cubes are considered, this makes the ECH representation of the frame, and hence its formula, more compact during the propagation phase.

Algorithm 10 Fixed point ECH computation.

Ensure: $\bigvee result \equiv \bigvee echs$
Ensure: No more ECH can be computed between the elements of $result$.

```
function fixedPoint(echs)
    old = echs
    result = mergeWithCubes(old, ∅)
    while (result.size ≠ old.size) do
        old = result
        result = mergeWithCubes(old, ∅)
    end while
    return result
end function
```

8.3.3 Conclusion

Before concluding on our adaptation of PDR to SMT, we want to point out that the double representation of the frames presented in the previous section could have a prohibitive memory cost. However, the expression data structure Stuff uses makes heavy use of expression memoization. Hence, whenever a (sub-)expression appears multiple times in the framework, only one instance actually exists in memory. In practice, we found that the memory consumption of this approach stays manageable.

Our implementation of PDR in Stuff was started relatively late and is still a prototype. It features the ideas presented in this section and is runnable. Unfortunately our experiments on the systems at our disposal indicate that it will be difficult to outbalance the fact that PDR is sensitive to the reocurrence diameter of the systems it analyzes, although it can outperform k-induction. This is a serious problem in the SMT case, where it is not rare to encounter systems with a huge, non-tractable state space. As mentioned before, most critical systems handling real numbers corresponding to measurements in the real world prevent their output from increasing too much in one step to avoid unrealistic discontinuities in the signal due to sensor perturbations. This is generally the case for instance of voting systems. As a result, it can take thousands or even millions of transitions to realize that, say, "the absolute value of the output is always less than some value". This means thousands or millions of cube generalization, cube blocking, propagation phases... for PDR to conclude the proof, which can be very expensive in the SMT case. The same applies to analysis of timers which can count thousands of cycles before confirming a failure.

Now, the interest of our architecture is that PDR is not alone and can benefit from in-

formation discovered by other techniques such as BrutalIQe, and HullQe (discussed in the next chapter). Once confirmed by k-induction, actual invariants ($invs(s)$) strengthen the transition predicate $T(s, s')$:

$$T_{str}(s, s') \triangleq invs(s) \wedge T(s, s') \wedge invs(s').$$

As a result, cubes to block are more likely to be blocked; if they are not blocked, the cube generalization is more precise in that it considers fewer states s pre-images of the cube to block. As we shall see however, BrutalIQe and HullQe alone conclude the proof on our voting and reconfiguration logic systems. We are thus looking for more realistic systems to evaluate the interest of the combination of our PDR implementation with the other techniques of the framework.

The work presented in this thesis focuses on finding a proof that a proof objective is an invariant for a system, but sometimes the proof objective is falsifiable. In this case, formal tools such as Stuff are expected to output a counterexample trace rooted on initial states leading to a violation of the proof objective. PDR has two interesting strengths when it comes to counterexample generation. First, it is faster than BMC and k-induction because while it combines backward analysis and reachability from initial states. The counterexample trace is constructed by computing pre-images of cubes rather than by unrolling the transition relation enough to reach a bad state. The second strength of PDR is that it does not generate a trace of states, but rather a trace of cubes, *i.e.* a whole class of concrete counterexample traces. The problem with traces of concrete states is that it can be hard for the system designer to identify the source of the problem, because of the arbitrary values given to the numerical constants. For instance, maybe a state of the trace is such that $x = 13.76$, while PDR would output a more general information such as $10 \leq x \wedge x \leq 16$ and provide the designer with more information. Notice that invariants found by other methods are also valuable when looking for a counterexample as they can block cubes corresponding to spurious counterexamples earlier.

CHAPTER 9

HullQe: QE And Convex Hull Computation

We now present our new potential invariant generation heuristic, HullQe. This work has been published at FTSCS 2012 [16] and selected for a journal paper in a special issue of Science of Computer Programming currently being reviewed. It builds on a backward property-directed reachability analysis. We use QE to compute successive pre-images of the negation of the PO, in the spirit of [30, 25].

In our approach, the states characterized by the pre-images are generated in such a way that

- they satisfy the PO and
- from them, it is possible to reach a state violating the PO if certain transitions are taken.

Such states will be referred to as *gray states*. In practice we calculate the pre-images as follows:

$$\begin{aligned} \mathcal{G}_1(s) &\triangleq QE(s', \mathrm{PO}(s) \wedge T(s,s') \wedge \neg\mathrm{PO}(s')) \\ \mathcal{G}_i(s) &\triangleq QE(s', \mathrm{PO}(s) \wedge T(s,s') \wedge \mathcal{G}_{i-1}(s')) \quad (\text{for } i > 1). \end{aligned} \tag{9.1}$$

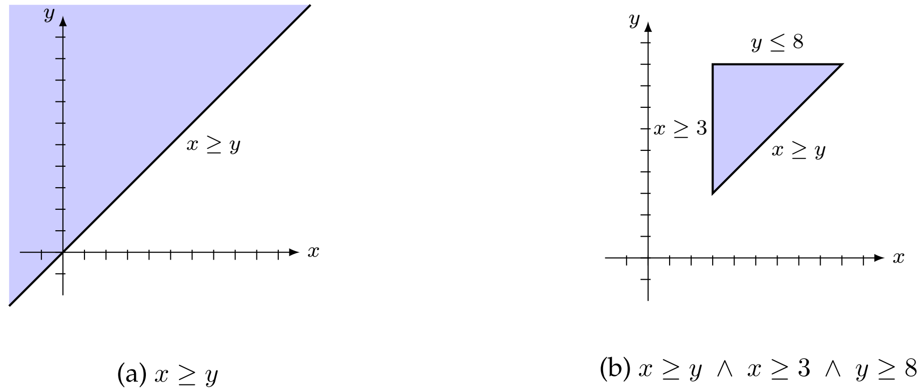
Notation. Let us assume that the QE procedure used in this chapter outputs DNF formulas – this is the case of the technique presented in Chapter 7. Each pre-image is hence a disjunction of cubes. In this chapter we take a more geometric view on formulas: cubes will be seen as *not necessarily closed polyhedra*, henceforth simply polyhedra. For instance $x \geq y$ (see Figure 9.1a) or $x \geq y \ \wedge \ x \geq 3 \ \wedge \ y \geq 8$ (see Figure 9.1b). Polyhedra can also have Boolean or real literals. Let p_1 and p_2 be two polyhedra; we talk about

- "the union of p_1 and p_2", noted $p_1 \cup p_2$, for $p_1 \vee p_2$;
- "the intersection of p_1 and p_2", noted $p_1 \cap p_2$, for $p_1 \wedge p_2$;
- "p_1 is included in p_2", noted $p_1 \subseteq p_2$, for $p_1 \Rightarrow p_2$.

So, each pre-image is a union of polyhedra. Given two polyhedra p_1 and p_2, the Inexact Convex Hull (ICH) of p_1 and p_2 is the smallest[1] polyhedron p such that $p_1 \cup p_2 \subseteq p$. The Exact

[1] *Smallest* in the sense that any for any polyhedron p such that $p_1 \cup p_2 \subseteq p$, if h is the ICH of p_1 and p_2 then $h \subseteq p$.

Convex Hull (ECH) of p_1 and p_2, if it exists, is a polyhedron p such that $p_1 \cup p_2 \subseteq p \subseteq p_1 \cup p_2$. The Parma Polyhedra Library [2] for instance can compute ICH and returns a Boolean flag indicating whether they are in fact ECH or not.



(a) $x \geq y$

(b) $x \geq y \ \wedge \ x \geq 3 \ \wedge \ y \geq 8$

Figure 9.1: Polyhedra examples.

In our approach, the backward analysis is not ran to a fixed point before proceeding further, it is rather meant to probe the gray state space *incrementally* around the negation of the PO. Two search heuristics based on convex hull computations work on the pre-images found so far. They are introduced and motivated in Section 9.1 and further detailed in Section 9.2. These heuristics run in parallel, alongside the backward analysis computing the next pre-image. We show that they are relevant for the verification of design patterns commonly encountered in functional chains in Section 9.3. Last, we report on the combination of Stuff with the abstract interpreter presented in [65] for the analysis of simplified functional chain design in Section 9.4.

9.1 Extracting Potential Invariants From Pre-Images

To extract information out of the pre-images at any point of the backward exploration, their disjunction is considered: it represents the gray states found so far as a union of polyhedra. The main idea underlying the potential invariant generation is to explore the ways in which those polyhedra can be grouped using convex hull calculation, thus discovering linear relations over state variables representing boundaries between convex regions of the gray state space. Since these convex boundaries enclose "bad" states, they are negated before being sent to the k-induction engine to check their validity and try to strengthen the PO. This is illustrated on Figure 9.2 as a refinement of Figure 8.1.

Note that, in the backward exploration, the choice of which state variables to eliminate by QE and which to keep is important. Eliminating the next state variables and keeping the current state variables is not satisfactory in the general case, as on large scale systems,

many state variables might not be relevant for the PO under investigation, and might hinder the performance of the convex hull calculation or k-induction. Therefore, the only state variables that are **not** eliminated are the ones found in the cone of influence of the PO, in their *current state* version. In particular, the system inputs are eliminated since they do not provide more information from a backward analysis point of view. Indeed, the inputs can take any value at any step regardless of the values of the memories of the system. When computing the pre-image of a predicate there must exist some values for the inputs leading to said predicate; what these values are is irrelevant for the computing the next pre-image.

Before going into the details of the potential invariant generation algorithm, let us illustrate how computing ICHs and ECHs can actually make new numerical relations appear, using the examples given in Figure 9.3a and Figure 9.3b.

Figure 9.3a represents the gray state space of a system with two integer state variables. States are represented as dots, polyhedron s_1 contains three states, polyhedra s_3 and s_5 only contain one state *etc*. Computing exact convex hulls over these base polyhedra in the LIA fragment yields (at least) two new borders, *i.e.* potential relational invariants, pictured as dashed lines. An example of merging order is to merge s_1 with s_2, s_3 with s_4, $\{s_1, s_2\}$ with $\{s_3, s_4\}$, and $\{s_1, s_2, s_3, s_4\}$ with s_5 (1).

On a system with real valued state variables however, as shown in Figure 9.3b, the only case in which we will *discover* a new border by computing exact convex hulls is when one is the limit of another, as illustrated on Figure 9.3b. Here $s_1 \triangleq (0 \leq x) \wedge (0 \leq y \leq 2) \wedge (y + x - 4 < 0)$ and $s_2 \triangleq (0 \leq y \leq 2) \wedge (y + x - 4 = 0)$, so the resulting hull will be $(0 \leq x) \wedge (0 \leq y \leq 2) \wedge (y + x - 4 \leq 0)$. The information learned this way has little chance of strengthening the PO.

As we will see in the next sections, when trying to discover new relations, ECH-based techniques work best for integer valued systems, while ICH can be beneficial for both real or integer valued systems.

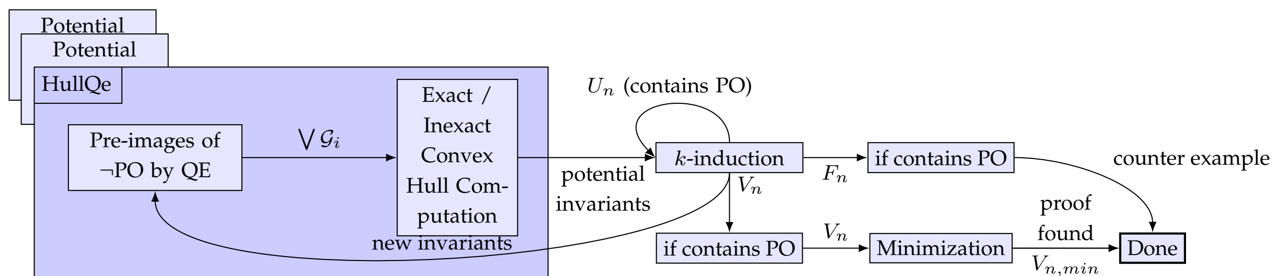
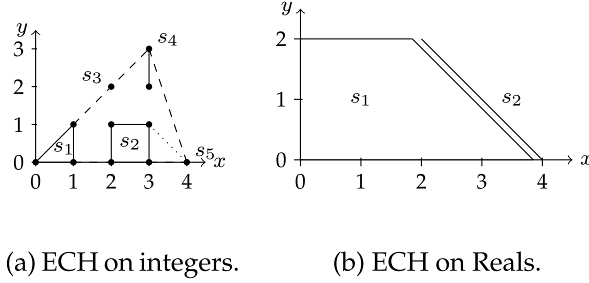


Figure 9.2: HullQe abstract view.



(a) ECH on integers. (b) ECH on Reals.

Figure 9.3: Discovering new relations using convex hulls.

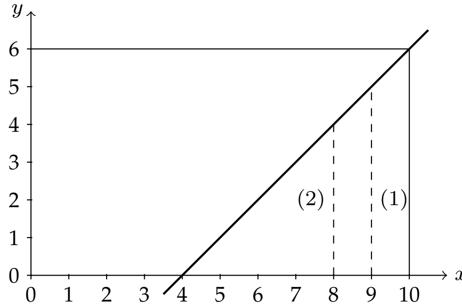


Figure 9.4: ECH calculation on the double counter with $\mathbf{n_x} = 10$ and $\mathbf{n_y} = 6$.

9.1.1 A First Example

We consider the double counter[2] system introduced in Example 6.2.1 with two integer state variables x and y and three Boolean inputs a, b and c. Variables x and y are initialized to 0, and are both incremented by one when a is true or keep their current value when a is false. The variable x is reset if $b \vee c$ is true, and saturates at $\mathbf{n_x}$. The variable y is reset when c is true, hence y cannot be reset without resetting x, and saturates at $\mathbf{n_y}$, and last $\mathbf{n_x} > \mathbf{n_y}$. The proof objective is $x = \mathbf{n_x} \implies y = \mathbf{n_y}$. Here is a possible transition relation for this system:

$$\begin{array}{rlllll} T(s, s') \triangleq & \big(\text{if } (b' \vee c') & \text{then } x' = 0 \text{ else if } (a' \wedge x < \mathbf{n_x}) & \text{then } x' = x + 1 & \text{else } x' = x\big) \\ \wedge & \big(\text{if } (c') & \text{then } y' = 0 \text{ else if } (a' \wedge y < \mathbf{n_y}) & \text{then } y' = y + 1 & \text{else } y' = y\big). \end{array}$$

Note that the proof objective is not k-inductive for any k on this system. As k-induction alone cannot realize that $0 \leq x \leq \mathbf{n_x}$ and $0 \leq y \leq \mathbf{n_y}$ are invariants, a step counter example can always be constructed:

	0	1	$\cdots$	$k-1$	k
x	$\mathbf{n_x} - k$	$\mathbf{n_x} - k + 1$	$\cdots$	$\mathbf{n_x} - 1$	$\mathbf{n_x}$
y	$\mathbf{n_y} - k - 1$	$\mathbf{n_x} - k$	$\cdots$	$\mathbf{n_y} - 2$	$\mathbf{n_y} - 1$

[2]Code available at https://cavale.enseeiht.fr/redmine/projects/smt-qe/files.

Using Abstract Interpretation or a template based invariant discovery technique such as BrutalIQe, the ranges of x and y are easily discovered. With these bounds, the PO becomes $\mathbf{n_y}$-inductive[3]: the step counter example trace given above is blocked by the fact that $y_0 = \mathbf{n_y} - k - 1 = \mathbf{n_y} - \mathbf{n_y} - 1 = -1$ violates the invariant $0 \leq y$. In practice however, even on such a simple system k-induction will never reach a conclusion for large values of $\mathbf{n_x}$ and $\mathbf{n_y}$, for instance thousands, as it implies unrolling the system on $\mathbf{n_y}$ transitions.

Let us see now how our approach performs on this system when fixing $\mathbf{n_x} = 10$ and $\mathbf{n_y} = 6$ for instance. Using the ranges on x and y once k-induction has confirmed them, we start the backward property-directed analysis, which outputs a first pre-image: $x = 9 \ \wedge \ 0 \leq y < 5$ (1). Unsurprisingly, it is too weak to conclude, *i.e.* its negation is not k-inductive for a small k. The next pre-image is $x = 8 \ \wedge \ 0 \leq y < 4 \ \vee x = 9 \ \wedge \ 0 \leq y < 5$ (2) which does not allow us to conclude either. Instead of iterating until a fixed point is found, consider the graph on Figure 9.4. It shows the two first pre-images as dashed lines which seem to suggest a relation between x and y, pictured as a bold line. This relation can be made explicit by calculating the convex hull of the disjunction of the first two pre-images. This yields $8 \leq x \leq 9 \ \wedge \ 0 \leq y < x - 4$. Note that this convex hull is an ECH, since both x and y are integers. The four inequalities are negated – they characterize gray states – and are sent separately to the k-induction engine. Potential invariants $\neg(8 \leq x)$, $\neg(x \leq 9)$ and $\neg(0 \leq y)$ are falsified, and the PO in conjunction with invariant $\neg(y < x - 4)$ is found to be 1-inductive.

We note that on this example, our technique is not sensitive to the actual value of numerical constants: it will always derive the strengthening invariant from the first two pre-images. The time needed to compute the pre-images is not impacted by changing the constants values either.

For more complex systems with pre-images made of more than two polyhedra, simply merging them in arbitrary order using convex hull calculation is not robust since the resulting convex hulls would depend on the merging order, and interesting polyhedra could be missed. Hence the idea of an exhaustive enumeration of the intermediary ECHs that can appear when iteratively merging a set of polyhedra, explored in Section 9.2.1.

9.1.2 A Second Example

Let us now consider briefly a two input (InA and InB), real valued voting logic system derived from the Rockwell Collins triplex voter, the equations of which are available on Figure 9.5. The inputs are supposed to satisfy $|InA| < 0.2 \wedge |InB| < 0.2$. Function *select* returns one of its two arguments in a non-deterministic fashion – it replaces the middle value function used in the original triplex voter. We will not discuss further the system itself since

[3] With the loop-free path constraint.

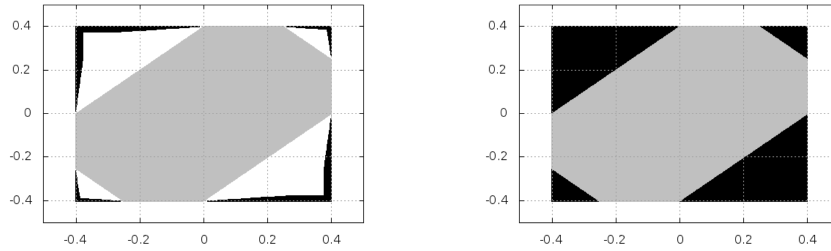
INITIALIZATION

$$
\begin{aligned}
EqualizationA_0 &= 0.0\\
EqualizationB_0 &= 0.0
\end{aligned}
$$

TRANSITION: $\forall k \in \mathbb{N}$,

$$
\begin{aligned}
EqualizedA_k &= InA_k - EqualizationA_k\\
EqualizedB_k &= InB_k - EqualizationB_k\\
EqualizationA_{k+1} &= \ 0.9 \ * EqualizationA_k\\
&\quad + 0.05 * (InA_k + ((EqualizationA_k - Output_k) - Centering_k))\\
EqualizationB_{k+1} &= \ 0.9 \ * EqualizationB_k\\
&\quad + 0.05 * (InB_k + ((EqualizationB_k - Output_k) - Centering_k))\\
Centering_k &= select(EqualizationA_k, EqualizationB_k)\\
Output_k &= select(EqualizedA_k, EqualizedB_k)
\end{aligned}
$$

Figure 9.5: Duplex voter equations.



(a) Two inputs voter, first pre-image. (b) Two inputs voter with ICH.

Figure 9.6: Simple voting logic.

the triplex voter is detailed in Section 9.3.2. It simply allows us to represent graphically the state space in a plane. The PO here is that two of the state variables, $Equalization_1$ and $Equalization_2$, take their value strictly between -0.4 and 0.4. Figure 9.6 depicts the corresponding square. On Figure 9.6a we can see the first pre-image calculated by our backward reachability analysis as black triangles, and the strengthening invariants found by hand in [32] transposed to the two input system as a gray octagon. Calculating ECH on this first pre-image does not allow us to conclude the proof of the PO.

A more relevant approach would be to calculate ICH. Yet, since the ICH of all the pre-image polyhedra is the $[-0.4, 0.4]^2$ square, we need to be more subtle and introduce a criterion for ICH to be actually computed between two polyhedra: they have to intersect. Intersection can be checked by a simple satisfiability test performed by a SMT solver. This check allows us to identify overlapping areas of the gray state space and to over-approximate

them, while not merging disjoint areas in the gray state space explored so far. This approximation obtained through ICH resembles widening techniques used in abstract interpretation [23] in the sense that it can *jump* forward in the analysis iterations. Yet it differs in the sense that, contrary to widening, it does not ensure termination. The only goal here is to generate potential invariants for the PO, and Figure 9.6 shows that the ICH yields exactly the dual, in the $[-0.4, 0.4]^2$ square, of the octagon invariant found by hand in [32]. This second idea of using ICHs to perform over approximations will be discussed in Subsection 9.2.4.

9.2 Exhaustive Exact Convex Hull Enumeration

We now detail two polyhedra merging heuristics outlined in the previous section, which use the pre-images output by the backward state space traversal. The first one follows the example from Section 9.1.1 and consists of a thorough, exact exploration of the partitionings of the gray state space. After explaining the basic algorithm in Section 9.2.1, optimizations are developed in Section 9.2.2. A small example illustrates the method in Section 9.2.3. The second heuristic over-approximates areas of the gray state space in the spirit of the discussion in Section 9.1.2, and is discussed in Section 9.2.4. Both aim at discovering new relations between the state variables which once negated are communicated as potential invariants. Figure 9.8 provides a high level view of the different components and the way they interact internally and with the exterior.

9.2.1 Hullification Algorithm

The algorithm presented in this section, called *hullification*, calculates all the convex hulls that can be created by iterating the convex hull merge attempts on a given set of polyhedra, called the *source* polyhedra. In this algorithm we will calculate Exact Convex Hulls (ECH) as opposed to Inexact Convex Hulls (ICH) – ICH are used in a different less expensive approach in Section 9.2.4, but would prevent very significant optimizations here. The difficulty is to not miss any of the ECH that can be possibly calculated from the source polyhedra while avoiding redundant computations. Indeed, back to the example on Figure 9.3a the merging order (1) misses the ECH of s_2 and s_5 (represented as a dotted line), and consequently the potential relational invariant $y \leq -x + 4$. If n pre-images have been calculated so far, the idea is to iterate until a fixed point is reached on the series of sets

$$\begin{aligned} H_0 &\triangleq \bigcup_{1 \leq i \leq n} G_i \\ H_k &\triangleq \{\mathrm{ECH}(\mathit{pivot}, \mathit{seed}), \quad \mathit{pivot} \in H_{k-1}, \quad \mathit{seed} \in \bigcup_{i \in [1,k-1]} H_i\} \quad k > 0, \end{aligned} \tag{9.2}$$

where G_i is the set of the disjuncts (*i.e.* polyhedra) in the ith pre-image. In the following we will detail our hullification algorithm, which improves this first approach by avoiding

redundant merge attempts.

Imperative and slightly object-oriented pseudo-code is provided in Algorithm 11. The hullification algorithm iterates on a set of pairs called the $generatorSet$: the first component of each of these pairs is a convex hull called the *pivot*. The second one is a set of convex hulls the pivot will be tried to be exactly merged with, called the *seeds* associated with this pivot. The $generatorSet$ is initialized such that for any pair (i, j) such that $0 \leq i \leq n$ and $i < j \leq n$, p_i is a pivot and p_j is one of its seeds, line 3. This corresponds to a more clever version of H_1 in the aforementioned series of sets in that if p_j is a seed of p_i, then p_i is not a seed of p_j. At the beginning of each fixed point iteration (line 6), a $newGeneratorSet$ is initialized with the same pivots as the $generatorSet$ but without any seeds (line 8). The $newGeneratorSet$ is used to construct the next $generatorSet$, with the goal of preventing the many redundant merge attempts done in H_k. At each iteration (line 6), a first loop enumerates the pairs of pivot and seeds of the generator set (line 9). This corresponds to $pivot \in H_{k-1}$ in H_k. Embedded in the first one, a second loop iterates on the seeds (line 11) – corresponding to $seed \in \bigcup_{i\in[1,k-1]} H_i$ in H_k – and tries to calculate the ECH of the current pivot and seed (line 14). If the exact merge was successful, the new ECH is added to the seeds of the pivots of the $newGeneratorSet$ (line 20, detailed below and in Algorithm 12) and as a new pivot with no seeds. Once the elements of the $generatorSet$ have all been inspected and if new ECH have been found, a new iteration begins with the $newGeneratorSet$. When no new convex hulls are discovered during an iteration, the algorithm returns all the ECHs found so far (line 27).

Note that our hullification algorithm cannot find convex hulls that require to merge more than two polyhedra at the same time to be exact since the ECHs are calculated by merging polyhedra two by two.

9.2.2 Optimizing Hullification

The hullification algorithm is highly combinatorial; this section presents optimizations that improve its scalability. The number of merge attempts increases dramatically depending on the number of elements added in the set $newGeneratorSet$ at each iteration. With this version of hullification, potential pivot / seed merges of this set can be redundant, in the sense that the new hulls derived from them, if any, would be the same even though the pivot and seed are different. The key idea to reducing redundancy is to keep a link between any ECH calculated and the source polyhedra merged to create it, thereafter called the ECH source, and use this information to skip redundant ECH calculation attempts. Note that all the optimizations proposed in this section would not work if we were using ICH in the sense that it could cause us to miss some combinations.

Consider for example Figure 9.7a. If we have already tried to merge the ECH of source

Algorithm 11 Hullification Algorithm:
$hullification(\{p_i|0 \leq i \leq n\})$.

1: $generatorSetMemory = \{\{p_i\}|0 \leq i \leq n\}$
2: $sourceMap = \{p_i \implies \{p_i\}|0 \leq i \leq n\}$
3: $generatorSet = \{(p_i, S_i)|0 \leq i \leq n \wedge S_i = \{p_k|i < k \leq n\}\}$
4: $generatorSetMemory = generatorSetMemory \cup \{\{p_i, p_j\}|0 \leq i \leq n,\ i < j \leq n\}$
5: $fixedPoint =$ **false**
6: **while** $(\neg fixedPoint)$ **do**
7: $fixedPoint =$ **true**
8: $newGeneratorSet = \{(p_i, \{\})|\exists S\ (p_i, S) \in generatorSet\}$
9: **for all** $((pivot, seeds) \in generatorSet)$ **do**
10: $sourcePivot = sourceMap.get(pivot)$
11: **for all** $(seed \in seeds)$ **do**
12: $sourceSeed = sourceMap.get(seed)$
13: $source = sourcePivot \cup sourceSeed$
14: $hull = computeHull(pivot, seed)$
15: $newGeneratorSet.update(pivot, newGeneratorSet.get(pivot) - seed)$
16: **if** $(hull \neq$ **false**$)$ **then**
17: $fixedPoint =$ **false**
18: $sourceMap.add(hull \rightarrow source)$
19: $newGeneratorSet =$
20: $updateGenSet(hull, source, pivot, seed, newGeneratorSet)$
21: **end if**
22: **end for**
23: **end for**
24: $generatorSet = newGeneratorSet$
25: ▷ Communication with the rest of the framework can take place here.
26: **end while**
27: **return** $\{p_i|\exists S\ (p_i, S) \in generatorSet\}$

$\{s_1, s_2, s_3\}$ with the one of source $\{s_4, s_5\}$ then it is not necessary to consider trying to merge say the ECH of source $\{s_3, s_4\}$ with the one of source $\{s_1, s_2, s_5\}$. The result would be the same, *i.e.* the same ECH or a failure to merge the convex hulls exactly – the same ECH in this case. Since we are generating all the existing ECHs from the source polyhedra, this happens every time an ECH can be calculated by merging its source in strictly more than one order, that is to say **very** often. More generally, we do not want to attempt merges of different hulls representing the same set of source polyhedra.

Algorithm 12 Updating the $newGeneratorSet$: $updateGenSet(hull, source, newGeneratorSet)$.

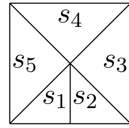
1: $result = \{\}$
2: **for all** $((pivotAux, seedsAux) \in newGeneratorSet)$ **do**
3: $\quad sourceAux = sourceMap.get(pivotAux)$
4: $\quad shallAdd = (sourceAux \cup source) \notin generatorSetMemory$ &&
5: $\quad\quad (sourceAux \not\subseteq source)$
6: $\quad$ **if** $(shallAdd)$ **then**
7: $\quad\quad result.update((pivotAux, seedsAux \cup \{hull\}))$
8: $\quad\quad generatorSetMemory.add(sourceAux \cup source)$
9: $\quad$ **else**
10: $\quad\quad result.add((pivot, seeds))$
11: $\quad$ **end if**
12: **end for**
13: $result.add((hull, \{\}))$
14: **return** $result$

Another source of redundancy is that, when a seed is added to a pivot during the $newGeneratorSet$ update, it represents a potential merge of the union of the pivot source and the seed source. Even if this merge has not yet been considered, a potential merge of the same source might have already been added to the $generatorSet$ through a different seed added to a different pivot. In this case we do not want the seed to be added. So, in order to prevent redundant elements from being added to the $generatorSet$, we introduce a memory called $generatorSetMemory$, and control how new hulls are added to the $newGeneratorSet$. For a new hull to be added to a pivot as a seed, $source(pivot) \cup source(hull) \notin generatorSetMemory$ must be true (Algorithm 12 line 4); if the hull is indeed added to the seeds of the pivot, then the memory set $newGeneratorSetMemory$ is augmented with $source(pivot) \cup source(hull)$ (Algorithm 12 line 8). Informally, this memory contains the sources of all the potential merges added to the generator set. This ensures that the merge of a source will never be considered more than once, and that the merges we did not consider were not reachable by successive pair-wise ECH calculation.

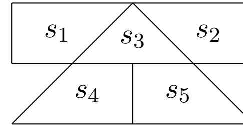
Also, we forbid adding a seed to a pivot's seeds if the source of the latter is a subset of the former, since the result would necessarily be the seed itself. Another improvement deals with *when* hullification interacts with the rest of the framework.

There is a drawback in comparing hulls using their sources: assume that two of the input (source) polyhedron p_i and p_j are such that $p_i \implies p_j$. Then the exact merge of p_i and p_j succeeds and yields the hull of source $\{p_i, p_j\}$, which is really p_j. As a consequence,

the pivots of the $generatorSet$ are redundant, and so are their seeds the merge attempts. To avoid this, we first check the set of input polyhedra and discard redundant ones.



(a) Square example.



(b) Hat example.

Figure 9.7: Hullification redundancy issues.

Last but not least, we present two higher level optimizations. First, merging results are memoized in between calls to the algorithm so that we do not call the merge algorithm when considering two polyhedra we already merged during a previous call. Since hullification is called on the ever-growing disjunction of all pre-images found so far, each new disjunction contains the previous one and this represents a significant improvement. Second, since our goal is to generate potential invariants, we do not need to wait for the hullification algorithm to terminate to communicate the potential invariants already found so far. They are therefore communicated, typically to k-induction, after each fixed point iteration of the algorithm (loop on Algorithm 11 line 25). This has the added benefit of launching k-induction on smaller potential invariant sets.

In the next subsection we illustrate hullification on a small example before introducing another potential invariant generation algorithm in Section 9.2.4. Hullification will be illustrated on a reconfiguration logic system in Section 9.3.1.

9.2.3 Hullification Example

Let us now unroll the algorithm on a simple example depicted on Figure 9.7b. For the sake of clarity a source $\{s_1, s_2, \cdots, s_n\}$ will be written $12 \cdots n$ – with n < 10.
We write generator sets in the following fashion: $\{(pivot, [seeds]), \cdots\}$. With this convention, the initial $generatorSet$ is

$$\{(1, [2, 3, 4, 5]), (2, [3, 4, 5]), (3, [4, 5]), (4, [5]), (5, [])\}.$$

The $newGeneratorSet$ for the first *big step* iteration trace is as follows:

1, []	2, []	3, []	4, []	5, []			
1, []	2, [13]	3, []	4, [13]	5, [13]	13, []		
1, []	2, [13]	3, []	4, [13, 23]	5, [13, 23]	13, []	23, []	
1, [45]	2, [13, 45]	3, [45]	4, [13, 23]	5, [13, 23]	13, [45]	23, [45]	45, []

At first $newGeneratorSet$ is the same as $generatorSet$ without seeds (first line of the trace). We first consider 1 as a pivot. The merge of 1 and 3 works while the other ones fail, leading to the second line of the trace. Note that 13 is not added to 1 nor 3 since $1 \subseteq 13$ and $3 \subseteq 13$. With this pivot we add three sources to the $generatorSetMemory$: 213, 413 and 513 (2). The next pivot is 2 which is merged with 3 while the merges with the other seeds fail. After the $newGeneratorSet$ update we obtain the third line of the trace. Note that 23 is not added to the seeds of 1 since source 213 has already been added to the $generatorSetMemory$ at (2) so $23 \cup 1 \in generatorSetMemory$. Similarly, it is not added to the seeds of 13 either. Next pivot 3 cannot be merged with any of its seeds. Pivot 4 can be merged with 5 producing the fourth line of the generator trace.
A new big step iteration begins during which 2 will be merged with 13 and 3 with 45 while all the other merges will fail. At the beginning of the third big step iteration the $generatorSet$ is

$$\{(1, [345]), (2, [345]), (3, []), (4, [123]), (5, [123]), (13, []), (23, []), (45, [123]), (123, []), (345, [])\}.$$

No new hull is found and the algorithm detects that a fixed point has been reached.

9.2.4 Inexact Convex Hulls

As mentioned before in Section 9.2.1, ECH calculation cannot do much for real state variables. We therefore propose a second approach based on Inexact Convex Hull (ICH) calculation modulo intersection as discussed in Section 9.1.2, simply called ICH calculation in the rest of this paper. That is, two polyhedra will be inexactly merged if and only if their intersection is not empty. This regroups areas of the gray state space that are not disjoint and over-approximates them to make new numerical relations appear. An efficient way to check for intersection is to check the satisfiability of the conjunction of the constraints describing the two polyhedra using an SMT solver. Note that this technique is also of interest in the integer case.

For a given set of polyhedra more ICHs than ECHs can be created, in practice often a lot more. The hullification algorithm using ICHs is thus a very expensive one. We propose the following algorithm, only briefly described because of it high similarity with hullification.
Select a pivot in the input polyhedra set and try to find an ICH with the other ones. If an ICH with another polyhedron (*seed*) exists, both the pivot and the *seed* are discarded, and the ICH becomes the new pivot. Once all the merges have been tried, the pivot is put aside and a new pivot is selected in the remaining polyhedra set. When the algorithm runs out of polyhedra, it starts again on the polyhedra put aside if at least one new hull was found. If not, a fixed point has been reached and the algorithm stops.
Although the ICH computed in this algorithm depends on the order in which the pivots

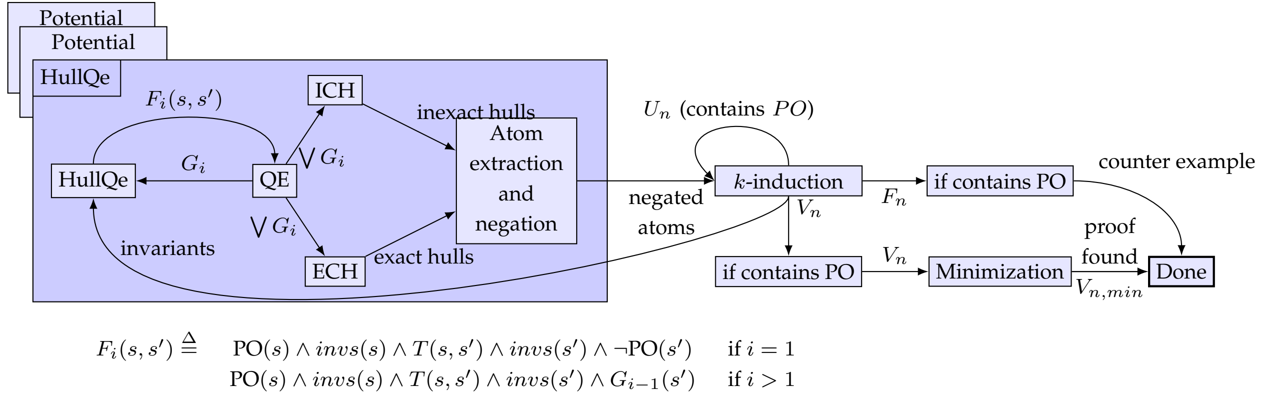


Figure 9.8: View of HullQe internal processes.

are selected and merged with the other polyhedron, its result does not. Indeed, the fact that two polyhedra have a non-empty intersection will stay true even if one or both of them are merged with other polyhedra. This result, as illustrated in Section 9.1.2, is an over-approximation of disjoint areas of the gray state space.

In practice, both the ECH based hullification and the ICH calculation heuristics run in parallel, and the sets of potential invariants they output are sent to the k-induction engine. This allows us to combine the precision of ECHs with the over-approximation effect of ICHs. A high level view of our approach is available on Figure 9.8. The next section will present two examples taken from a functional chain as presented in Section 4.2, each illustrating the ideas introduced in this section: a reconfiguration logic system and a voting logic system.

9.3 Applications

In this section we discuss the proposed invariant discovery techniques on two real world examples[4] a reconfiguration logic and the triplex voter provided by Rockwell Collins. Durations correspond to a typical run of our prototype framework Stuff on a dual-core laptop with four gigabytes of RAM memory.

9.3.1 Reconfiguration Logic

Distributed reconfiguration logic as presented in Section 4.2 would be best described as a distributed priority mechanism. In each redundant channel, the reconfiguration logic comes last and watches the warning flags raised by the monitoring logic implemented earlier in the data flow. Integer timers and latches are used to confirm warnings over a number

[4]Code available at https://cavale.enseeiht.fr/redmine/attachments/download/59/systems.tar.gz.

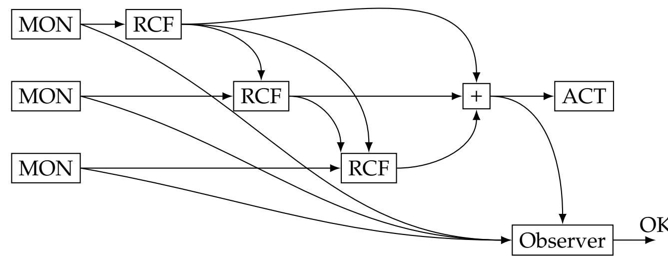


Figure 9.9: Reconfiguration subsystem with observer.

of consecutive time steps and trigger a reconfiguration. The duration of the various confirmations can vary from a few steps to hundreds or thousands of steps and are tuned by system designers to be not overly sensitive to transient perturbations – which would unnecessarily trigger reconfigurations of otherwise healthy channels – while being fast enough to ensure safety. Assuming at most two sensors, network or CPU faults, the following generic property is expected to hold for the reconfiguration mechanism: “No unhealthy channel shall be in control for more than N steps”. This property can be decomposed and instantiated per channel. However, a property such as “No more than one channel shall be in command at any time”, or “The actuator must never stay idle for more than m_4 steps” – where m_4 is an integer given by the specification – are more challenging because they cover all three channels simultaneously and drag many state variables in their cone of influence. For instance, the formal verification of the second property is done by assembling a model of the distributed system and by using the synchronous observer technique as shown in Figure 9.9. The observer uses a timer and is coded so that its output becomes true as soon as the absence of control of the actuator has been confirmed for the requested amount of m_4 consecutive steps. The proof objective on the system/observer composition is to show that the output of this observer can never be false. The code of the system we analyzed is given in Appendix A.

The timer logic found in this system is similar to that of the double counter example, instantiated several times. A channel becoming corrupt triggers several timers with different bounds, running into each other or in parallel. As was the case for the double counter system, ranges on the counters need to be found to prevent spurious counter examples from appearing. BrutalIQe finds them in several seconds. Once this is done, HullQe starts computing the first pre-image which turns out insufficient for hullification to generate potential invariants either strengthening the PO or k-inductive by themselves. The union of the first and second pre-images however allows hullification to generate about 150 potential invariants. Once they are negated, k-induction indicates the PO was found (1-)inductive conjoined

with about 70 invariants after less than 15 seconds of computation.

After the minimization phase, it turns out that only three invariants are required. If we call $timer_i$ the integer variable used to count the time channel i is not in command for $1 \leq i \leq 3$, and $timer_o$ the timer used by the observer, the invariants are: $\neg(timer_o - timer_i \geq m_4 - m_i - 1)$ where $1 \leq i \leq 3$. These invariants are found in the same amount of time no matter the values of the m_i for $1 \leq i \leq 4$. We insist on the interest of enumerating all ECHs. Merging polyhedra in some single arbitrary order is too coarse and the resulting hull cannot strengthen the PO, whereas the thorough exploration generates useful strengthening invariants.

9.3.2 The Triplex Voter

Let us now turn to the Rockwell Collins triplex sensor voter, an industrial example of voting logic as introduced in Section 4.2, implementing redundancy management for three sensor input values: InA, InB and InC. Its code is available in Appendix A This version does not feature fault detection nor reset. This voter does not compute an average value, but uses the $middleValue(x, y, z)$ function, which returns the input value, bounded by the minimum and the maximum input values (*i.e.* z if $y < z < x$). Other voter algorithms which use a (possibly weighted) average value are more sensitive to one of the input values being out of the normal bounds. The values considered for voting are *equalized* by subtracting equalization values from the inputs. The recursive equations in Figure 9.10 describe the behavior of the voter. The role of the equalization values is to compensate offset errors of the sensors, assuming that the middle value gives the most accurate measurement.

We are interested in proving Bounded-Input Bounded-Output (BIBO) stability of the voter, which is a fundamental requirement for filtering and signal processing systems, ensuring that the system output cannot grow indefinitely as long as the system input stays within a certain range. In general, it is necessary to identify and prove auxiliary system invariants in order to prove BIBO stability.

So, we want to prove the stability of the system, *i.e.* we want to prove that the voter output is bounded as long as the input values differ by at most the maximal authorized deviation $MaxDev$ from the true value of the measured physical quantity represented by the variable $TrueValue$. In our analysis, we fixed the maximal sensor deviation to 0.2, a value that domain experts gave us as typical value in practical applications. It is straightforward to prove that the system is stable if the equalization values are bounded.

When applied to Rockwell Collins triplex sensor voter, our prototype implementation manages to prove the PO in less than 10 seconds. Once the first pre-image is computed, ICH calculation outputs about 60 potential invariants, 30 of which are 1-inductive and strengthen

$$
\begin{array}{lll}
\textsc{Initialization} & & \\
\quad EqualizationA_0 & \triangleq & 0.0 \\
\quad EqualizationB_0 & \triangleq & 0.0 \\
\quad EqualizationC_0 & \triangleq & 0.0 \\
\hline
\textsc{Transition:} \quad \forall k \in \mathbb{N}, & & \\
\quad EqualizedA_k & \triangleq & InA_k - EqualizationA_k \\
\quad EqualizedB_k & \triangleq & InB_k - EqualizationB_k \\
\quad EqualizedC_k & \triangleq & InC_k - EqualizationC_k \\
\quad EqualizationA_{k+1} & \triangleq & \quad 0.9 \;\; * EqualizationA_k \\
 & & + 0.05 * (InA_k + ((EqualizationA_k - Output_k) - Centering_k)) \\
\quad EqualizationB_{k+1} & \triangleq & \quad 0.9 \;\; * EqualizationB_k \\
 & & + 0.05 * (InB_k + ((EqualizationB_k - Output_k) - Centering_k)) \\
\quad EqualizationC_{k+1} & \triangleq & \quad 0.9 \;\; * EqualizationC_k \\
 & & + 0.05 * (InC_k + ((EqualizationC_k - Output_k) - Centering_k)) \\
\quad Centering_k & \triangleq & middleValue(EqualizationA_k, EqualizationB_k, EqualizationC_k) \\
\quad Output_k & \triangleq & middleValue(EqualizedA_k, EqualizedB_k, EqualizedC_k)
\end{array}
$$

Figure 9.10: Triplex voter equations.

the proof objective. After minimization, it turns out that invariants

$$-0.9 \leq EqualizationA + EqualizationB + EqualizationC \leq 0.9 \tag{9.3}$$

alone are enough to strengthen the property. Again, the time taken to complete the proof does not depend on the system numerical constants, and the strengthened PO is (1-)inductive. We insist on the importance of these characteristics for both industrials and certification organisms: the proof is trustworthy and can be redone easily for similar, slightly altered designs.

9.3.3 Comparison With Existing Tools

The reconfiguration logic. was also analyzed using NBac, SCADE Design Verifier and Kind. NBac did not succeed in proving the property after 1 hour of computation. Both the SCADE Design Verifier and Kind kept on incrementing the induction depth without finding a proof after 30 minutes of run time. The invariant generation of Kind was also ran on this system, and yielded a number of small theorems, not sufficient to strengthen the PO and prove it.

In conclusion, the proposed combination of backward analysis, hullification and k-induction

allows us to complete a proof in a few seconds on a widely used avionics design pattern, where other state of the art tools fail.

In addition, we see two very interesting points worth highlighting about hullification:

(i) The PO is made (1-)inductive, implying the proof can easily and quickly be re-run and checked by any existing induction tool;

(ii) the time needed to complete the proof does not depend on the numerical values of the system – less than 20 seconds.

This is very important for critical embedded systems manufacturers as point *(i)* means that the proofs are trustworthy, both for the industrials themselves and the certification organisms. On the other hand, point *(ii)* implies that strengthening invariants can be very quickly generated for similar design patterns with altered numerical values, easing the integration of formal verification in the development process. Indeed, it avoids the need for an expert to manually transpose the invariants on the new system, as can be the case for complicated and resource/time consuming proofs.

The Rockwell Collins triplex voter. (without fault detection nor reset) was already proven stable in [32], but the necessary invariants had to be found by hand after the Scade Design Verifier, Kind as well as Astrée (which was run on C-Code generated from the Lustre source) failed at automatically verifying the BIBO property.

Also, once the strengthening invariant 9.3 has been found, a template can easily be inferred:

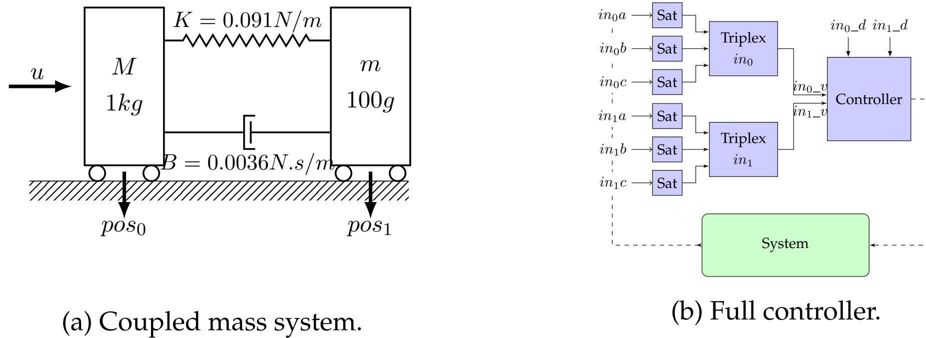
$$\begin{gathered} template(min, max) \triangleq \\ min \leq EqualizationA + EqualizationB + EqualizationC \leq max \end{gathered} \tag{9.4}$$

This template is already instantiated on $EqualizationA$, $EqualizationB$ and $EqualizationC$ since HullQe already found that one such strengthening invariant exists. Using BrutalIQe, we can find values for min and max making the template strengthening again very quickly. Although it might seem vain to rediscover an invariant we already know, it can be interesting in the context of verification integrated to a development process. The system on which we found this strengthening invariant might not be the final version. Likewise, the proof objective(s) can be changed at some point, *i.e.* the specification has been revised. In a later version of the system or of the specification, it might be the case that (9.3) does not strengthen the proof objective, while still being an invariant. This invariant can hence be (re)discovered by our template-based approach. It might be enough to conclude on the new system, or it might guide HullQe in its backward exploration towards finding a new strengthening invariant. On the example presented in this section for instance, it could be the case that the specification is changed from "each equalization value ranges between . . . "

to "the output ranges between ...". Were it to happen, (9.3) would still be a strengthening invariant and the proof could be concluded very quickly.

9.4 End-to-end Verification of a Functional Chain Using a Combination of Linear and Non-Linear Analyses

Thanks to HullQe, our framework can verify reconfiguration and voting logic systems commonly found in avionics critical embedded system. The command law of the system on the other hand models the controller reacting to the environment, and verification of the *open loop controller stability* – *i.e.* a BIBO stability for the command law – typically require non-linear invariants. The following is joint work with the authors of [65] and has been published in FIMCS 2013 [17] and received the best paper award. We show how to combine our approach with an abstract interpreter able to infer non-linear invariants to verify a full functional chain using a running example as a command law. We consider the control of the coupled mass system, extracted from [66], shown in Figure 9.11a. Such a coupling can be used to model physical phenomena such as vibration propagation patterns in fluids and flexible structures, among others. The system is presented on Figure 9.11b and is **not** a triplicated architecture. There is hence no reconfiguration logic in it.



(a) Coupled mass system.

(b) Full controller.

Figure 9.11: Running example.

For this running example, we can identify the following contracts:

- the system should ensure that the output on data flow u does not exceed the capability of the moving mass m_1 hardware actuator. For example, if the maximal capability is $200N$, we should ensure that $|u| \leq 200$;
- the controller itself has specific system-level properties as described above. In this case we only consider the BIBO property on the output u;

- the triplex voter contract is that given bounds on its inputs, it guarantees bounds on its output. It is also a BIBO property;
- the Sat nodes correspond to the validator nodes evoked above for the replicated sensors, a typical specification would be that $lb \leq o \leq ub$ where o is the output flow, lb and ub are lower and upper saturation bounds.

These requirements should be satisfied by the blocks and their later implementations, and each of them should be formally expressed in a logic language format that can be parsed and processed by available tools. This running example is simple but representative of control command software, and its analysis is meaningful since most of these properties are not currently analyzable at model or code level with existing formal tools.

Using the AI approach based on ellipsoids discussed in [17] in conjunction with our formal framework Stuff, it is possible to perform an analysis of the complete system presented in Figure 9.11b. The analyzers run in parallel and communicate their results to each other. We only assume the following knowledge:

- the **Triplex** node description is fitted with the BIBO contract

$$\begin{aligned} BIBO(a) \equiv \quad & \forall k \in \mathbb{N},\ \ |InA_k| \leq a \wedge |InB_k| \leq a \wedge |InC_k| \leq a \implies |Output_k| \leq 3a \wedge \\ & |EqualizationA_k| \leq 2a \wedge |EqualizationB_k| \leq 2a \wedge |EqualizationC_k| \leq 2a; \end{aligned}$$

- the **Controller** node is known to be a linear open stable controller; its internal variables are automatically considered for analysis with our quadratic templates analysis.

We would like to guarantee that the system does not diverge whatever the inputs are. This contract could either be implicit, *i.e.* no overflow, or specific, *e.g.* the output should satisfy a given constraint $|u| \leq 200(N)$. We recall that – up to our knowledge – none of these specifications is provable by any academic nor commercial tool available except ours. Let us describe the sequence of analyses needed to achieve the global proof:

1. Using abstract interpretation, **Sat** node outputs are bounded to their saturation value of 1.2;
2. These bounds enable the instantiation of the **Triplex** contracts. HullQe with k-induction generates the missing invariants and proves the contracts, effectively bounding the output of both voters by 3.6. The analysis is fully automatic once the proof objective is given.
3. Once the inputs of the controller node are bounded (*i.e.* voter outputs), an ellipsoid-based AI analysis begins: the control flow graph is computed, the quadratic template is synthesized and the policy iteration is performed, bounding the internal variables

and the output u of the controller: $|u| \leq 194.499$. The analysis is fully automatic and only requires the list of internal variables of the controller as well as bounds on the inputs.

Finally the global contract can be checked. The quadratic invariant synthesized by abstract interpretation satisfies the required bound of $200N$.

PART III

The Stuff Formal Framework

All the techniques discussed in this thesis are implemented in our formal framework Stuff (**S**tuff's **T**he **U**ltimate **F**ormal **F**ramework) written in Scala. Stuff analyzes Lustre programs which are first translated to a transition system representation. The analysis follows the collaborative k-induction-based scheme presented in Chapter 8. Invariant discovery techniques run in parallel and the k-induction engine confirms actual invariants, until the proof objective is proven or falsified.

For parallelism, Stuff relies on the actor paradigm thanks to the Akka actor library. Actors communicate with each other by means of an asynchronous message passing protocol. Extensibility is a crucial concern in Stuff. First, the high level architecture of the framework allows addition of methods with virtually no impact on existing code. The *method laboratory*, or MethLab, maintains the context of the analysis: potential invariants and proof objectives are split in valid, invalid, and unknown categories as the analysis goes. The MethLab interacts with the techniques through a well-defined interface which specifies how methods impact the context of the analysis, and hides the implementation details of the techniques. Stuff is also extensible in the solvers it can use. Thanks to the Assumptio [14] actor-oriented solver wrapper, the SMT solvers handling the satisfiability queries can be swapped without impact on the client code using them. We present the actor paradigm briefly and the MethLab architecture in Chapter 10 and show that our data structure for transition systems and unrolling algorithms respect the semantics defined in Part I. We then discuss Assumptio, a wrapper for SMT-LIB 2 compliant SMT solvers in Section 11.

CHAPTER 10

Architecture and Transition System Representation

In this chapter we discuss the high level architecture of Stuff. We begin by presenting briefly the actor paradigm and more precisely the Akka actor framework in Section 10.1. We then expose the high level extensible architecture of Stuff in Section 10.2. Last, we discuss the implementation of transition systems as well as our unrolling algorithms in Section 10.3

10.1 The Actor Formalism

The *actor formalism* as introduced in [39] is a general model to describe and reason about programs, and in particular concurrent programs. There are two basic entities, *actors* and *messages*. Actors have an internal state including a list of messages: their mailbox. They send messages – which can contain data – to each other. Upon reception of a message an actor can do any of the following, as long as it terminates in finite time: *(i)* send messages to other actors, *(ii)* change its internal state, and/or *(iii)* create actors. When an actor changes its internal state, it can in particular change the way it reacts to messages.

The Akka framework is an implementation of the actor formalism and provides a nice programming paradigm to parallelism as a Scala API [75]. Communication is asynchronous, there is no shared data between actors, and the abstract nature of the paradigm allows actors to be deployed on different cores or in a distant data center. In fact, deployment can be specified after compilation in a configuration file. As a result, developing a parallel application in Akka allows its users to focus on the behavior of each actor as the parallelism details are abstracted away. Each Akka actor has its own mailbox which is a queue of messages. When an actor is done handling a message it is suspended by a scheduler, also responsible for awaking actors when their mailbox contains a message. The scheduler is fair: any actor with a non-empty mailbox is awoken at some point in the future. In practice Akka actors differ from the formalism in that they can loop forever when reacting to a message, and can crash due to programming bugs. The Akka framework provides means to allow systems to be resilient to errors.

Messages are of type `Any`, which is a supertype of all types in Scala. Message are processed by a partial function of type `Any` to `Unit`, the `Receive` type in Akka. The partial function matches the messages against different patterns, and executes the appropriate computation of type `Unit`. For instance:

```
class SimpleActor extends Actor {
  def receive: Receive = {
    case s: String =>
      println("Received string " + s + " from actor " + sender + ".")
    case i: Int =>
      println("Received int " + i + " from actor " + sender + ".")
    case msg: MessageDefinedSomewhere => work(msg)
  }
}
```

Messages trigger reaction(s), after which the actor is put to sleep until a message arrives in its mailbox, to be handled by the `receive` partial function. Messages not caught by the partial function end up in an `unhandled` function. In `sender` resides the *address*, or `ActorRef` in Akka, of the actor which sent the last message received.

The actor formalism is concise, powerful and flexible. Let us describe an incremental k-induction engine using pseudo (Scala) code and Akka actors. Assume that function `kInduction` attempts to find a k-induction proof up to a given depth sequentially, and returns either `Falsified(k)`, `Valid(k)` or `Unknown`. Then k-induction can be thought of as an actor which receives a job descriptor containing a system, a proof objective and a maximal k value. In response, the k-induction actor sends a textual notification once function `kInduction` terminates:

```
class KInduction extends Actor {
  val solver = ...

  /** Runs a k-induction analysis from k to job.maxK. */
  def kInduction(job: KInductionJobDescriptor, k: Int): KinductionResult =
    if (k > job.maxK) Unknown else solver.checkSat(constructBase(job,k)) match {
    case Sat   => Falsified(k)
    case Unsat => checkSat(constructStep(job,k)) match {
      case Sat   => kInduction(job,k+1)
      case Unsat => Valid(k)
    }
  }

  /** Performs the actual work. */
  def work(client: ActorRef, job: KInductionJobDescriptor) = kInduction(job,1)
      match {
```

```
    case Falsified(k) => client ! ("PO falsified in " + (k-1) + " transitions.")
    case Valid(k)     => client ! ("PO is " + k + "-inductive.")
    case Unknown      => client ! ("PO is not " + job.maxK + "-inductive.")
  }

  /** Defines the actor reaction to the message. */
  def receive: Receive = {
    case job: KInductionJobDescriptor => work(sender,job)
  }
}
```

Note the "`!`" Akka operator used for sending messages to another actor. Actors can also modify their internal state. The k-induction actor could maintain a list of invariants to be asserted on each state of the step instances in order to reduce the search space. New invariants are communicated to the k-induction actor *via* the `NewInvariants` message:

```
class KInduction extends Actor {

  def kInduction(
    job: KInductionJobDescriptor, invs: List[Invariant], k: Int
  ): KinductionResult = ...

  var invariants: List[Invariant] = Nil

  def work(client: ActorRef) = kInduction(job,invariants, 1) match { ... }

  def receive: Receive = {
    case job: KInductionJobDescriptor => work(sender)
    case NewInvariants(invs) => invariants ++= invs
  }
}
```

In Stuff the k-induction implementation is composed of three actors. First, an actor handles the base instance, while a second takes care of the step instance. Both unroll the transition relation separately and communicate their results to the third actor. This third actor is responsible for communicating with the rest of the framework and drawing conclusions on the state of the k-induction analysis. In order to benefit from parallelization as much as possible, base and step should increment the unrolling length independently. If one of the instances is going faster than the other, synchronizing the transition relation unrolling would slow it down.

More precisely, the base actor keeps on unrolling further as long as the base instance is unsat. Message `BaseUnsat(k)` (*resp.* `BaseSat(k)`) indicates that the base instance with $k-1$ unrolling of the transition relation is unsat (*resp.* sat). The step actor keeps on unrolling as long as the step instance is sat, and sends similar messages. To draw conclusions from

the messages the new actors can send to the k-induction actor, we define a new message behavior for the `KInduction` class:

```
class KInduction extends Actor {
  /** @param baseUnsatAt Last k such that the base instance is unsat (default
      -1).
    * @param stepDoneAt  Default is -1: step is not done.
    *                    Modified when a StepUnsat(k) message is received. */
  def runningBehavior(
    client: ActorRef, baseUnsatAt: Int = -1, stepDoneAt: Int = -1
  ): Receive = {
    case BaseSat(k)   =>
      client ! ("PO falsified in " + (k-1) + " transitions.")
    case BaseUnsat(k) =>
      if (stepDoneAt > 0 && k >= stepDoneAt)
        client ! ("PO is " + stepDoneAt + "-inductive.")
      else context become runningBehavior(client, k, stepDoneAt)
    case StepSat(k)   => () // Unit, i.e. nothing.
    case StepUnsat(k) =>
      if (baseUnsatAt >= k) client ! ("PO is " + k + "-inductive.")
      else context become runningBehavior(client, baseUnsatAt, k)
    case StepReachedMaxK => ...
    case BaseReachedMaxK => ...
  }
```

If at some point the base instance is sat (line 8) then the proof objective does not hold. If the step instance is found unsat for some k, then if the base instance is also unsat for some l greater than or equal to k (line 16) the proof objective holds. If the base instance has not reached k yet, the function memorizes k (line 17) and waits for base to reach it (line 11) – or to find a counterexample (line 8).

Notice the two `context become` calls at lines 12 and 17. They change the behavior for the reception of the next messages, until it is changed again. This is to keep track of information[1]: the highest k for which the base instance is unsat, and an integer indicating if step is unsat. Both are used to make sure step and base are unsat at coherent unrolling length before claiming that the proof objective is valid. We do not include code for the case when step and base reach the bound on k as it is rather simple but impairs readability. We now modify `receive` to spawn the new actors, start them and switch to the running behavior.

```
class KInduction extends Actor {
  def receive: Receive = {
    case job: KInductionJobDescriptor => {
      val base = spawn(new Base(job))
      val step = spawn(new Step(job))
```

[1] It is a side effect-free way for an actor to "*(ii)* change its internal state".

```
      base ! Start
      step ! Start
      context become runningBehavior(client,-1,-1)
    }
  }

  def runningBehavior(
    client: ActorRef, baseUnsatAt: Int = -1, stepDoneAt: Int = -1
  ): Receive = { ... }
```

The base and step actors perform a series of satisfiability checks, unrolling further and further the transition relation, sending a message at each step. Code for the base actor follows – we omit code for the step actor as it is highly similar.

```
class Base(job: KInductionJobDescriptor) extends Actor {
  def receive: Receive = {
    case Start => work(1)
  }

  val solver = ...

  def work(k: Int): Unit =
    solver.checkSat(constructBase(job,k)) match {
      case Sat => sender ! BaseSat(k)
      case Unsat => {
        sender ! BaseUnsat(k)
        if (k < job.maxK) work(k+1, And(toCheck,
            transition(state(k-1),state(k))))
        else sender ! BaseReachedMaxK
      }
    }
}
```

In Stuff, techniques are implemented as one or more actors. K-induction in particular follows closely the scheme presented in this section: base and step are two actors supervised by a third one also handling communication with the rest of the framework. The other techniques follow a similar architecture: a *master* actor handles communication with the rest of the framework and supervises other actors performing the computations needed. In our implementation of HullQe for instance, the master actor supervises the pre-image computation actor, the two convex hull computation actors – for ECH and ICH – and an actor responsible for checking if a fixed point of the pre-image computation has been reached.

10.2 Stuff: Architecture

Extensibility consists, informally, in being able to add features without modifying existing code. Extensibility with respect to the techniques implemented is an important concern in Stuff: it should be possible to add or modify a technique without changing the other ones. Indeed, modification of existing code is time consuming and error-prone. A stricter definition of extensibility is to be able to add features without recompiling existing code. For instance, the Akka actor library is extensible in this sense for the deployment of its actors, and so is Stuff. This is also the case of the solvers used by the techniques in Stuff, as discussed in Chapter 11. On the other hand, implementing a new technique in Stuff does require to re-generate the Stuff byte code executable. But since it does not impact the code of existing techniques written in separate Scala files, the Scala compiler does not recompile the other techniques.

In Stuff the context of the analysis – invariants found, falsified proof objectives, *etc.* – and communication between techniques is handled by the *method laboratory* actor, or MethLab. The MethLab is only aware of the techniques existing in the framework through the interface defined in the `Method` trait[2]:

```
trait Method {
  def launch(): Unit
  def kill(): Unit

  def handleMsg(message: Any, methodsLeft: List[Method]): Unit

  def newInformation(
    newValidPOs: Exprs, newFalsifiedPOs: Exprs,
    newValidLemmas: Exprs, newFalsifiedLemmas: Exprs,
       newPotentialLemmas: Exprs
  ): Unit
}
```

When launching Stuff the user specifies which techniques should be used in the analysis, and the MethLab is instantiated with the corresponding list of `Method`s.

When implementing a new technique, the developer must also provide its `Method` in the same compilation unit. It acts as a correspondent for the technique on the MethLab side and is responsible for starting and terminating the execution of its technique. It must also be able to handle the messages sent by the technique thanks to the function `handleMsg` which follows the *chain of responsibility* paradigm illustrated on Figure 10.1. It first checks

[2] A Scala *trait* is similar to a Java interface that can be partially implemented.

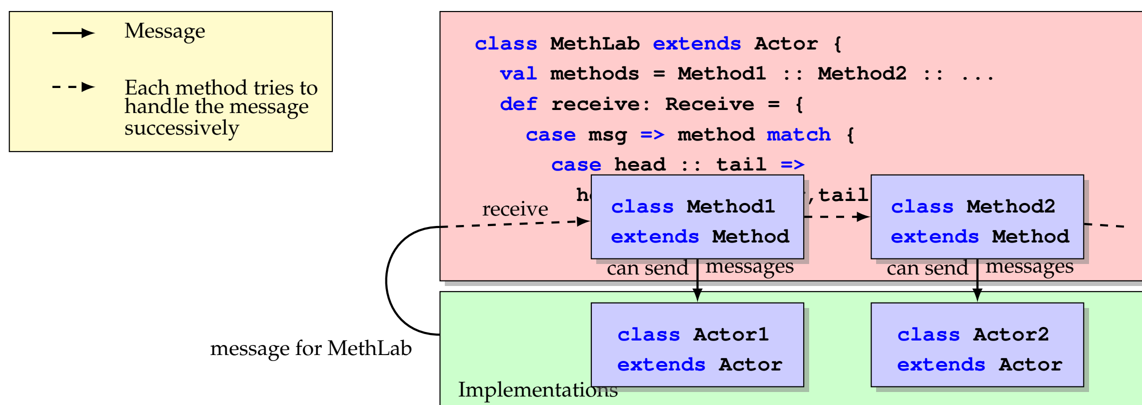


Figure 10.1: Message handling.

if the message comes from the technique it is associated with. If it is not the case, it calls `handleMsg` on the next `Method` in the list and so on until one is able to handle the message. In practice, the methods which have not tried to handle the message yet are passed as an argument to `handleMsg`: the `methodsLeft` argument. Thus, integration of a new technique depends solely on the contract of the `Method` trait and on the structure of the context, *i.e.* how the information established so far is accessed to and modified.

When handling a message, a `Method` can modify the context of the MethLab, send messages to its implementation actor, and / or notify other `Method`s of new invariants, potential invariants... For instance, the k-induction actor could send some new invariants it discovered, and its `Method` would update the context and send a new k-induction job on some new potential invariants. It would also notify the other methods that something new was discovered. To allow this, methods must implement the function `newInformation`.

When something new is discovered by a method, it will call this function on all other methods, which will forward the information to their respective implementation or ignore it. For instance, the HullQe method would forward new invariants discovered by k-induction to its implementation to constrain the pre-image computation. But it would ignore falsified potential invariants.

Figure 10.2 shows a code extract of the `handleMessage` function in the k-induction `Method`. When new invariants are received, the context is updated and each method is notified of the new information. Notice the guard when matching a `KindValidPOs` message. The message is caught only if the sender is indeed the instance associated to the method. This allows users to run two or more versions of the same technique: without this guard, the first method to catch the message would not realize it does not belong to it. Running

```
class KInductionMethod extends Method {
  def handleMessage(message: Any, methodsLeft: List[Method]) = message
     match {
    case msg: KindValidPOs if sender == instance => {
      msg.pos foreach (
        po => context.addValid(
          po, "proved by " + msg.prefixSize + "-induction", msg.pos - po
          //         Proved in conjunction with invartiants ^^^^^^^^^^^^
        )
      )
      methods foreach (meth => if (meth != this)
          meth.newInformation(...))
    }
    ...
  }
```

Figure 10.2: A code extract of the `handleMessage` function of the k-induction `Method`.

two incremental k-inductions for instance can be rewarding. Invariant discovery techniques such as HullQe can generate many new potential invariants all the time. A first k-induction instance going up to a relatively small k is often enough to refute or prove most of them quickly. It is reactive in that potential invariants discovered by the other techniques that are easy to refute or prove do not have the time to accumulate too much in-between runs. A second k-induction instance goes up to a much larger k to analyze potential invariants that might require to unroll the transition relation further to be refuted or verified. The first (fast) instance allows the second (slow) instance to focus on difficult proof objectives by setting them apart.

The actor architecture of Stuff for k-induction and HullQe is presented on Figure 10.3. Circles represent actors or system processes. Colors identify actors and processes belonging to the same technique, while connections between actors and processes indicate bi-directional communication channels. Assumptio is a solver wrapper developed for Stuff; actors using an SMT solver have their own Assumptio instance which maintains the actual SMT solver in a separate system process. Assumptio will be discussed in the next chapter. Thanks to this architecture and the Akka actor library, Stuff can take advantage of multi-core and many-core computers and can even be deployed over a distributed architecture. We end this chapter by discussing the transition system representation and unrolling algorithms in Stuff in the next section.

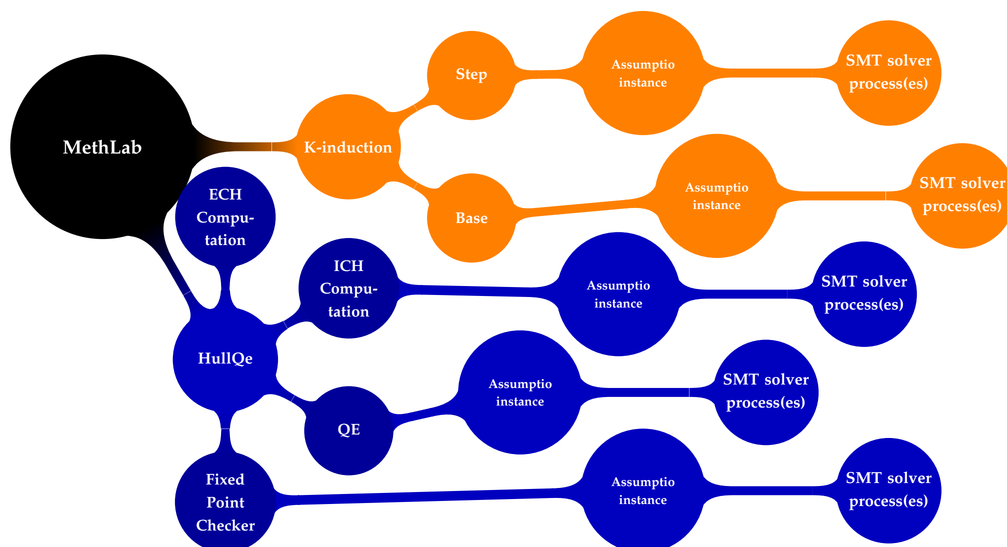


Figure 10.3: Actor hierarchy in Stuff.

10.3 Stream Systems and Unrolling

In our presentation of transition systems in Section 6.1 we differentiated between state variables (s), unrolled state variables (s_0, s_1, ...) which give a temporal meaning to state variables, and states, *i.e.* valuation of state variables. This section shows how our implementation respects this formalism thanks to *stream systems* to represent the transition system, and unrolling to construct traces of unrolled state variables.

A compilation phase transforms Lustre programs to transition systems represented in Stuff as stream systems. This structure is a straightforward extension of the classical *Boolean sequential circuit graph data structure* to the SMT case. It aggregates *streams* which correspond to state variables. This vector of streams has a dataflow semantics, *i.e.* its models are exactly all sequences of states satisfying the transition relation of the system. Once unrolled to length k, the vector of streams becomes a vector of length k of vector of *unrolled streams*, the models of which are concrete states. Stream systems also contains the constraints on the system – if any – and proof objectives, all as expressions over streams. Each stream is declared as a pair $\langle id, \sigma \rangle$ where id is an identifier and σ is a sort. Streams can be defined as an expression e over other streams, and can specify an initial expression $init$. More precisely, there are three types of streams.

First, *input streams* or *ins* are streams with no definition. Indeed, the input flows of a

transition system are *a priori* not constrained. It is possible to express preconditions on the system and thus on the input streams, but they are not specified by means of a declaration. They are additional constraints. Second, *memory streams* or *mems* are defined by an expression over the other streams and can specify an optional *initial expression* consistent with their sort. A memory stream which does specify an initial expression is *initialized*; otherwise, it is *uninitialized*. Memory streams represent flows depending solely on the previous state. At the stream system level however, there is no concept of current, previous or next state. Last, *combinatorial streams* or *combs* are defined by an expression over the other streams. They differ from *mems* in that they represent flows of data depending only on the other flows in the current state.

The expressions defining the combinatorial streams depend on other *combs*, *ins* and *mems* and form a Directed Acyclic Graph (DAG). The presence of a cycle at this point would indicate that it is not possible to establish the value of a combinatorial flow because of a circular dependency between *combs*. The expressions defining the memory streams can, and generally do introduce cycles to this DAG as they depend on the other streams as well. But while the dependencies between streams are circular, there is no circularity from a temporal point of view as memory streams encode flows of data based on the previous state.

The *raison d'être* of a stream system is to provide a convenient way to *unroll* the system it represents – *i.e.* construct MSFOL formulas encoding traces of finite length – as was the case for the transition system formalism introduced in Section 6.1 at a more theoretic level. The motivation behind stream systems is that a state $s_{\mathbf{i}} \triangleq \langle ins_{\mathbf{i}}, mems_{\mathbf{i}}, combs_{\mathbf{i}} \rangle$ is fully determined by the values of its inputs ($ins_{\mathbf{i}}$) and of its memories in the previous state ($mems_{\mathbf{i-1}}$). Furthermore a trace of states $s_{\mathbf{0}}$, $s_{\mathbf{1}}$, $s_{\mathbf{2}}$, ... is determined by the values of its memories at $s_{\mathbf{0}}$ ($mems_{\mathbf{0}}$) and of its inputs for each step. Figure 10.4 depicts the principle, where $\phi(s_{\mathbf{i-1}})$ (*resp.* $\psi(ins_{\mathbf{i}}, mems_{\mathbf{i}})$) yields the values of the memories (*resp.* combinatorial flows) for state $\mathbf{i}$. Last, stream systems all have a mandatory Boolean memory stream $Init$ defined as the expression $\bot$ with initial expression $\top$ to encode the flow of data $\top, \bot, \bot, \ldots$ to identify the first instant of an initialized trace.

In order to give a precise presentation of our unrolling algorithm, let us adapt Definition 6.1.2 to stream systems.

Definition 10.3.1 Given a logic $\mathcal{L} = \langle \Sigma, \mathcal{T}, \mathcal{F} \rangle$ and a stream system S with memory streams $m^1 : \sigma^1, \ldots, m^n : \sigma^n$ and inputs $i^1 : \sigma^{n+1}, \ldots, i^m : \sigma^{n+m}$, $\mathcal{L}^S$ is the logic
$\mathcal{L}(\{m^1_{\mathbf{0}} : \sigma^1, \ldots, m^n_{\mathbf{0}} : \sigma^n\} \cup \{i^1_{\mathbf{0}} : \sigma^{n+1}, \ldots, i^n_{\mathbf{0}} : \sigma^{n+m}\} \cup \{i^1_{\mathbf{1}} : \sigma^{n+1}, \ldots, i^n_{\mathbf{1}} : \sigma^{n+m}\} \cup \ldots)$.

Given a stream system S and a logic $\mathcal{L}$ such that all sorts used in S exist in $\mathcal{L}$, assume an

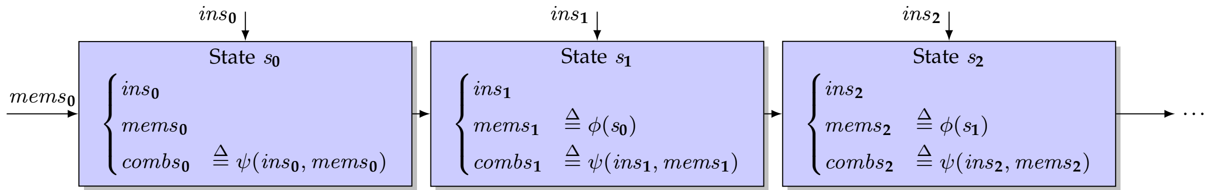


Figure 10.4: A trace of a Stream System.

expression `Expr` is either a function symbol (`Sym`) from $\mathcal{L}$ taking expressions as arguments, an input stream (`In`), a memory stream (`Mem`) or a combinatorial stream (`Comb`). Then, the following algorithm produces well-sorted $\mathcal{L}^S$ terms.

```
1  def unroll(expr: Expr, i: Int): Expr = expr match {
2    case Sym(args)  => Sym(args map (arg => unroll(arg,i)))
3    case input: In  => input_i
4    case comb: Comb => unroll(getExpression(comb),i)
5    case mem: Mem   =>
6      if (i == 0)
7        if (isInitialized(mem) && (mem != Init))
8          (if (Init_0) then unroll(getInitialExpr(mem),0) else mem_0)
9        else mem_0
10     else unroll(getExpression(mem),i)
11 }
```

Notice that the initial expression of *Init*, *i.e.* $\top$ is never unrolled. This is because state 0 is not necessarily an initial state, for example in the step instance of a k-induction proof attempt. On the other hand, any state $i > 0$ is not an initial state. Thus `unroll(`*Init*`,i)` with $i > 0$ is `unroll(getExpression(`*Init*`),i)` – line 10 – which is $\bot$. To force state 0 to be initial, one needs to specify that *Init* shall be true as a constraint. For instance, to create the formula which models are the initial states such that some predicate $P(s)$ on the streams is true, one would unroll $\mathit{Init} \wedge P(s)$ at 0, thus forcing Init_0 to be true.

Stuff differentiates between expressions on streams and $\mathcal{L}^S$-formulas in that it has a different expression structure for each. The MethLab for instance only handles expressions on streams: the stream system itself, proof objectives, constraints, (potential) invariants *etc.* The techniques on the other hand unroll expressions on streams to construct formulas, most of the times to query an SMT solver. This means that the Scala type system is aware of the difference between expressions on streams (`Expr`) and $\mathcal{L}^S$-formulas (thereafter called `Term`): any confusion is thus detected at compile time.

Unfortunately this unrolling strategy generates rather large formulas even on relatively small systems. This is a consequence of replacing every stream by its definition recursively. In Stuff unrolling creates formulas that are more compact by factorizing the unrolled definitions of the streams in let bindings. Creation of formulas consists of two steps. First, an unrolling phase annotates **all** streams identifiers without expending their definition:

```
def unroll(expr: Expr, i: Int): Term = expr match {
  case Sym(args)   => Sym(args map (arg => unroll(arg,i)))
  case input: In   => input_i
  case comb: Comb => comb_i
  case mem: Mem    =>
    if (i == 0 && isInitialized(mem) && (mem != Init))
        (Init_0 ∧ unroll(getInitialExpr(mem),0)) ∧ (¬Init_0 ∧ mem_0)
    else mem_i
}
```

So now `unroll` generates well-sorted $\mathcal{L}^S$-terms in the environment

$$\{\mathtt{comb}_\mathbf{i} : \sigma(\mathtt{comb}) \mid \mathtt{comb} \in \mathit{combs},\ i \geq 0\} \cup \{\mathtt{mem}_\mathbf{i} : \sigma(\mathtt{mem}) \mid \mathtt{mem} \in \mathit{mems},\ i \geq 1\}$$

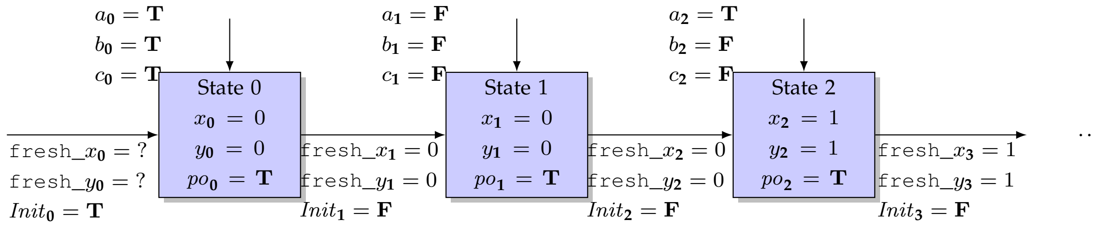
where $\sigma(\mathit{stream})$ is the sort of stream stream. The second step consists in embedding such terms in let-binding encoding the stream definitions between the streams and yielding well-sorted $\mathcal{L}^S$-terms (in an empty environment). Note that doing so requires topological sorting of the combinatorial streams which we do not discuss here. We continue the discussion on this technique on an example: a slightly altered version of the double counter system from Example 6.2.1.

Example 10.3.1 (Double Counter with observer) *We augment the double counter from Example 6.2.1 with an observer po. This new system is defined as*

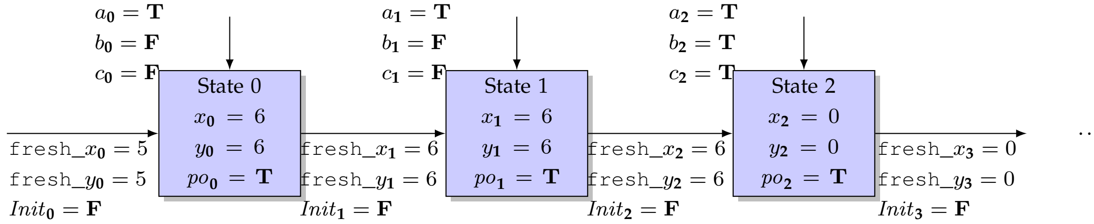
$$S_{DCO}(\mathbf{n_x}, \mathbf{n_y}) \triangleq \langle\langle a, b, c, x, y, po\rangle,\ \mathtt{Bool} \times \mathtt{Bool} \times \mathtt{Bool} \times \mathtt{Int} \times \mathtt{Int} \times \mathtt{Bool},\ I,\ T\rangle$$

where a, b and c are inputs, and

$$\begin{aligned}
I(s) &\triangleq (x = 0) \wedge (y = 0) \\
T(s, s') &\triangleq \left(x' = \begin{cases} \quad \textit{if}\,(b' \vee c') & 0 \\ \textit{else if}\,(a' \ \wedge\ x < \mathbf{n_x}) & x+1 \\ \textit{else} & x \end{cases}\right) \wedge \\
&\quad\ \left(y' = \begin{cases} \quad \textit{if}\,(c') & 0 \\ \textit{else if}\,(a' \ \wedge\ y < \mathbf{n_y}) & y+1 \\ \textit{else} & y \end{cases}\right) \wedge \\
&\quad\ po' = (0 \leq x \ \wedge\ x \leq \mathbf{n_x}).
\end{aligned}$$



(a) An initialized trace of the double counter system.



(b) A uninitialized trace of the double counter system.

Figure 10.5: Traces of the double counter system.

Inputs a, b and c become input streams in the stream system representation. Now, notice that when calculating a new state, it is enough to remember the values of x and y in the previous one. So we declare two memory streams $\langle \texttt{pre_}x, \texttt{int}\rangle$ and $\langle \texttt{pre_}y, \texttt{int}\rangle$ defined as expressions x and y respectively. Then, x and y are really combinatorial streams. The problem now is to encode the initial state predicate in their expression. Thanks to *Init* we can define combinatorial stream x as the expression

$$\begin{cases} \quad\text{if } (\texttt{Init}) & 0 \\ \text{else if } (b \vee c) & 0 \\ \text{else if } (a \ \wedge \ \texttt{pre_}x < \mathbf{n_x}) & \texttt{pre_}x + 1 \\ \text{else} & \texttt{pre_}x \end{cases}$$

and similarly for y. We see that the initial value of $\texttt{pre_}x$ and $\texttt{pre_}y$ do not matter as they do not impact x and y in any way. Thus, $\texttt{pre_}x$ and $\texttt{pre_}y$ are uninitialized memory streams. Last, combinatorial stream po is defined simply as $0 \leq x \ \wedge \ x \leq \mathbf{n_x}$. Figure 10.5 depicts an initialized (Figure 10.5a) and an uninitialized (Figure 10.5b) trace of the double counter. We now illustrate our two step unrolling strategy by constructing the base and step instances of an induction proof attempt.

Base instance. First, we construct the expression corresponding to the base instance on the streams of the system: $Init \wedge \neg(0 \leq x \leq \mathbf{n_x})$. We then unroll (the streams of) this expression at 0:

$$\texttt{unroll}(Init \wedge \neg(0 \leq x \leq \mathbf{n_x}),\ 0) \ \triangleq\ Init_{\mathbf{0}} \wedge \neg(0 \leq x_{\mathbf{0}} \leq \mathbf{n_x})$$

Also, the unroller memorizes which streams were unrolled, in this case x and $Init$, for the upcoming binding phase: $\texttt{bind}(Init_{\mathbf{0}} \wedge \neg(0 \leq x_{\mathbf{0}} \leq \mathbf{n_x})) \triangleq$

$$\text{let} \left(x_{\mathbf{0}} = \begin{cases} \quad\text{if } (Init_{\mathbf{0}}) & 0 \\ \text{else if } (b_{\mathbf{0}} \vee c_{\mathbf{0}}) & 0 \\ \text{else if } (a_{\mathbf{0}} \ \wedge\ \texttt{pre_}x_{\mathbf{0}} < \mathbf{n_x}) & \texttt{pre_}x_{\mathbf{0}} + 1 \\ \text{else} & \texttt{pre_}x_{\mathbf{0}} \end{cases} \right) (Init_{\mathbf{0}} \wedge \neg(0 \leq x_{\mathbf{0}} \leq \mathbf{n_x}))$$

which is indeed the correct base instance. There is no let binding for $y_{\mathbf{0}}$ since the unroller did not unroll it: it knows it does not appear in the expression. Memory $Init$ is not bound as it is a free variable of the trace (of one state). The SMT solver will quickly realize that to evaluate this formula to true, $Init_{\mathbf{0}}$ must be given the value true. This forces $x_{\mathbf{0}}$ to evaluate to 0, and $\neg(0 \leq x_{\mathbf{0}} \leq \mathbf{n_x})$ to false. The formula is unsat.

Step instance. For the step instance we need to express constraints on two consecutive states. The first state must verify the proof objective:

$$\texttt{unroll}(0 \leq x \leq \mathbf{n_x},\ 0) \ \triangleq\ 0 \leq x_{\mathbf{0}} \leq \mathbf{n_x}.$$

The second state should falsify the proof objective:

$$\texttt{unroll}(\neg(0 \leq x \leq \mathbf{n_x}),\ 1) \ \triangleq\ \neg(0 \leq x_{\mathbf{1}} \leq \mathbf{n_x}).$$

At this point, the unroller knows it has unrolled x at 0 and at 1. The first state is 0: as all following states depend on it, a first binding corresponding to the definition of $x_{\mathbf{0}}$ is constructed:

$$\text{let} \left(x_{\mathbf{0}} = \begin{cases} \quad\text{if } (Init_{\mathbf{0}}) & 0 \\ \text{else if } (b_{\mathbf{0}} \vee c_{\mathbf{0}}) & 0 \\ \text{else if } (a_{\mathbf{0}} \ \wedge\ \texttt{pre_}x_{\mathbf{0}} < \mathbf{n_x}) & \texttt{pre_}x_{\mathbf{0}} + 1 \\ \text{else} & \texttt{pre_}x_{\mathbf{0}} \end{cases} \right) (\ldots$$

The unroller knows it unrolled x at 1, so it also need to produce a let binding corresponding to its definition. But to do that, it first needs to encode the definitions of the memory streams

$\texttt{pre_}x$, $\texttt{pre_}y$ and Init, at state 1. Thanks to a simple analysis on the definition of x, the unroller realizes it does not need the binding for $y_\mathbf{1}$ as $y_\mathbf{1}$ is not used anywhere.

$$\text{let } (x_\mathbf{0} = \dots)(\text{let } (\texttt{pre_}x_\mathbf{1} = x_\mathbf{0},\ \mathit{Init}_\mathbf{1} = \texttt{false})(\dots$$

Everything is set up now for the definition of $x_\mathbf{1}$ which is the same as for $x_\mathbf{0}$ except that all indices "0" become "1". In the end, $\texttt{bind}((0 \leq x_\mathbf{0} \leq \mathbf{n_x}) \wedge \neg(0 \leq x_\mathbf{1} \leq \mathbf{n_x})) \triangleq$

$$\text{let } \left(x_\mathbf{0} = \begin{cases} \text{if } (\mathit{Init}_\mathbf{0}) & 0 \\ \text{else if } (b_\mathbf{0} \vee c_\mathbf{0}) & 0 \\ \text{else if } (a_\mathbf{0} \ \wedge\ \texttt{pre_}x_\mathbf{0} < \mathbf{n_x}) & \texttt{pre_}x_\mathbf{0} + 1 \\ \text{else} & \texttt{pre_}x_\mathbf{0} \end{cases} \right) ($$

$$\text{let } (\texttt{pre_}x_\mathbf{1} = x_\mathbf{0},\ \mathit{Init}_\mathbf{1} = \texttt{false})($$

$$\text{let } \left(x_\mathbf{1} = \begin{cases} \text{if } (\mathit{Init}_\mathbf{1}) & 0 \\ \text{else if } (b_\mathbf{1} \vee c_\mathbf{1}) & 0 \\ \text{else if } (a_\mathbf{1} \ \wedge\ \texttt{pre_}x_\mathbf{1} < \mathbf{n_x}) & \texttt{pre_}x_\mathbf{1} + 1 \\ \text{else} & \texttt{pre_}x_\mathbf{1} \end{cases} \right) ($$

$$(0 \leq x_\mathbf{0} \leq \mathbf{n_x}) \wedge \neg(0 \leq x_\mathbf{1} \leq \mathbf{n_x})$$

$$)))$$

which is indeed the step instance. Note that as expected it is a well-sorted $\mathcal{L}^S$-formula, and the only symbols to interpret are $\mathit{Init}_\mathbf{0}$, $\texttt{pre_}x_\mathbf{0}$, and the inputs at 0 and 1.

We find this approach to unrolling easy to use once understood. It is also efficient as the size of the formulas is kept as small as possible thanks to let-bindings – as opposed to systematic inlining of the streams definition. Also, the unrolling algorithm follows the cone of influence of the constraints it unrolls: only necessary streams are unrolled. Additionally, calls to the `unroll` function are memoized in our implementation. We do not give the details of this feature as it is rather straightforward.

CHAPTER 11

Assumptio: an SMT-lib 2 Compliant Solver Wrapper

Stuff uses SMT solvers extensively: between 10 and 20 of them can be used at the same time during an analysis for k-induction(s), HullQe, BrutalIQe, and PDR. Assumptio is a GPL v3, actor-based API written in Scala designed to allow its user to perform on the fly queries to an SMT solver, whose internal state is maintained until closed by the user. The solver is launched in a separate system process, and communicates with Assumptio in interactive, text mode *via* system pipes. As of today, the officially supported solvers are Microsoft Research's Z3 [29] (version 4.3.1 and later), University of Trento's MathSAT 5 [20] and CVC4 [3]. The idea behind Assumptio is that developers using SMT solvers should not have to worry about what the actual underlying solver is, and should be able to change it easily without modifying the existing code. This idea follows the spirit of the SMT-LIB initiative which introduced a unique language to express SMT benchmarks, in order to ease comparison of SMT solvers. Assumptio naturally relies heavily on the SMT-LIB 2 language to interact with the underlying solver.

An interesting feature of Assumptio is that it does not have any internal representation for the data structure encoding the formulas. It is parameterized by the type of the user's structure. There is hence no need for the users to translate their structure, maintain two of them in parallel or to adopt an imposed structure. Since Assumptio needs to communicate formulas to the solver, it also requires an SMT-LIB 2 printer for the user's expressions. Last, a parser building a subset of the expressions must be provided. This subset can be very small – even empty – depending on what SMT-LIB commands the user will use.

We briefly present Assumptio as a library, independently from Stuff. Assumptio is a Scala *trait*, an object-oriented construct similar to interfaces. We discuss its main features in Section 11.1 before looking at its architecture in Section 11.2. Last, we briefly conclude and report on its integration in Stuff in Section 11.3.

11.1 Overview

As mentioned above Assumptio does not have an internal representation for expressions and is parameterized by the user's structure. Let us take a quick look at how this works. Here are the first lines of code of the Assumptio trait:

```
/** @tparam Expr The type of the user's expression structure. */
trait AssumptioTrait[Expr] extends RegexParsers {

  /** SMT lib 2 parser for the user's structure. */
  def exprParser: Parser[Expr]

  /** SMT lib 2 printer for the user's structure. */
  def toSmtLibExpr(expression: Expr, bw: Writer): Unit
```

So, the first thing the users need to do is extend this trait and specify their structure for expressions. They must also provide an SMT-LIB 2 printer for their expressions and a parser for the SMT-LIB expression language. Needless to say, only the subset of the user's expression structure that will actually be communicated to the solver needs to be printable. Regarding the parser, it is not always necessary for it to cover the whole SMT-LIB expression language. If the only queries used are check-sat's, then no parser is required at all. Indeed, answers to a check-sat query never result in expression parsing. The same applies for get-unsat-core which produces a list of labels. When using get-model however, the user should provide a parser for whatever this query can produce: at least constants of the logic at hand and identifiers. Similarly for get-value queries, where not only the values but also the expressions for which the values were asked should be parsable.

Once the trait has been instantiated on an expression structure, messages are simply sent using the traditional Akka actor syntax to Assumptio actors. Messages correspond directly to SMT-LIB queries except for a few additions such as `Script`, to send a list of commands in one message. Some messages do not produce a result[1]: `SetLogic`, `DeclareFun`, `Assert`, ... On the other hand, messages such as `CheckSat` and `GetModel` are queries explicitly calling for an answer (`Sat`, `Unsat`, `Model(...)`, ...). We call them *query messages*. Now, if the user queries more than one Assumptio instance in the same actor, it can be tedious when receiving an answer to determine which query it corresponds to. So when sending a query to an Assumptio instance, the user must provide an identifier which will appear in the answer. For instance, `CheckSat(42)` produces the answer `Sat(42)`, `Unsat(42)` or `Unknown(42)`.

```
val assumptio = context.actorOf(Props(Assumptio(...)), name="assumptio")
```

[1] SMT-LIB commands which do not produce a result explicitly can cause the underlying solver to produce an output, such as an error or the string "success". This will be discussed below.

```
assumptio ! SetLogic(QF_LRA)
assumptio ! DeclareFuns(...)
assumptio ! Assert(...)
assumptio ! CheckSat(42)
// More quickly:
// assumptio ! Script(
//   SetLogic(QF_LRA) :: DeclareFuns(...) :: Assert(...) ::
//   CheckSat(42) :: Nil
// )
context become {
  case Sat(42) => { println("Sat.") ; context stop assumptio }
  case Unsat(42) => { println("Unsat.") ; context stop assumptio }
  case Unknown(42) => { println("Cannot decide.") ; context stop assumptio }
  case m => { println("Unexpected message: " + m + ".") ; context stop assumptio
      }
}
```

Although Assumptio focuses on parallel applications, a sequential version is also available. For the sake of uniformity, the syntax of the interaction with the sequential version mimics message sending and reception. A simple example follows:

```
val assumptio = AssumptioSeq(...)
assumptio ! SetLogic(QF_LRA)
assumptio ! DeclareFuns(...)
assumptio ! Assert(...)
assumptio ! CheckSat(42) match {
  case Sat(42) => println("Sat.")
  case Unsat(42) => println("Unsat.")
  case Unknown(42) => println("Cannot decide.")
}
// Close the instance.
assumptio ! KillSolver
```

Despite the syntactic similarity, no actor is created for the Assumptio instance and all solver dependent code will be executed within the same thread (or actor).

Instantation. When instantiating Assumptio, four parameters must be provided:

- `caller`: the `ActorRef` the results of the queries will be sent to;
- `sConf`: the solver configuration: which solver should be used, whether models or unsat cores should be enabled...;
- `name`: the solver name, mainly for log and debug purposes;
- `freeRestarts`: if true, forces Assumptio to create two solver processes using the same configuration; when the active solver process is reset, the up-to-now-passive

solver process becomes active, resulting in seemingly free restarts of the solver at user level.

The `sConf` mandatory parameter lets the user decide which solver should be used, and which options should be activated. It is possible to omit this information and instead use an identifier, such as `"qeSolverConf"`. A solver configuration should be associated to this identifier in a separate configuration file: Assumptio loads the configuration file at runtime and applies the corresponding configuration. This allows the user to change the underlying solver after compilation depending on what solvers are available to the client of the application. In addition to these four mandatory parameters, the Assumptio constructor has four optional ones:

- `logic`: a logic to initialize the solver with (default `None`);
- `functions`: some functions to declare during the solver initialization (default `None`);
- `constraints`: some expressions to assert during the solver initialization (default `None`);
- `log`: indicates whether the queries and their results should be logged, and where.

Parameters `logic`, `functions` and `constraints` can make the solver start with a given logic and predefined function symbols and a set of constraints. This avoids sending messages right after instantiation or a solver restart to set the same information. Last, the `log` parameter lets the user log all queries sent to the solver as executable SMT-LIB files[2].

11.2 Architecture and Backend Solvers

This section details the architecture surrounding the backend solver process. More precisely we show how we solve the following problems. First, avoiding circular read / write lockups when communicating with the solver process. Second, easing the process of adding support for a new backend solver despite the differences in the format of the output the solvers produce.

The internal architecture of an Assumptio instance is presented in Figure 11.1. The user creates an `Assumptio` instance, which triggers the creation of an actor called `SolverReader`, and of an external process called `Solver`[3]. `Assumptio` can write on the `Solver` process standard input while the reader actor, `SolverReader`, can read its standard output. These

[2] Actually, restarts have no legal SMT-LIB counterpart and are logged as comments. Running an Assumptio log containing restart would hence probably crash because of the post-restart commands.

[3] We assume the *freeRestarts* mode is not activated for the sake of clarity: only one solver process is created.

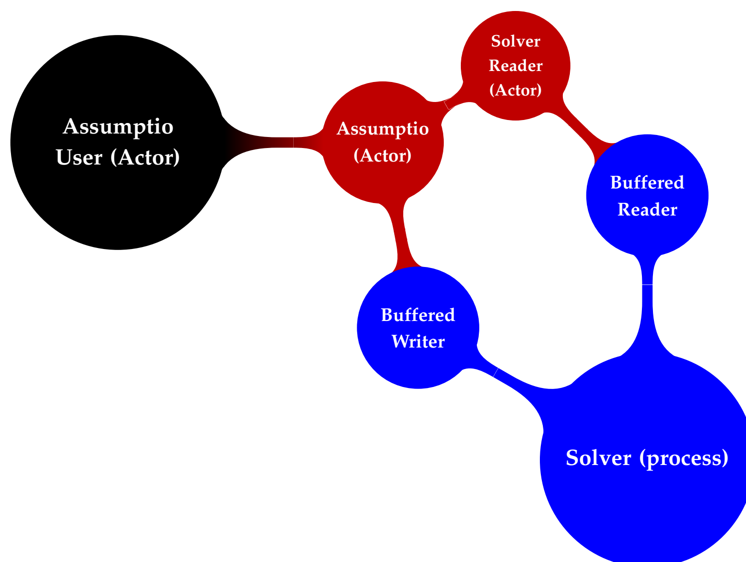


Figure 11.1: Hi-level view of user interaction and internal architecture.

two distinct actors are needed to communicate with the external process in order to avoid read / write process lockups. This is an actor oriented adaptation of the classical *stream gobbling* pattern, traditionally implemented with separate threads for input and output streams. So, the user only interacts with the `Assumptio` instance: upon reception of a message, the `Assumptio` instance writes whatever needs to be written on the `Solver` input, and resumes message reception. If the message was in fact a query, it also forwards the message to the `SolverReader`. This makes the reader aware that it should parse a result, and of the manner it should parse it depending on what the query was. Now, when the `SolverReader` receives a message forwarded by `Assumptio` it starts watching the `Solver` output. Once the `Solver` outputs the result of the query, the `SolverReader` parses it and sends the result back to the `Assumptio` actor, which then forwards it to the user. Note that it is necessary that the message containing the result is sent by the `Assumptio` actor for the sake of `ActorRef` consistency. It would not be coherent for the user to send a message to the `Assumptio` instance but receive the result from the `SolverReader`.

Backend solvers are handled through a class inheriting from a trait called `SolverConf`. Each SMT solver supported by Assumptio has its own and the user is asked to supply one when instantiating Assumptio – the `sConf` parameter. The `SolverConf` trait contains all the parsers used in Assumptio. As a consequence, adding support for a new SMT-LIB 2 compliant solver boils down to creating a sub-class of `SolverConf` to specify how the solver should be created, overriding parsers if necessary. Z3 and CVC4 for instance respect the default parsers defined in `SolverConf`. Their respective classes do not override any parser

and simply indicate how to launch and initialize the solver process. For instance, the code for the Z3 solver configuration class is:

```
case class Z3(parseSuccess: Boolean = true,
              models: Boolean = true,
              unsatCores: Boolean = false,
              postCmd: String  = "") extends SolverConf {
  val command = "z3 -smt2 -in" +
                (if (models) " -m" else "") +
                " " + postCmd

  def init(bw: BufferedWriter, reader: SolverReader) = {
    if (parseSuccess) {
      bw.write("(set-option :print-success true)\n")
      reader ! ParseSuccess("Checking for (set-option :print-success true)'s
          success.")
    }
    else bw.write("(set-option :print-success false)\n")
    if (unsatCores) {
      bw.write("(set-option :enable-cores true)\n")
      if (parseSuccess) reader ! ParseSuccess("Checking for (set-option :" +
                                              "enable-cores true)'s success.")
    }
  }

  override def toString = "Z3"
}
```

MathSAT on the other hand differs slightly in the way it prints models out. We account for this difference directly in the MathSAT class as follows:

```
case class MathSat(parseSuccess: Boolean = true,
                   models: Boolean = true,
                   unsatCores: Boolean = false,
                   postCmd: String   = "") extends SolverConf {
  val command = "mathsat" +
                (if (models) " -model_generation=true" else "") +
                " " + postCmd

  def init(bw: BufferedWriter, reader: SolverReader) =
    if (parseSuccess) {
      bw.write("(set-option :print-success true)\n")
      reader ! ParseSuccess("Checking for (set-option :print-success true)'s
          success.")
    }

  override def model: Parser[Map[Expr, Expr]] =
    "(" ~> rep1(define) <~ ")" ^^ {case x => Map() ++ (x.toTraversable)}
```

```
  override def define: Parser[(Expr,Expr)] =
    "(" ~> exprParser ~ exprParser <~ ")" ^^ {case id~value => (id,value)}

  override def toString = "MathSat 5"
}
```

We conclude this section with a discussion on the `print-success` SMT-LIB option. It needs to be activated before the logic of the solver is set and forces the solver to print `"success"` whenever a command that does not produce a result did not cause any error. When creating a solver configuration class, the `parseSuccess` argument indicates whether the `print-success` option should be set to `true`. But activating it does not cause the client actor to receive a `Success` message whenever `"success"` is parsed on the backend solver output, so how does it benefit the user?

The purpose of this option is to be able to detect an error as soon as it appears. Not activating `parseSuccess` in Assumptio implies no parsing is performed until a query message expected is received. For example, were one to send an incorrect `Assert` message – by using undeclared symbols for instance – followed by a `CheckSat`, the error output by the solver would only be discovered when parsing for the `CheckSat` result. Assumptio would thus send an error message containing the error from the solver and the message it was handling when it received it, *i.e.* the `CheckSat` message.

When `parseSuccess` is activated on the other hand, even non-query messages are forwarded to the `SolverReader` so that it parses for `"success"`. Errors are detected as soon as they appear and the user is notified with the real message that caused it. This feature is useful for debugging purposes but should be deactivated when aiming for performance as it introduces an overhead on all non-query messages due to the additional parsing and message passing.

11.3 Source Code, User Manual and Integration in Stuff

Assumptio as well as its 40 page user manual is available at

https://cavale.enseeiht.fr/redmine/projects/assumptio.

It is provided with an example of linear system solving by SMT. The example comes in several versions to illustrate Assumptio with and without the `AssumptioUser` trait, free restarts and sequential Assumptio. The Assumptio package also includes a stress test which instantiates a given number of Assumptio wrappers using a given set of backend solvers and runs on each of them simultaneously a test scenario using all the SMT-LIB queries and most of its commands. With a JVM heap of 1024M, the stress test manages to run around 50 solvers at the same time on a standard laptop.

In Stuff in particular we have found that Assumptio scales up nicely; it is not unusual during an analysis to run 20 wrapped backend solvers. In our experience the overhead of text-based communication with the solver processes is worthwhile considering the gains in development time and the flexibility brought by Assumptio. Also, the Assumptio layer ensures we do not depend on a precise version of any SMT solver: changes in the printing and parsing conventions of the commands are absorbed by the wrapper and no modification in the code of the tool using Assumptio is necessary.

Part IV

Conclusion

CHAPTER 12

Summary and Perspectives

This chapter concludes this thesis by summarizing our contribution in Section 12.1 and discussing perspectives in Section 12.2.

12.1 Conclusion

Software verification is a very active research topic, but despite the huge progress made during the last decades, some systems still elude the capabilities of current techniques (*cf.* Chapter 6). The purpose of the work presented in this thesis is to improve the state of the art in automatic verification of software components in avionics systems, as presented in Chapter 4. We studied in particular (relational) invariant discovery methods in a collaborative framework based on k-induction. We hope that our work will help the spread of automatic formal verification in our industrial community.

First, we formalized transition systems as predicates in many-sorted first-order logic (MSFOL) in Chapter 5 and Chapter 6. We defined a category of MSFOL signatures in which traces of transition systems, proof objectives, validity, *etc.* are expressed with a precise mathematical meaning. While this representation is widely used in the literature, the mathematical details and in particular the precise semantics of unrolling are often omitted.

We presented our main contributions in Part II. We began by improving on the SMT-based quantifier elimination (QE) algorithm on linear real arithmetic due to Monniaux (Chapter 7). This lead to a speed up of several orders of magnitude, confirmed by our experiments on a large number of benchmarks. Also, we adapted the algorithm so that it is able to handle formulas with Boolean and integer octagon literals in addition to linear real literals. We then proposed invariant discovery methods taking advantage of the improved QE algorithm. First, a template-based invariant discovery method called BrutalIQe in Section 8.2, which follows the ideas of [49]. In practice it is used to find bounds on the state variables of the system. Then, we introduced in Chapter 9 a new invariant discovery heuristic called HullQe. HullQe computes pre-images of unsafe states and studies their partitioning in order to generate relational potential invariants. Last, we proposed a parallel proof engine archi-

tecture based on k-induction in Section 8.1, allowing the invariant discovery techniques to cooperate by a continuous integration of the invariants discovered.

Part III discussed the most interesting aspects of our implementation of this collaborative parallel architecture, called Stuff. In Chapter 10 we presented the actor-oriented and highly extensible nature of Stuff. We also showed in detail (Section 10.3) that our representation of transition systems respects the semantics introduced in Part I, in particular when constructing MSFOL formulas by unrolling the transition relation. Last, we discussed the most important features of our SMT-LIB 2 compliant solver wrapper Assumptio in Chapter 11.

We evaluated the benefit of the collaborative architecture and methods proposed in this thesis on a reconfiguration system and a voting logic system provided by Rockwell Collins. Stuff concludes in a matter of seconds thanks to the cooperation between the techniques of the framework.

- BrutalIQe finds bounds on the state variables reducing the search space;
- HullQe discovers relational property-directed invariants;
- k-induction proves that the invariants discovered strengthen the proof objective.

These systems correspond to design patterns ubiquitous in the development of critical avionics system, and to the best of our knowledge no other formal technique is able to verify them fully automatically. Our methodology is of high interest in a certification context because *(i)* it is fast and automatic and hence does not impact the development process or introduce additional cost, and *(ii)* the strengthened proof objective is (1-)inductive, therefore the proof can be easily re-checked by third-party (qualified) tools to make a stronger argument for certification authorities.

We also reported on progress towards the analysis of a full functional chain at the model level. By combining our framework with the abstract interpreter introduced in [65], we were able to verify automatically part of a functional chain making use of sensor monitors and voting logic systems before the command law of the chain. This achievement on a simplified, yet representative, running system is a step towards the verification of realistic functional chains.

12.2 Future Directions

We now discusse future directions for research in the continuity of this study. We discuss quantifier elimination in Section 12.2.1, HullQe in Section 12.2.2, PDR in Section 12.2.3 and our formal framework Stuff in Section 12.2.4.

12.2.1 Quantifier Elimination

Quantifier Elimination is the backbone of all the invariant discovery techniques introduced in this thesis. A first direction for future work is therefore to extend our QE technique to more general fragments such as linear integer arithmetic and non-linear real arithmetic. Doing so would benefit the framework as a whole and allow us to evaluate it on systems expressed in richer fragments. Now, SMT solvers are already able to handle those fragments, and so does the structural extrapolation method we proposed. The limitations lie in the projection phase.

Cylindrical Algebraic Decomposition (CAD [21]) and Virtual Substitution (VS [79, 80]) are projection methods over non-linear real arithmetic. Unfortunately, they are far from scaling up to QE challenges stemming from the verification of industrial systems such as ours. The increase in expressive power should not be at the cost of performance. All our techniques, except k-induction, rely on the fact that our QE implementation is very fast. Changing the projection phase for CAD or VS in our QE algorithm would result in huge drop in performance making the whole framework unusable. This calls for more research in the field of quantifier elimination on the non-linear real fragment, for instance to identify smaller non-linear fragments for which an efficient QE algorithm exists, or combination of QE approaches such as [72].

12.2.2 HullQe

HullQe can be decomposed in two parts: the incremental pre-image computation and the heuristics working on the pre-images to generate potential invariants. Interesting relational invariants are discovered by merging parts –polyhedra– of the pre-images according to certain criteria to avoid a blowup in the number of potential invariant generated. More precisely, the merge should be exact or the polyhedra should have at least one point in common. Yet, it can be the case that it is not possible to merge any polyhedra with these criteria, *i.e.* if the pre-images are made of completely disjoint polyhedra. It would be interesting to research more permissive heuristic criteria based on realistic use cases.

Also, introducing abstraction in the pre-image computation would speed up the computation of gray states and potentially discover extra invariants. Interesting work proposed recently in [22] performs a similar backwards pre-image computation with a mechanism for revisable abstraction: a counterexample trace from the initial states is analyzed in order to determine which, if any, of the approximations are responsible for the trace, and revert them.

12.2.3 Property-Directed Reachability

PDR is a very promising technique developed initially for purely propositional systems. Efficient generalization to SMT is non-trivial however: in [19], the authors propose an adaptation of PDR to programs represented as Control Flow Graphs (CFG). They propose a "tree-based" version of PDR which lessens the combinatorial explosion of the reachable state space by blocking cubes for abstract paths instead of blocking them for the entire system. It would be very interesting to experiment with this idea in the context of Lustre programs.

12.2.4 Stuff

An interesting topic has been omitted in this thesis: finding counterexamples to proof objectives when they are not invariants for the system. Indeed, while HullQe discovers strengthening invariants in a very short time on our systems, finding counterexamples is a different story. While the pre-image iteration could eventually reach the initial states, this is very unlikely to succeed on realistic systems. PDR on the other hand takes the initial states into account from the very start in its construction of the frame sequence. Combined with the backward traversal of the state space directing the refinement of the frames, PDR reportedly has the ability to find counterexamples faster than BMC and most other methods due to its improved locality, even more so in its aforementioned tree-based version.

Finding counterexamples is of tremendous interest for industrial use to avoid the expensive and time-consuming activity of bug detection and correction. Unfortunately counterexamples traces returned by formal tools are generally made of concrete states, the values of which are found by an SMT solver. As a result, it is often difficult for developers to identify the bug in the model allowing this trace to appear, especially if the trace is composed of hundreds or thousands of concrete states. PDR differs from most verification methods in that it does not return a single trace but rather a class of counterexample traces as a sequence of cubes. It would be interesting to research an SMT version of PDR specially designed for the discovery of such abstract counterexample classes on realistic systems in our application field, and to evaluate their value for system designers in a debugging process, in an industrial context.

Last, we are very excited by the fruitful collaboration of Stuff with the abstract interpreter from [65] presented in Section 9.4. We would like to conduct further research on the composition of tools and techniques for the modular analysis of realistic functional chains. This calls for more use cases, ideally industrial ones. An important class of open loop properties has been successfully studied in this work, but additional research must be conducted to enhance this results: improving the automation of the analysis –automatic recognition of which part of the system needs to be analyzed with each technique. Last, it is mandatory

to improve the scalability of the methods used to real world systems which can have more than ten times as many state variables.

CHAPTER 13

References

Bibliography

[1] Hirokazu Anai and Volker Weispfenning. Reach set computations using real quantifier elimination. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *Lecture Notes in Computer Science*, pages 63–76. Springer, 2001.

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT–LIB standard: Version 2.0, 2010.

[5] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, September 1991.

[6] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 178–183. Springer Berlin Heidelberg, 2011.

[7] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[8] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electr. Notes Theor. Comput. Sci.*, 66(2):160–177, 2002.

[9] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[10] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[11] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Paths to property violation: A structural approach for analyzing counter-examples. In *HASE*, pages 74–83. IEEE Computer Society, 2010.

[12] Aaron R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[13] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188. ACM Press, 1987.

[14] Adrien Champion. Assumptio and stuff: using Z3 in a collaborative parallel formal verification framework (invited presentation). In *Z3 Special Interest Group (SIG)*. Microsoft Research, 2011.

[15] Adrien Champion and Rémi Delmas. Elimination de quantificateurs: Extrapolation efficace par analyse structurelle et coeurs insatisfiables. In *Modélisation des Systèmes Réactifs (MSR)*, 2013, to be published.

[16] Adrien Champion, Rémi Delmas, and Michael Dierkes. Generating property-directed potential invariants by backward analysis. In Peter Csaba Ölveczky and Cyrille Artho, editors, *FTSCS*, volume 105 of *EPTCS*, pages 22–38, 2012.

[17] Adrien Champion, Rémi Delmas, Michael Dierkes, Pierre-Loïc Garoche, Romain Jobredeaux, and Pierre Roux. Formal methods for the analysis of critical control systems models: Combining non-linear and linear analyses. In Charles Pecheur and Michael Dierkes, editors, *FMICS*, volume 8187 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[18] John J. Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[19] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In P. Madhusudan and SanjitA. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer Berlin Heidelberg, 2012.

[20] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[21] George E. Collins. Hauptvortrag: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the 2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, London, UK, 1975. Springer-Verlag.

[22] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-based Model Checker for Parameterized Systems. In Madhusudan Parthasarathy and Sanjit A. Seshia, editors, *CAV 2012: Proceedings of the 24th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berkeley, California, USA, July 2012. Springer Verlag.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[24] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.

[25] Werner Damm, Stefan Disch, Hardi Hungar, Swen Jacobs, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *ATVA*, volume 4762 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2007.

[26] Werner Damm, Stefan Disch, Hardi Hungar, Jun Pang, Florian Pigorsch, Christoph Scholl, Uwe Waldmann, and Boris Wirtz. Automatic verification of hybrid systems with large discrete state space. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2006.

[27] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[28] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[29] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[30] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a). In Jr. and Somenzi [45], pages 14–26.

[31] David Deharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *13th International Conference on Formal Methods in Computer-Aided Design(FMCAD'13)*. IEEE Press, 2013.

[32] Michael Dierkes. Formal analysis of a triplex sensor voter in an industrial context. In G. Salaün and B. Schätz, editors, *Proceedings of the 16th edition of FMICS*, volume 6959 of *LNCS*, pages 102–116. Springer, 2011.

[33] Isil Dillig, Thomas Dillig, Kenneth L. McMillan, and Alex Aiken. Minimum satisfying assignments for SMT. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12, pages 394–409, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 125–134. FMCAD Inc., 2011.

[35] Marie-Claude Gaudel. Formal specification techniques. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 223–227, 1994.

[36] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[37] Michael Gonzalez Harbour. Real-time POSIX: An overview, 1993.

[38] David Harel and Amir Pnueli. Logics and models of concurrent systems. chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[39] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *IJCAI*, pages 235–245. William Kaufmann, 1973.

[40] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[41] Hoon Hong. Heuristic search strategies for cylindrical algebraic decomposion. In Jacques Calmet and John A. Campbell, editors, *AISMC*, volume 737 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1992.

[42] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artifical intelligence*, IJCAI'07, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[43] 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.

[44] Ajith K. John and Supratik Chakraborty. A quantifier elimination algorithm for linear modular equations and disequations. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 486–503. Springer, 2011.

[45] Warren A. Hunt Jr. and Fabio Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003.

[46] Temesghen Kahsai, Yeting Ge, and Cesare Tinelli. Instantiation-based invariant discovery. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 192–206, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. In J. Barnat and K. Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

[48] Temesghen Kahsai and Cesare Tinelli. PKIND: a parallel k-induction based model checker. In J. Barnat and K. Heljanko, editors, *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verifiCation (Snowbird, Utah, USA)*, Electronic Proceedings in Theoretical Computer Science, 2011.

[49] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Deduction and Applications*, 2005.

[50] Deepak Kapur. Program analysis using quantifier-elimination heuristics - (extended abstract). In Manindra Agrawal, S. Barry Cooper, and Angsheng Li, editors, *TAMC*, volume 7287 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2012.

[51] Bronislaw Knaster. Un théorème sur les fonctions d'ensembles. pages 133–134. Ann. Soc. Polon. Math., 1928.

[52] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.

[53] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (BNF). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK, 2003.

[54] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, May 1992.

[55] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Jr. and Somenzi [45], pages 1–13.

[56] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2008.

[57] David Monniaux. Automatic modular abstractions for template numerical constraints. *Logical Methods in Computer Science*, 6(3), 2010.

[58] David Monniaux. Quantifier elimination by lazy model enumeration. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2010.

[59] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.

[60] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2004.

[61] P. Pedersen, Marie-FranÃğoise Roy, and Aviva Szpirglas. Counting real zeros in the multivariate case. In Frédéric Eyssette and André Galligo, editors, *Computational Algebraic Geometry*, volume 109 of *Progress in Mathematics*, pages 203–224. Birkhäuser Boston, 1993.

[62] Anh-Dung Phan, Nikolaj Bjørner, and David Monniaux. Anatomy of Alternating Quantifier Satisfiability (Work in progress). In *SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories*, page 6, Manchester, Royaume-Uni, June 2012.

[63] Mojżesz Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du* 1*er* *congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[64] I. Reed. Boolean difference calculus and fault finding. *SIAM Journal on Applied Mathematics*, 24(1):134–143, 1973.

[65] Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Eric Feron. A generic ellipsoid abstract domain for linear time invariant systems. In Thao Dang and Ian M. Mitchell, editors, *HSCC*, pages 105–114. ACM, 2012.

[66] D. Rowell. Dicrete time observers and LQG control. MIT, Dpt. of Mechanical Engineering – 2.151 Advanced System Dynamics and Control – http://web.mit.edu/2.151/www/Handouts/Kalman.pdf, 2004.

[67] John Rushby. New challenges in certification for aircraft software. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 211–218, New York, NY, USA, 2011. ACM.

[68] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.

[69] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[70] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[71] Ian Sommerville. *Software Engineering (Chap. 27)*. Addison-Wesley, Harlow, England, 9 edition, 2010.

[72] Thomas Sturm and Ashish Tiwari. Verification and synthesis using real quantifier elimination. In Éric Schost and Ioannis Z. Emiris, editors, *ISSAC*, pages 329–336. ACM, 2011.

[73] Alfred Tarski. A decision method for elementary algebra and geometry: Prepared for publication with the assistance of J.C.C. McKinsey. Technical report, RAND Corporation, Santa Monica, CA., 1951.

[74] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific Journal of Mathematics*, 1955.

[75] Martin Thurau. Akka framework. University of Lübeck, available online at http://media.itm.uni-luebeck.de/teaching/ws2012/sem-sse/martin-thurau-akka.io.pdf [consulted March 29, 2014], 2012.

[76] Boris Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Proceedings of the USSR Academy of Science*, pages 569–572, 1950.

[77] G.S. Tseitin. On the complexity of derivation in propositional calculus. In JÃČÂűrgH. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[78] Willard Van Orman Quine. *On Cores and Prime Implicants of Truth Functions*. Mathematical Association of America, 1959.

[79] Volker Weispfenning. Quantifier elimination for real algebra – the cubic case. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '94, pages 258–263, New York, NY, USA, 1994. ACM.

[80] Volker Weispfenning. Quantifier elimination for real algebra – the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997.

[81] Volker Weispfenning. A new approach to quantifier elimination for real algebra. In BobF. Caviness and JeremyR. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 376–392. Springer Vienna, 1998.

PART V

Appendices

APPENDIX A

Use Cases

A.1 The Rockwell Collins Triplex Voter

```
node top(input1, input2, input3: real) returns (output: real);
var
  equalized1,    equalized2,    equalized3 : real;
  equalization1, equalization2, equalization3 : real;
  satCentering, centering : real;
  df1, df2, df3 : real;
  st1, st2, st3 : real;
  c1, c2, c3 : bool;
  d1, d2, d3 : bool;
  po: bool;
let

  assert (input1 <  0.2);
  assert (input1 > -0.2);
  assert (input2 <  0.2);
  assert (input2 > -0.2);
  assert (input3 <  0.2);
  assert (input3 > -0.2);

  equalized1 = input1 - equalization1;
  df1 = equalized1 - output;
  st1 = if (df1 > 0.5) then 0.5 else (
    if (df1 < -0.5) then -0.5 else df1
  );
  equalization1 =
    0.0 -> pre (equalization1) + (pre (st1) - pre (satCentering))
       * 0.05;
```

```
equalized2 = input2 - equalization2;
df2 = equalized2 - output;
st2 = if (df2 > 0.5) then 0.5 else (
  if (df2 < -0.5) then -0.5 else df2
);
equalization2 =
  0.0 -> pre (equalization2) + (pre (st2) - pre (satCentering))
     * 0.05;

equalized3 = input3 - equalization3;
df3 = equalized3 - output;
st3 = if (df3 > 0.5) then 0.5 else (
  if (df3 < -0.5) then -0.5 else df3
);
equalization3 =
  0.0 -> pre (equalization3) + (pre (st3) - pre (satCentering))
     * 0.05;

c1 = equalized1 > equalized2;
c2 = equalized2 > equalized3;
c3 = equalized3 > equalized1;

output = if (c1 = c2) then equalized2 else (
  if (c2 = c3) then equalized3 else equalized1
);

d1 = equalization1 > equalization2;
d2 = equalization2 > equalization3;
d3 = equalization3 > equalization1;

centering = if (d1 = d2) then equalization2 else (
  if (d2 = d3) then equalization3 else equalization1
);
satCentering = if (centering > 0.25) then 0.25 else (
  if (centering < -0.25) then -0.25 else centering
);
```

```
  po = ( equalization1 <=  2.0 * 0.2) and
       ( equalization1 >= -2.0 * 0.2) and
       ( equalization2 <=  2.0 * 0.2) and
       ( equalization2 >= -2.0 * 0.2) and
       ( equalization3 <=  2.0 * 0.2) and
       ( equalization3 >= -2.0 * 0.2);

--%PROPERTY po;

tel
```

A.2 A Reconfiguration Logic System

```
node conf1(x:bool; n:int) returns (y: bool);
var
  pre_cpt: int;
  cpt: int;
let
  assert(0 <= pre_cpt and pre_cpt <= n);
  pre_cpt = 0 -> pre(cpt);
  cpt =
    if x then (
      if pre_cpt < n then pre_cpt+1 else pre_cpt
    ) else 0;
  y = cpt >= n;
tel

node latch(x: bool) returns (y: bool);
var
  pre_y: bool;
let
  pre_y = x -> pre(y);
  y = x or pre_y;
tel


node range_monitor(
```

```
  value, default_value, min, max:real; n:int
) returns (
  out_of_range, corrupt: bool; ranged_value:real
);
let
  out_of_range = value > max or value < min;
  corrupt = latch(conf1(out_of_range, n));
  ranged_value =
    if corrupt then default_value else (
      if value > max then max else (
        if value < min then min else value
      )
    );
tel

node priority(
  command, safe_command_value: real;
  command_failure, other_in_command: bool;
  n: int
) returns (
  safe_command: real; in_command: bool
);
let
  in_command =
    not command_failure and conf1(not other_in_command, n);
  safe_command =
    if in_command then command else safe_command_value;
tel


node system(
  sensor_value1, sensor_value2, sensor_value3: real
) returns (
  command, safe_command1, safe_command2, safe_command3: real;
  in_command1, in_command2, in_command3: bool;
  po: bool
);
```

```
var
  max, min, default_value, safe_value: real;
  n1, n2, n3, m1, m2, m3, m4: int;
  ranged_sensor1, ranged_sensor2, ranged_sensor3:real;
  out_of_range1,  out_of_range2,  out_of_range3: bool;
  corrupt1,       corrupt2,       corrupt3: bool;
  no_command: bool;
let
  max = 90.0;
  min = -90.0;
  default_value = 10.0;
  safe_value = 0.0;
  n1 = 10;
  n2 = n1+2;
  n3 = n1+5;
  m1 = 20;
  m2 = m1+2;
  m3 = m1+5;
  m4 = m3+12;
  (out_of_range1, corrupt1, ranged_sensor1) =
    range_monitor(sensor_value1, default_value, min, max, n1);
  (out_of_range2, corrupt2, ranged_sensor2) =
    range_monitor(sensor_value2, default_value, min, max, n2);
  (out_of_range3, corrupt3, ranged_sensor3) =
    range_monitor(sensor_value3, default_value, min, max, n3);

  (safe_command1,  in_command1) =
    priority(ranged_sensor1, safe_value, corrupt1, false, m1);
  (safe_command2,  in_command2) =
    priority(ranged_sensor2, safe_value, corrupt2, in_command1,
       m2);
  (safe_command3,  in_command3) =
    priority(ranged_sensor3, safe_value, corrupt3, in_command1 or
       in_command2, m3);

  command = safe_command1 +  safe_command2 +  safe_command3;
  no_command = not(in_command1 or in_command2 or in_command3);
```

```
  po = not conf1(no_command, m4);
  assert(not corrupt1 or not corrupt2 or not corrupt3);
  --%PROPERTY po;
tel
```